

A sudoku solver



John W. Shipman

2009-12-09 18:06

Abstract

Describes a program to solve Sudoku puzzles, using the Python programming language.

This publication is available in Web form¹ and also as a PDF document². Please forward any comments to tcc-doc@nmt.edu.

Table of Contents

1. Introduction	2
2. Encoding the puzzle	3
3. Operation of the solver	3
4. Design considerations	4
5. The <code>sudosolver.py</code> module: Puzzle-solving logic	5
5.1. Prologue	5
5.2. Manifest constants	5
5.3. <code>class SudokuSolver</code>	5
5.4. <code>SudokuSolver.__init__()</code> : Constructor	8
5.5. <code>SudokuSolver.__readPuzzle()</code> : Convert puzzle to internal form	8
5.6. <code>SudokuSolver.__readChar()</code> : Translate one input character	10
5.7. <code>SudokuSolver.get()</code> : Fetch one value from the board	10
5.8. <code>SudokuSolver.__rowColToX()</code> : Convert row and column to board position	11
5.9. <code>SudokuSolver.write()</code> : Output the state of the puzzle	11
5.10. <code>SudokuSolver.writeRow()</code> : Write one row of the puzzle	12
5.11. <code>SudokuSolver.solve()</code> : Find all solutions	13
5.12. <code>SudokuSolver.__reSolve()</code> : Recursive solver method	14
5.13. <code>SudokuSolver.__solution()</code> : Signal a solution	16
5.14. <code>SudokuSolver.__findPossibles()</code> : What digits are legal at a given position?	16
5.15. <code>SudokuSolver.__usedInRow()</code> : Eliminate digits by row	18
5.16. <code>SudokuSolver.__usedInColumn()</code> : Eliminate digits by column	19
5.17. <code>SudokuSolver.__usedInSubmat()</code> : Eliminate digits by submatrix	20
5.18. <code>SudokuSolver.__set()</code> : Store a value in a cell	21
5.19. <code>SudokuSolver.__xToRowCol()</code> : Translate board position to row and column	22
6. <code>sudoku</code> : Simple text-based solver	22
6.1. <code>sudoku</code> : Prologue	22
6.2. Imports	22
6.3. <code>main()</code> : Main procedure	23
6.4. <code>message()</code> : Write to the standard error stream	23
6.5. <code>fatal()</code> : Report a fatal error	23

¹ <http://www.nmt.edu/tcc/help/lang/python/examples/sudoku/>

² <http://www.nmt.edu/tcc/help/lang/python/examples/sudoku/sudoku.pdf>

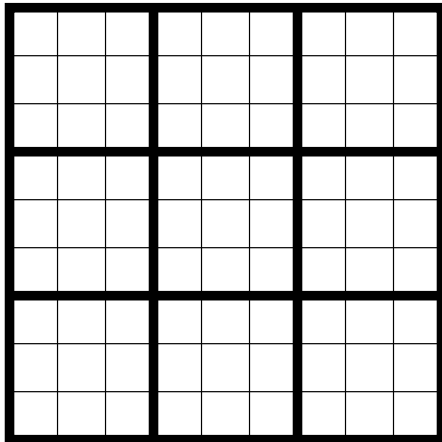
6.6. <code>solveFile()</code> : Solve one puzzle	24
6.7. <code>solutionFound()</code> : The observer callback for solutions	25
6.8. <i>sudoku</i> : Epilogue	26
7. Trace tables	26
7.1. Trace table: <code>SudokuSolver.__init__()</code>	26
7.2. Trace table: <code>SudokuSolver.__readPuzzle()</code>	28
7.3. Trace table: <code>SudokuSolver.get()</code>	31
7.4. Inspection: <code>SudokuSolver.write()</code>	32
7.5. Inspection: <code>SudokuSolver.writeRow()</code>	34

1. Introduction

This document describes the operation and implementation of a Python program to solve *sudoku* puzzles, a Japanese fad that has been recently caught on in the United States. The program is an illustration of “lightweight literate programming,” in which the program's executable code is embedded in its documentation. For an introduction to lightweight literate programming and numerous examples, see the author's *Lightweight literate programming* page³.

This project uses the Zero-defect or Cleanroom development methodology. For a discussion of this method, see the author's Cleanroom pages⁴.

The framework for a sudoku puzzle is a 9×9 grid. This grid is divided into nine 3×3 *submatrices*, and the lines between submatrices are thicker.



In an unsolved puzzle, some of the cells of this grid are filled in with digits from 1 through 9, and the rest of the cells are empty. The challenge is to place digits in the empty cells so that:

- Each digit appears exactly once in each row.
- Each digit appears exactly once in each column.
- Each digit appears exactly once in each 3×3 submatrix.

Files mentioned in this document are available online:

- `sudosolver.py`⁵: The module containing the solution logic.

³ <http://www.nmt.edu/~shipman/soft/litprog/>

⁴ <http://www.nmt.edu/~shipman/soft/clean/>

⁵ <http://www.nmt.edu/tcc/help/lang/python/examples/sudoku/sudosolver.py>

- `sudoku`: The plain-text driver.⁶
- `sudogui`: The GUI driver (not yet written).

2. Encoding the puzzle

The first step to getting a solution is to encode the initial state of the puzzle. Use an ordinary text editor to create a text file containing nine lines representing the nine rows of the puzzle.

On each line, type “.” for each empty cell, or the digit if the cell is initially filled in.

To make it easier to verify correct entry, and to give the puzzle more structure, you can add whitespace anywhere within a line, and you can add blank lines between rows.

Here is a file representing a complete puzzle. The spacing shown here is only a suggestion, but this spacing does make it easier to check the file against the original puzzle.

```
7 2 . . . 5 1 . .
. 1 5 9 8 . 4 . .
. 8 . . . 1 . . 6

. 4 . . 3 . 6 . 5
3 . . 6 . 2 . . 7
9 . 1 . 7 . . 2 .

5 . . 7 . . . 9 .
. . 2 . 9 6 3 5 .
. . 4 2 . . . 6 8
```

3. Operation of the solver

To solve one or more puzzles, run the `sudoku` command and supply one or more file names as command line arguments:

```
sudoku file ...
```

Typically a puzzle will have a single unique solution. However, `sudoku` will attempt to find and print all solutions. If the puzzle is misconstructured, there may be no solutions at all.

Here is the actual output of the script for an input file named `test.puz`. The input file is displayed, followed by any solutions found.

```
===== test.puz =====
5 . . 6 . . . 8 .
. . 1 5 . 9 2 . .
6 8 . 1 . . . . .

. 1 9 . . . . . 7
. . . 9 . 3 . . .
2 . . . . . 9 5 .

. . . . . 1 . 2 9
. . 4 3 . 7 5 . .
```

⁶ <http://www.nmt.edu/tcc/help/lang/python/examples/sudoku/sudoku>

```

. 6 . . . 5 . . 3
--- Solution #1:
5 9 7 6 3 2 1 8 4
3 4 1 5 8 9 2 7 6
6 8 2 1 7 4 3 9 5

4 1 9 2 5 6 8 3 7
8 7 5 9 1 3 6 4 2
2 3 6 7 4 8 9 5 1

7 5 3 8 6 1 4 2 9
1 2 4 3 9 7 5 6 8
9 6 8 4 2 5 7 1 3

Number of solutions found: 1
Elapsed cpu time: 0.52 seconds.

```

4. Design considerations

To prove that the program works, it is sufficient to print the solution or solutions in plain text. However, the author also envisions use of this script as a flashy GUI demo that shows the program's steps in solving the puzzle as it performs them.

Consequently, the solution will be divided into three major components:

- A module named `sudosolver.py` will encapsulate all the mechanics of solving the puzzle, but will not have any specific mechanism for input or output.
- A simple script named `sudoku` will use that module to find the solutions, and print them in plain-text format.
- A GUI-based script named `sudogui` will use the `sudosolver.py` module to display each solution graphically as the solver works on it, writing digits on the display when the program tries a step of the solution, and then erasing digits if they prove to be unworkable.

Clearly, the `sudosolver.py` module cannot do any output, since it doesn't know whether its output will go to a regular file stream or to a GUI. However, this module can't just return a solution when it gets one, because there may be multiple solutions. Also, how is it to handle the requirement that the GUI be notified to update its display whenever any digit in the puzzle gets placed or removed?

The solution is to use *callbacks*—that is, the module should be able to call a procedure outside itself to report solutions or state changes. This is a good example of the Observer pattern discussed in the classic work *Design Patterns* by Gamma et al. (ISBN 0-201-63361-2).

The `sudosolver.py` module contains a class named **SudokuSolver** that encapsulates the puzzle-solving logic. This class has two Observer callbacks; this allows us to decouple the input/output considerations from the logic.

- One Observer is called whenever a solution is found.
- Another, optional Observer is called whenever the puzzle's state changes.

5. The `sudosoLver.py` module: Puzzle-solving logic

This module contains the `SudokuSolver` class that encapsulates the logic for solving the puzzle.

Each module's semantics are described by an *intended function*. Intended function notation is a core principle of the Cleanroom methodology⁷.

5.1. Prologue

The `sudosoLver.py` module starts with a nominal documentation string that points back to this documentation.

```
sudosoLver.py
```

```
"""sudosoLver.py: Sudoku puzzle solver

For documentation, see:
    http://www.nmt.edu/tcc/help/lang/python/examples/sudoku/
"""
```

5.2. Manifest constants

First we define some constants needed everywhere in the solution.

```
sudosoLver.py
```

```
#=====
# Manifest constants
#-----
```

SUBMAT_L

Size of a submatrix.

```
sudosoLver.py
```

```
SUBMAT_L = 3
```

MAT_L

Dimensions of the matrix (9).

```
sudosoLver.py
```

```
MAT_L = SUBMAT_L ** 2
```

BOARD_L

Size of the list representing the entire matrix (81).

```
sudosoLver.py
```

```
BOARD_L = MAT_L ** 2
```

EMPTY

The value used to represent empty cells.

```
sudosoLver.py
```

```
EMPTY = 0
```

5.3. `class SudokuSolver`

Here is the interface to the `SudokuSolver` class.

⁷ <http://www.nmt.edu/~shipman/soft/clean/>

```
# - - - - - class SudokuSolver - - - - -
class SudokuSolver:
    """Represents one sudoku puzzle.
```

The constructor takes three arguments:

givenPuzzle

The initial puzzle as a string. The format of this string is the same as the format of the file described in Section 2, "Encoding the puzzle" (p. 3). You can use the `.read()` method on a file object to read the entire file and store it in a string.

solutionObserver

An Observer function that will be called whenever the **SudokuSolver** instance detects a complete solution to the puzzle. Write the procedure **S** with this calling sequence:

```
S ( solver )
```

where the **solver** argument will be the **SudokuSolver** instance.

changeObserver

An Observer function that will be called whenever a cell of the puzzle changes during the solution process. Write this procedure **C** with calling sequence:

```
C ( solver, row, col, state )
```

solver

The **SudokuSolver** instance after the state change.

row

The row index of the cell that changed, in the interval [0,8].

col

The column index of the cell that changed, in the interval [0,8].

state

The new state of the cell, an integer in the interval [0,9]. A value of 0 indicates that the cell is being set back to empty.

Here is the intended function for the constructor. Note that it raises a **ValueError** exception if the **givenPuzzle** argument is not a correctly formed puzzle.

```
Exports:
SudokuSolver ( givenPuzzle, solutionObserver=None,
               changeObserver=None ):
[ (givenPuzzle is a sudoku puzzle as a string) and
  (solutionObserver is a procedure or None) and
  (changeObserver is a procedure or None) ->
  if givenPuzzle is a valid sudoku puzzle ->
    return a SudokuSolver object representing that
    puzzle in its initial state
  else ->
    raise ValueError ]
```

Methods and attributes of the **SudokuSolver** include:

```
.givenPuzzle:      [ as passed to constructor, read-only ]
.solutionObserver: [ as passed to constructor, read-only ]
.changeObserver:   [ as passed to constructor, read-only ]
.solve()
  [ call self.changeObserver (if any) for every change
    to a cell value and call self.solutionObserver for
    every solution of self ]
```

The **.get()** method is used to query the state of one cell of the puzzle.

```
.get(r, c):
  [ r and c are integers ->
    if (0<=r<MAT_L) and (0<=c<MAT_L) ->
      return the state of the cell at row r and column c
      as 0 for empty, or an integer in [1,9] if set
    else -> raise KeyError ]
```

As a convenience for a quick display of the state of a puzzle (initially or at solution time), we provide a **.write()** function that displays the puzzle in the same format as the input files (with our recommended extra spaces added).

```
.write(outFile):
  [ outFile is a writeable file ->
    outFile += a representation of self's state in
    input-file format ]
```

Two more attributes accumulate statistics that may be of interest to the caller after the **.solve()** method has returned:

```
State/Invariants:
.nStateChanges:   [ number of cell state changes so far ]
.nSolutions:      [ number of solutions seen so far ]
```

Here are the internal attributes of the instance. The **.__given** attribute holds a copy of the puzzle in its initial state, so we can determine whether a cell's value was set initially or added in the solving process. The **.__board** attribute holds the current state of the puzzle.

```
.__given:
  [ the initial state of the puzzle as a list of integers,
    0 for empty, or in [1,9] if set ]
.__board:
  [ the current state of the puzzle as a list of integers,
    0 for empty, or in [1,9] if set ]
"" ""
```

Both these attributes represent the puzzle as a list of 81 integers. In this list, 0 represents a blank cell, and integers 1 through 9 represent the digit in a cell that has been filled in. The mapping from row and column indices (**R**, **C**) to indices **X** in these lists works like this:

- To convert from board index to row and column:

```
R, C = divmod ( X, 9 )
```

- To convert from row and column to board index:

$$X = (R * 9) + C$$

5.4. SudokuSolver.__init__(): Constructor

The constructor's main job, other than saving its argument values, is to translate the puzzle from its external form (as described in Section 2, “Encoding the puzzle” (p. 3)) to its internal form as a list of 81 integers.

```
sudosolver.py
```

```
# - - -   S u d o k u S o l v e r .   _ _   i n i t   _ _   - - -

def __init__ ( self, givenPuzzle, solutionObserver=None,
              changeObserver=None ):
    """Constructor for SudokuSolver."""
```

First we save the constructor arguments and initialize the various data structures.

```
sudosolver.py
```

```
#-- 1 --
self.givenPuzzle      = givenPuzzle
self.solutionObserver = solutionObserver
self.changeObserver   = changeObserver
self.nStateChanges    = 0
self.nSolutions       = 0
```

The next step is to convert the puzzle from its external form to its internal form, or raise **ValueError** if it is invalid. See Section 5.5, “**SudokuSolver.__readPuzzle():** Convert puzzle to internal form” (p. 8).

```
sudosolver.py
```

```
#-- 2 --
# [ self.givenPuzzle is a string ->
#   if self.givenPuzzle is a valid sudoku puzzle as a string ->
#     self.__board := a list of BOARD_L integers
#                   representing that puzzle
#   else -> raise ValueError ]
self.__board = self.__readPuzzle()
```

We need to make a copy of **self.__board** so we can remember what the puzzle looked like initially. We could use Python's **copy** module to make the copy, but a “slice of the whole” works just as well.

```
sudosolver.py
```

```
#-- 3 --
# [ self.__given := a copy of self.__board ]
self.__given = self.__board[:]
```

For trace table verification, see Section 7.1, “Trace table: **SudokuSolver.__init__()**” (p. 26).

5.5. SudokuSolver.__readPuzzle(): Convert puzzle to internal form

This method parses the puzzle in its external form and converts it to a list of integers.

```
# - - -   S u d o k u S o l v e r . _ _ r e a d P u z z l e   - - -

def __readPuzzle ( self ):
    """Translate the puzzle from external to internal form.

    [ self.givenPuzzle is a string ->
      if self.givenPuzzle is a valid sudoku puzzle ->
        return a list of BOARD_L integers representing
        that puzzle
      else -> raise ValueError ]
    """
```

Converting the puzzle from string form to a vector of numbers is pretty straightforward. If we ignore all whitespace in the input file, what is left should be a string of exactly **BOARD_L** characters, each of which should be "." or a digit in string form.

First we smash out all the whitespace and make **self.givenPuzzle** into a single string. This can be done in one line as a list comprehension⁸:

```
#-- 1 --
# [ charList := a list whose elements are the
#   non-whitespace characters of self.givenPuzzle in
#   the same order ]
charList = [ c
             for c in list(self.givenPuzzle)
             if not c.isspace() ]
```

At this point we can test for the correct number of characters.

```
#-- 2 --
if len(charList) != BOARD_L:
    raise ValueError, ( "Puzzle has %d nonblank "
                       "characters; it should have exactly %d." %
                       (len(charList), BOARD_L) )
```

All that remains is to check each character for validity and convert them to their integer value. This step raises **ValueError** if any of the characters are not either "." or a digit. See Section 5.6, "**SudokuSolver.__readChar()**: Translate one input character" (p. 10).

```
#-- 3 --
# [ if each element of charList is either "." or in
#   the interval ["1", "9"] ->
#   result := a list of integers corresponding to the
#   elements of charList consisting of integer 0
#   where the value is "." and an integer in [1,9]
#   where the value is a digit
#   else ->
#   raise ValueError ]
result = [ self.__readChar ( c )
           for c in charList ]
```

⁸ <http://docs.python.org/ref/lists.html>

```
#-- 4 --
return result
```

For trace table verification, see Section 7.2, “Trace table: `SudokuSolver.__readPuzzle()`” (p. 28).

5.6. `SudokuSolver.__readChar()`: Translate one input character

This method checks one of the non-whitespace characters in the input for validity. It returns 0 for ".", and the equivalent integer for digits. All other values fail.

sudosolver.py

```
# - - -   S u d o k u S o l v e r . _ _ r e a d C h a r   - - -

def __readChar ( self, c ):
    """Translate one input character.

    [ c is a one-character string ->
      if c is "." ->
        return 0
      else if c is a digit in ["1", "9"] ->
        return int(c)
      else -> raise ValueError ]
    """
    if c == ".":
        return 0
    elif "1" <= c <= "9":
        return int(c)
    else:
        raise ValueError, ( "Invalid sudoku character: '%s'" %
                             c )
```

There is no trace table for this method. Verification is by inspection.

5.7. `SudokuSolver.get()`: Fetch one value from the board

This method retrieves the contents of one cell as an integer.

sudosolver.py

```
# - - -   S u d o k u S o l v e r . g e t   - - -

def get ( self, r, c ):
    """Get the cell at row r, column c.
    """
```

First we check the row index `r` and column index `c`, and raise **KeyError** if either is out of bounds.

sudosolver.py

```
#-- 1 --
# [ if r is in [0,MAT_L) ->
#   I
#   else -> raise KeyError ]
if not ( 0 <= r < MAT_L ):
    raise KeyError, ( "SudokuSolver.get(): Bad row "
                     "index, %d." % r )
```

```

#-- 2 --
# [ if c is in [0,MAT_L) ->
#   I
#   else -> raise KeyError ]
if not ( 0 <= c < MAT_L ):
    raise KeyError, ( "SudokuSolver.get(): Bad column "
                    "index, %d." % c )

```

To convert a row and column number to an index in the **self.__board** structure, we use Section 5.8, “**SudokuSolver.__rowColToX()**: Convert row and column to board position” (p. 11).

sudosolver.py

```

#-- 3 --
# [ x := index in self.__board corresponding to row r,
#   column c ]
x = self.__rowColToX ( r, c )

```

Finally, return the cell's value.

sudosolver.py

```

#-- 4 --
return self.__board[x]

```

For trace table verification, see Section 7.3, “Trace table: **SudokuSolver.get()**” (p. 31).

5.8. SudokuSolver.__rowColToX(): Convert row and column to board position

This method converts a cell position as a row and column index to the corresponding position in the **self.__board** list.

sudosolver.py

```

# - - -   S u d o k u S o l v e r . _ _ r o w C o l T o X   - - -

def __rowColToX ( self, r, c ):
    """Convert row/column indices to board index.

    [ r and c are integers in [0,8] ->
      return the index in self.__board corresponding to
        that row and column ]
    """
    return ( r * MAT_L ) + c

```

There is no trace table for this method. Verification is by inspection. The transforms between (row,column) and indices in **self.__board** are discussed in Section 5.3, “**class SudokuSolver**” (p. 5).

5.9. SudokuSolver.write(): Output the state of the puzzle

sudosolver.py

```

# - - -   S u d o k u S o l v e r . w r i t e   - - -

def write ( self, outFile ):
    """Write the puzzle state as plain text.

```

Output format (minus the enclosing box):

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

To generate the output, we run two stacked loops.

- The outer loop iterates over the rows of the puzzle. It adds an extra blank line between rows 2 and 3 and between rows 5 and 6 (counting from zero).
- The inner loop iterates over the columns. It adds a space between each item, and adds extra spaces after columns 2 and 5. That loop resides in Section 5.10, “**SudokuSolver.writeRow()**: Write one row of the puzzle” (p. 12).

sudosolver.py

```
#-- 1 --
for rowx in range(MAT_L):
    #-- 1 body --
    # [ rowx is in [0,MAT_L) ->
    #     outFile += a representation of row rowx of self ]

    #-- 1.1 --
    # [ if ( ( rowx > 0 ) and
    #       ( rowx % SUBMAT_L ) == 0 ) ) ->
    #     outFile += an empty line
    # else -> I ]
    if ( ( rowx > 0 ) and
        ( ( rowx % SUBMAT_L ) == 0 ) ):
        print >> outFile

    #-- 1.2 --
    # [ outFile += a representation of row rowx of self ]
    self.writeRow ( outFile, rowx )
```

For a discussion of the verification of this method, see Section 7.4, “Inspection: **SudokuSolver.write()**” (p. 32).

5.10. **SudokuSolver.writeRow()**: Write one row of the puzzle

This function writes one row of the puzzle's state to the given **outFile**. For the overall output design, see Section 5.9, “**SudokuSolver.write()**: Output the state of the puzzle” (p. 11).

```

# - - -   S u d o k u S o l v e r . w r i t e R o w   - - -

def writeRow ( self, outFile, rowx ):
    """Display one row in plain text.

    [ (outFile is a writeable file) and
      (rowx is in [0,MAT_L) ->
        outFile += a representation of row rowx of self ]
    """

    #-- 1 --
    for colx in range(MAT_L):
        #-- 1 body --
        # [ sys.stdout += a representation of the cell at
          #       row rowx and column colx ]

        #-- 1.1 --
        # [ c := self's cell at (rowx,colx) in character
          #       form ]
        cell = self.get ( rowx, colx )
        if cell == 0: c = '.'
        else:         c = chr ( ord ( '0' ) + cell )

        #-- 1.2 --
        # [ sys.stdout += c ]
        if ( ( colx > 0 ) and
            ( ( colx % SUBMAT_L ) == 0 ) ):
            print >>outFile, " ",
            print >>outFile, c,

```

Having written the line's contents, it is now our job to terminate it.

```

#-- 2 --
print >>outFile

```

For a discussion of the verification of this method, see Section 7.5, "Inspection: **SudokuSolver.writeRow()**" (p. 34).

5.11. SudokuSolver.solve(): Find all solutions

This is the meat of the class: the method that tries to solve the puzzle.

```

# - - -   S u d o k u S o l v e r . s o l v e   - - -

def solve ( self ):
    """Find all solutions to the puzzle.
    """

```

There are doubtless many approaches to solving the puzzle, but the author's preferences are well expressed in this quotation from a former professor in Computer Science at New Mexico Tech:

There are very few problems in computer science that are not susceptible to brute force.

So, consider the board as a list of 81 integers, some of which are nonzero (the initial clues) and some of which are zero (the empty cells). One approach to solving the problem works like this: we work from the start of the board to the end, and at each cell we do this:

1. If the cell already has a value in it, move one to the right. If that takes us out of the board at the far end, this is a solution—call the **self.solutionObserver** callback to report a solution.
2. If the cell doesn't have a value in it, see what values are still legal given the current state of the board. This is done by starting with a full set of all nine digits, then eliminating any that occur elsewhere in that row, elsewhere in that column, or elsewhere in that submatrix.
3. If the set of legal values computed in the previous step is empty, there cannot be any solution with the board in this state. Move one to the left; if that takes us before cell 0, we are done.
4. For each of the set of legal values, put that value in the current cell and move one to the right. When the process returns to this cell, put the next legal value in and repeat until all values are gone.

Rather than using clunky iterative techniques to handle moving right and left on the board, this approach seems to call for recursion. We can reframe the problem of solving the whole puzzle as a call to a recursive method that solves the puzzle starting at position 0. Assuming there are any legal choices at position 0, we put each legal choice into the cell at position 0, then recur to solve the puzzle starting at position 1. The basis case that stops recursion is an attempt to solve the puzzle starting at the last-plus-one position (index **BOARD_L**). Recursion also stops when there are no legal moves at the current position.

So, deferring all the ugly logic to the recursive method, here is the body of the **.solve()** method. The recursive method is described in Section 5.12, “**SudokuSolver.__reSolve()**: Recursive solver method” (p. 14).

sudosolver.py

```
# - - 1 - -
# [ call self.solutionObserver for all solutions of
#   self.__board and call self.changeObserver (if any)
#   for all state changes in finding those solutions ]
self.__reSolve ( 0 )
```

5.12. SudokuSolver.__reSolve(): Recursive solver method

This method is the recursive part of the solution mechanism. It is given a starting position on the board, and tries to find digits that will work at that position. If there are any such digits, it places each at the current position, and then recurs to solve the board starting at the next position.

sudosolver.py

```
# - - - S u d o k u S o l v e r . _ _ r e S o l v e - - -

def __reSolve ( self, x ):
    """Recursively solve the puzzle.

    [ (self.__board has no blank cells before position x) and
      (x is an integer) ->
      if x >= BOARD_L ->
          call self.solutionObserver for the current state
          of self.__board
      else if x is in [0,BOARD_L) ->
          call self.solutionObserver for all solutions of
          self.__board starting at position x, and call
```

```

        self.changeObserver (if any) for all state changes
        in finding those solutions ]
    """

```

As with any good recursive function, we check for the basis case first. If the index **x** is past the end of the board, then the current state of the board is a solution. See Section 5.13, “**SudokuSolver.__solution()**: Signal a solution” (p. 16) for the actions taken on detecting a solution.

sudosolver.py

```

#-- 1 --
# [ if x >= BOARD_L ->
#     call self.solutionObserver
#     return
# else -> I ]
if x >= BOARD_L:
    self.__solution()
    return

```

If this position on the board is one of the original clues (that is, if **self.__given[x]** is nonzero), we recur to solve the rest of the board, and then return.

sudosolver.py

```

#-- 2 --
# [ if self.__given[x] != EMPTY ->
#     call self.solutionObserver for all solutions of
#     self.__board starting at position x+1, and call
#     self.changeObserver (if any) for all state changes
#     in finding those solutions
#     return
# else -> I ]
if self.__given[x] != EMPTY:
    self.__reSolve ( x + 1 )
    return

```

At this point, we know that position **x** is currently empty. We look around the board to see what digits are still legal here, if any. For that logic, see Section 5.14, “**SudokuSolver.__findPossibles()**: What digits are legal at a given position?” (p. 16).

sudosolver.py

```

#-- 3 --
# [ possibles := list of all digits that are not
#     duplicate in the same row, column, or submatrix
#     as cell x ]
possibles = self.__findPossibles ( x )

```

Our caller may have left us with a board in which there is no possible digit that fits at position **x**. In that case, the **possibles** list will be empty. If there are one or more possible digits, though, we must try solving the puzzle for each possibility. So, for each element of **possibles**, we store that element in cell **x** (see Section 5.18, “**SudokuSolver.__set()**: Store a value in a cell” (p. 21)), and recursively try to solve the rest of the puzzle.

One very important part of this step is that we must set cell **x** back to empty before returning to the caller. We wouldn't want to set it empty if the cell is one of the original clues, but that case was eliminated above.

```

#-- 4 --
# [ call self.solutionObserver for all solutions of
#   self.__board starting at position x+1 with cell x
#   set to each member of possibles, and call
#   self.changeObserver (if any) for all state changes
#   in finding those solutions ]
for trial in possibles:
    #-- 4 body --
    # [ call self.solutionObserver for all solutions of
    #   self.__board starting at position x+1 with cell
    #   x set to trial, and call self.changeObserver
    #   (if any) for all state changes in finding those
    #   solutions ]
    self.__set ( x, trial )
    self.__reSolve ( x + 1 )

#-- 5 --
self.__set ( x, EMPTY )

```

5.13. SudokuSolver.__solution(): Signal a solution

This method is called when the puzzle has been solved. It has two jobs: call the solution observer callback, and increment the number of solutions found.

```

# - - -   S u d o k u S o l v e r . _ _ s o l u t i o n   - - -

def __solution ( self ):
    """A solution has been found.

    [ if self.solutionObserver is not None ->
      call self.solutionObserver(self)
      in any case ->
        self.nSolutions += 1 ]
    """

    #-- 1 --
    self.nSolutions += 1

    #-- 2 --
    if self.solutionObserver is not None:
        self.solutionObserver(self)

```

5.14. SudokuSolver.__findPossibles(): What digits are legal at a given position?

For a given position **x** in **self.__board**, this method returns a list of the digits that can legally occupy that position, given the current state of the board.

```
# - - - S u d o k u S o l v e r . _ _ f i n d P o s s i b l e s

def __findPossibles ( self, x ):
    """What digits are legal at position x on the board?

    [ x is an integer in [0,BOARD_L) ->
      return a list of digits not found in the row,
      column, or submatrix containing position x ]
    """
```

We'll divide the work up into three pieces corresponding to the three rules of the game: elimination by row, elimination by column, and elimination by submatrix.

For each rule, we'll call a method that returns a nine-element list describing the digits that are eliminated because they are used elsewhere in the row, column, or submatrix. In these lists, position **[0]** corresponds to the digit 1, position **[1]** to the digit 2, and so on. Each position's value is 1 if the digit is used elsewhere, 0 if it isn't. In the intended functions here, we use the term "bitmap" to signify this convention.

```
#-- 2 --
# [ rowElim := a bitmap of the digits that occur
#           elsewhere in the same row as position x ]
rowElim = self.__usedInRow ( x )

#-- 3 --
# [ colElim := a bitmap of the digits that occur
#           elsewhere in the same column as position x ]
colElim = self.__usedInColumn ( x )

#-- 4 --
# [ subElim := a bitmap of the digits that occur
#           elsewhere in the same submatrix as position x ]
subElim = self.__usedInSubmat ( x )
```

For the subsidiary routines, see:

- Section 5.15, "**SudokuSolver.__usedInRow()**: Eliminate digits by row" (p. 18)
- Section 5.16, "**SudokuSolver.__usedInColumn()**: Eliminate digits by column" (p. 19)
- Section 5.17, "**SudokuSolver.__usedInSubmat()**: Eliminate digits by submatrix" (p. 20)

Once we have all three lists, we will use the Boolean "or" operator (**|**) to derive a master list of the digits that are eliminated—values of 1 again signifying elimination.

```
#-- 5 --
# [ elim := (rowElim | colElim | subElim) ]
elim = [ rowElim[x] | colElim[x] | subElim[x]
        for x in range(MAT_L) ]
```

Then we'll use that list to build a list of the digits that are not eliminated.

```
#-- 6 --
# [ return a list containing digits in [1,9]
#   corresponding to zero elements of elim ]
result = [ i+1
```

```

        for i in range(0,MAT_L)
        if elim[i]==0 ]

#-- 7 --
return result

```

5.15. SudokuSolver. __usedInRow(): Eliminate digits by row

The purpose of this method is to scan the row that contains a given board position x , look for digits used in that row at other positions, and return a “bitmap”: a nine-element list whose elements correspond to the nine digits in [1,9], such that each element is 1 if that digit is used elsewhere in the row, and 0 otherwise.

sudosolver.py

```

# - - - S u d o k u S o l v e r . _ _ u s e d I n R o w - - -

def __usedInRow ( self, x ):
    """What digits are used elsewhere in the row containing x?

    [ x is an integer in [0,BOARD_L) ->
      return a 9-element bitmap whose elements are true
      iff the corresponding digit is used elsewhere in
      the row containing board position x ]
    """

```

First we figure out which row contains board index x . Then we set up the initial bitmap as a list of zeroes.

sudosolver.py

```

#-- 1 --
# [ rx      := the row index of board position x
#   cx      := the column index of board position x
#   result  := a list of MAT_L zeroes ]
rx, cx = self.__xToRowCol ( x )
result = [0] * MAT_L

```

As we scan the row, it is important to skip over the column containing x . The process of eliminating possibilities for a given cell must not depend on the state of the cell under consideration.

sudosolver.py

```

#-- 2 --
# [ result := result with elements corresponding to
#   digits found in row rx (except for column cx) ]
for col in range(MAT_L):
    #-- 2 body --
    # [ if (col != cx) and
    #   (self has a nonzero value D in cell (rx, col) ->
    #   result[D-1] := 1
    #   else ->
    #   I ]
    if col != cx:
        cell = self.get(rx, col)
        if cell != EMPTY:
            result[cell-1] = 1

```

```

#-- 3 --
return result

```

5.16. SudokuSolver.__usedInColumn(): Eliminate digits by column

This routine is structurally very similar to Section 5.15, “SudokuSolver.__usedInRow(): Eliminate digits by row” (p. 18). The only difference is that we step through the rows of the column containing **x**, skipping the row containing **x**.

sudosolver.py

```

# - - -   S u d o k u S o l v e r . _ _ u s e d I n C o l u m n

def __usedInColumn ( self, x ):
    """What digits are used elsewhere in the column containing x?

    [ x is an integer in [0,BOARD_L) ->
      return a 9-element bitmap whose elements are true
      iff the corresponding digit is used elsewhere in
      the column containing board position x ]

    """

    #-- 1 --
    # [ rx      := the row index of board position x
    #   cx      := the column index of board position x
    #   result  := a list of MAT_L zeroes ]
    rx, cx = self.__xToRowCol ( x )
    result = [0] * MAT_L

```

Again, we skip over cell **x** so that our result doesn't depend on the state of the cell currently under consideration.

sudosolver.py

```

#-- 2 --
# [ result := result with elements corresponding to
#   digits found in column cx (except for row rx) ]
for row in range(MAT_L):
    #-- 2 body --
    # [ if (row != rx) and
    #   (self has a nonzero value D in cell (rx, col) ->
    #     result[D-1] := 1
    #   else ->
    #     I ]
    if row != rx:
        cell = self.get(row, cx)
        if cell != EMPTY:
            result[cell-1] = 1

#-- 3 --
return result

```

5.17. SudokuSolver.__usedInSubmat(): Eliminate digits by submatrix

This method inspects the submatrix containing board location **x** and returns a bitmap with 1 elements corresponding to digits used elsewhere in that submatrix.

sudosolver.py

```
# - - - S u d o k u S o l v e r . _ _ u s e d I n S u b m a t r i x

def __usedInSubmat ( self, x ):
    """What digits are used in the submatrix containing x?

    [ x is an integer in [0, BOARD_L) ->
      return a 9-element bitmap whose elements are true
      iff the corresponding digit is used elsewhere in
      the submatrix containing board position x ]
    """
```

The **result** bitmap is initialized to nine zeroes. Then we set **rx** and **cx** to the row and column index corresponding to board position **x**. Then we set **rSub** and **cSub** to the coordinates of the upper left cell of the submatrix.

sudosolver.py

```
#-- 1 --
# [ result := a list of MAT_L zeroes
#   rx     := index of the row containing board
#           position x
#   cx     := index of the column containing board
#           position x
result = [0] * MAT_L
rx, cx = self.__xToRowCol ( x )

#-- 2 --
# [ rSub := index of the row containing the upper left
#   cell of the submatrix containing row rx
#   cSub := index of the column containing the upper left
#   cell of the submatrix containing column cx ]
rSub = (rx / 3) * 3
cSub = (cx / 3) * 3
```

We iterate over the elements of the submatrix, setting elements of the **result** bitmap to 1 wherever we find digits. Again, it is important not to look at the cell at (**rx**, **cx**), so we don't have to depend on that cell being in any known state.

sudosolver.py

```
#-- 3 --
# [ result := result with its elements set to 1 at
#   each position corresponding to a digit that
#   occurs in the submatrix whose upper left corner
#   is at row rSub, column cSub, ignoring the cell at
#   row rx, column cx ]
for rowx in range(rSub, rSub+SUBMAT_L):
    for colx in range(cSub, cSub+SUBMAT_L):
        #-- 3 body --
        # [ if (rowx != rx) or (colx != cx) and
        #   self's cell value V at row rowx, column colx is
        #   not EMPTY ->
```

```

#     result[V-1] := 1
#     else -> I ]
if ( ( rowx != rx ) or ( colx != cx ) ):
    cell = self.get(rowx, colx)
    if cell != EMPTY:
        result [ cell-1 ] = 1

#-- 4 --
return result

```

5.18. SudokuSolver. __set(): Store a value in a cell

This method changes the value of a cell in the puzzle. It also notifies the state change observer callback if there is one.

sudosolver.py

```

# - - - S u d o k u S o l v e r . _ _ s e t - - -

def __set ( self, x, value ):
    """Set or clear one cell of the board.

    [ (x is an integer in [0,BOARD_L)) and
      (value is an integer in [0,MAT_L]) ->
      if self.changeObserver is not None ->
          notify self.changeObserver that cell x is being
          set to value
      in any case ->
          self.__board[x] := value ]
    """

```

The interface specification stipulates that the change observer callback is called *after* the state has changed, so our first task is to store the new value in the board. We also track the number of state changes in **self.nStateChanges**.

sudosolver.py

```

#-- 1 --
self.__board[x] = value
self.nStateChanges += 1

```

If there is a change observer callback, we have to translate the internal index **x** to row and column values. See Section 5.19, “**SudokuSolver.__xToRowCol()**: Translate board position to row and column” (p. 22).

sudosolver.py

```

#-- 2 --
# [ if self.changeObserver is not None ->
#     notify self.changeObserver that cell x is being
#     set to value
if self.changeObserver is not None:
    row, col = self.__xToRowCol ( x )
    self.changeObserver ( self, row, col, value )

```

5.19. SudokuSolver.__xToRowCol(): Translate board position to row and column

This method translates an internal position in `self.__board` to row and column indices.

sudosolver.py

```
# - - -   S u d o k u S o l v e r . _ _ x T o R o w C o l   - - -  
  
def __xToRowCol ( self, x ):  
    """Translate board index to row and column indices.  
  
    [ x is an integer in [0,BOARD_L) ->  
      return (r,c) where r is x's row and c is x's column ]  
    """  
    return divmod ( x, 9 )
```

6. sudoku: Simple text-based solver

The `sudoku` script is a simple driver for the `sudosolver.py` module. There's not much to it: the command line arguments are the names of puzzles in input form. It reads each puzzle, passes it to the solver object, and waits for callbacks when (if) the puzzle is solved.

6.1. sudoku: Prologue

The executable code for the `sudoku` script starts with the usual "pound-bang line" to make it executable under Unix, followed by a brief comment pointing to this documentation.

sudoku

```
#!/usr/bin/env python  
#=====  
# Sudoku puzzle solver.  For documentation, see:  
#   http://www.nmt.edu/tcc/help/lang/python/sudoku/  
#-----
```

6.2. Imports

We'll need the standard `sys` module to access the standard output and error streams and `sys.exit()`. We'll also import the `time` module so we can get solution timings.

sudoku

```
#=====  
# Imports  
#-----  
  
import sys  
import time
```

We must also import the solver module.

sudoku

```
import sudosolver
```

6.3. main () : Main procedure

The main iterates over the command line arguments. As a courtesy to undertrained users, we complain if there are no arguments at all. Each puzzle file name in turn is sent to Section 6.6, “**solveFile()**: Solve one puzzle” (p. 24) for solution.

sudoku

```
# - - -   m a i n   - - -

def main():
    """sudoku main program
    """

    #-- 1 --
    argList = sys.argv[1:]

    #-- 2 --
    # [ if argList is empty ->
    #     sys.stderr += error message
    # else ->
    #     sys.stdout += solutions to valid puzzles named in
    #         argList
    #     sys.stderr += error messages about invalid puzzles
    #         named in argList ]
    if len(argList) == 0:
        message ( "*** You must supply names of "
                  "at least one sudoku puzzle file.\n" )
    else:
        for arg in argList:
            solveFile ( arg )
```

6.4. message () : Write to the standard error stream

Writes a message to the standard error stream. One or more strings are supplied as arguments.

sudoku

```
# - - -   m e s s a g e   - - -

def message ( * L ):
    """Write a message to sys.stderr.

    [ L is a list of strings ->
      sys.stderr += concatenation of L ]
    """
    sys.stderr.write ( "*** %s\n" % "".join(L) )
```

6.5. fatal () : Report a fatal error

Write a message and die. The arguments are a list of messages in string form.

sudoku

```
# - - -   f a t a l   - - -

def fatal ( *L ):
```

```

"""Write a message and terminate.

[ L is a list of strings ->
  sys.stderr += concatenation of L
  stop execution ]
"""
message ( *L )
sys.exit(1)

```

6.6. solveFile(): Solve one puzzle

This function reads one puzzle and feeds it to the solver. The output comes back through an Observer function that is called whenever a solution is found.

sudoku

```

# - - -   s o l v e F i l e   - - -

def solveFile ( fileName ):
    """Try to solve one puzzle.

    [ fileName is a string ->
      if fileName names a readable, valid sudoku puzzle
      file ->
        sys.stdout += solution(s) to that puzzle if any
        sys.stderr += message if there are no solutions
      else ->
        sys.stderr += error message about a bad puzzle ]
    """

```

First we'll try to open and read the file all into a single string, which is how the **SudokuSolver** wants it.

sudoku

```

#-- 1 --
# [ if fileName names a readable file ->
#   rawPuzzle := contents of that file
#   else ->
#     sys.stderr += error message
#     return ]
try:
    inFile = open ( fileName )
    rawPuzzle = inFile.read()
except IOError, detail:
    message ( "*** Can't open file '%s' for reading. %s\n" %
              (fileName, detail) )
    return

```

Next we instantiate a **SudokuSolver** and feed it the puzzle. If any solutions are found, they will be reported by a callback to Section 6.7, "**solutionFound()**: The observer callback for solutions" (p. 25).

sudoku

```

#-- 2 --
# [ if rawPuzzle is a valid sudoku puzzle ->
#   solver := an instance of SudokuSolver to solve
#   rawPuzzle that will call solutionFound when it

```

```

#         finds a solution
#     else ->
#         sys.stderr += error message
#         return ]
try:
    solver = sudosolver.SudokuSolver ( rawPuzzle, solutionFound )
except ValueError, detail:
    message ( "Invalid puzzle: %s" % detail )
    return

```

Next, we echo the puzzle's name and content.

```

#-- 3 --
# [ sys.stdout += (fileName) + (puzzle in its initial
#         state ]
separator = "=" * 9
print "\n%s %s %s" % (separator, fileName, separator)
solver.write ( sys.stdout )

```

sudoku

Now we ask the solver to solve. If any solutions are found, they'll be written in this step.

```

#-- 4 --
# [ sys.stdout += solutions to solver, if any ]
startClock = time.clock()
solver.solve()
endClock = time.clock()

```

sudoku

Lastly, we report the number of solutions found. If the puzzle is properly constructed, this step should display a solution count of one. A value of zero means there was no solution.

```

#-- 5 --
# [ sys.stdout += report on the number of solutions in
#         solver ]
print "\nNumber of solutions found:", solver.nSolutions
print "Elapsed cpu time: %.2f seconds." % (endClock-startClock)

```

sudoku

6.7. solutionFound(): The observer callback for solutions

This function is called by the **SudokuSolver** instance whenever it finds a solution to a puzzle.

```

# - - -   s o l u t i o n F o u n d   - - -
def solutionFound ( solver ):
    """Report a successful solution of the puzzle.

    [ sys.stdout += display of the state of solver ]
    """
    print "\n--- Solution #d:" % solver.nSolutions
    solver.write ( sys.stdout )

```

sudoku

6.8. *sudoku*: Epilogue

These lines at the tail end of *sudoku* execute the main.

sudoku

```
#=====
# Epilogue
#-----

if __name__ == "__main__":
    main()
```

7. Trace tables

One important technique from the Cleanroom methodology⁹ is the use of *trace tables* to assist the verification panel in the inspection of the code.

For some simpler modules, there is a single sequence of trace tables that traces the state changes through that module. For each numbered *prime* (*primary program refinement*) of the module, we display first the intended function for that prime, followed by a table showing the values of all state items whose state has changed. The final set of state changes should match the overall intended function of the module.

In general, however, there are multiple trace table sequences for different cases of the logic. In that case, the trace table exposition for that module starts out with a list of all the cases. For each case, there is a trace table sequence showing the state changes in that case.

7.1. Trace table: `SudokuSolver.__init__()`

In the trace table for the constructor, the final state should show that all invariants are true. For the code, see Section 5.4, “`SudokuSolver.__init__()`: Constructor” (p. 8).

Here are the cases:

1. `self.givenPuzzle` is not a valid puzzle. See Section 7.1.1, “Case 1: Not a valid puzzle” (p. 26).
2. `self.givenPuzzle` is a valid puzzle. See Section 7.1.2, “Case 2: Valid puzzle” (p. 27).

7.1.1. Case 1: Not a valid puzzle

In prime 1 of this constructor, the actual code serves as the intended function.

```
#-- 1 --
self.givenPuzzle      = givenPuzzle
self.solutionObserver = solutionObserver
self.changeObserver   = changeObserver
self.nStateChanges    = 0
self.nSolutions       = 0
```

<code>self.givenPuzzle</code>	<code>givenPuzzle</code> as passed to constructor
<code>self.solutionObserver</code>	<code>solutionObserver</code> as passed to constructor
<code>self.changeObserver</code>	<code>changeObserver</code> as passed to constructor

⁹ <http://www.nmt.edu/~shipman/soft/clean/>

self.nStateChanges	0
self.nSolutions	0

```

#-- 2 --
# [ self.givenPuzzle is a string ->
#   if self.givenPuzzle is a valid sudoku puzzle as a string ->
#     self.__board := a list of BOARD_L integers
#                   representing that puzzle
#   else -> raise ValueError ]

```

By case assumption, **self.givenPuzzle** is not a valid puzzle, so the next state change is to raise **ValueError**.

Here is the overall intended function:

```

[ (givenPuzzle is a sudoku puzzle as a string) and
  (solutionObserver is a procedure or None) and
  (changeObserver is a procedure or None) ->
  if givenPuzzle is a valid sudoku puzzle ->
    ...
  else ->
    raise ValueError ]

```

7.1.2. Case 2: Valid puzzle

The case assumption here is that the **givenPuzzle** string is a valid puzzle.

```

#-- 1 --
self.givenPuzzle      = givenPuzzle
self.solutionObserver = solutionObserver
self.changeObserver   = changeObserver
self.nStateChanges    = 0
self.nSolutions       = 0

```

self.givenPuzzle	givenPuzzle as passed to constructor
self.solutionObserver	solutionObserver as passed to constructor
self.changeObserver	changeObserver as passed to constructor
self.nStateChanges	0
self.nSolutions	0

```

#-- 2 --
# [ self.givenPuzzle is a string ->
#   if self.givenPuzzle is a valid sudoku puzzle as a string ->
#     self.__board := a list of BOARD_L integers
#                   representing that puzzle
#   else -> raise ValueError ]

```

self.givenPuzzle	givenPuzzle as passed to constructor
self.solutionObserver	solutionObserver as passed to constructor
self.changeObserver	changeObserver as passed to constructor

self.nStateChanges	0
self.nSolutions	0
self.__board	a list of BOARD_L integers representing self.givenPuzzle

```
#-- 3 --
# [ self.__given := a copy of self.__board ]
```

self.givenPuzzle	givenPuzzle as passed to constructor
self.solutionObserver	solutionObserver as passed to constructor
self.changeObserver	changeObserver as passed to constructor
self.nStateChanges	0
self.nSolutions	0
self.__board	a list of BOARD_L integers representing self.givenPuzzle
self.__given	a list of BOARD_L integers representing self.givenPuzzle

Compare the states above with the class invariants:

```
Exports:
  .givenPuzzle:    [ as passed to constructor, read-only ]
  .solutionObserver: [ as passed to constructor, read-only ]
  .changeObserver: [ as passed to constructor, read-only ]
State/Invariants:
  .nStateChanges:  [ number of cell state changes so far ]
  .nSolutions:     [ number of solutions seen so far ]
  .__given:        [ the initial state of the puzzle as a list of integers ]
  .__board:        [ the current state of the puzzle as a list of integers ]
```

7.2. Trace table: `SudokuSolver.__readPuzzle()`

Cases:

1. The number of non-whitespace characters in **self.givenPuzzle** is not exactly **BOARD_L**. See Section 7.2.1, “Case 1: Wrong number of non-whitespace characters” (p. 28).
2. The character count is correct, but at least one of the non-whitespace characters is not a digit or “.”. See Section 7.2.2, “Case 2: Invalid characters” (p. 29).
3. **self.givenPuzzle** contains exactly **BOARD_L** non-whitespace characters, and each is a digit or “.”. See Section 7.2.3, “Case 3: Valid puzzle” (p. 30).

7.2.1. Case 1: Wrong number of non-whitespace characters

The case assumption is that the number of non-whitespace characters in **self.givenPuzzle** is not **BOARD_L**.

```
#-- 1 --
# [ charList := a list whose elements are the
```

```
#      non-whitespace characters of self.givenPuzzle in
#      the same order ]
```

charList	list of non-whitespace characters from self.givenPuzzle
-----------------	--

```
#-- 2 --
if len(charList) != BOARD_L:
    raise ValueError, ( "Puzzle has %d nonblank "
                        "characters; it should have exactly %d." %
                        (len(charList), BOARD_L) )
```

By case assumption, the length of **charList** is not **BOARD_L**, so we raise **ValueError**. The overall intended function:

```
[ self.givenPuzzle is a string ->
  if self.givenPuzzle is a sudoku puzzle ->
    ...
  else -> raise ValueError ]
```

7.2.2. Case 2: Invalid characters

The case assumptions are: **self.givenPuzzle** contains exactly **BOARD_L** non-whitespace characters, but at least one of those characters is not a digit or ".".

```
#-- 1 --
# [ charList := a list whose elements are the
#      non-whitespace characters of self.givenPuzzle in
#      the same order ]
```

charList	list of non-whitespace characters from self.givenPuzzle
-----------------	--

```
#-- 2 --
if len(charList) != BOARD_L:
    raise ValueError, ( "Puzzle has %d nonblank "
                        "characters; it should have exactly %d." %
                        (len(charList), BOARD_L) )
```

By case assumption, the number of non-whitespace characters is exactly **BOARD_L**, so we proceed to the next prime. The trace table is unchanged.

```
#-- 3 --
# [ if each element of charList is either "." or in
#   the interval ["1", "9"] ->
#     result := a list of integers corresponding to the
#               elements of charList consisting of
#               integer 0 where the value is "." and
#               int(c) for other values
#   else ->
#     raise ValueError ]
```

By case assumption, at least one non-whitespace character of **self.givenPuzzle** is neither a digit nor ".", so we raise **ValueError**. Here is the overall intended function:

```
[ self.givenPuzzle is a string ->
  if self.givenPuzzle is a valid sudoku puzzle ->
    ...
  else -> raise ValueError ]
```

7.2.3. Case 3: Valid puzzle

Case assumptions: the number of non-whitespace characters in **self.givenPuzzle** is exactly **BOARD_L**; and each of those characters is either a digit or ".".

```
#-- 1 --
# [ charList := a list whose elements are the
#   non-whitespace characters of self.givenPuzzle in
#   the same order ]
```

charList	list of non-whitespace characters from self.givenPuzzle
-----------------	--

```
#-- 2 --
if len(charList) != BOARD_L:
  raise ValueError, ( "Puzzle has %d nonblank "
    "characters; it should have exactly %d." %
    (len(charList), BOARD_L) )
```

By case assumption, the number of non-whitespace characters is exactly **BOARD_L**, so we proceed to the next prime. The trace table is unchanged.

```
#-- 3 --
# [ if each element of charList is either "." or in
#   the interval ["1", "9"] ->
#   result := a list of integers corresponding to the
#   elements of charList consisting of integer 0
#   where the value is "." and an integer in [1,9]
#   where the value is a digit
#   else ->
#   raise ValueError ]
```

charList	list of non-whitespace characters from self.givenPuzzle
result	a list of values, each 0 if the corresponding non-whitespace character of self.givenPuzzle is ".", or an integer in [1,9] if the corresponding non-whitespace character of self.givenPuzzle is a digit

```
#-- 4 --
return result
```

This returns a list of values, each 0 if the corresponding non-whitespace character of **self.givenPuzzle** is ".", or an integer in [1,9] if the corresponding non-whitespace character of **self.givenPuzzle** is a digit. Compare this to the overall intended function:

```
[ self.givenPuzzle is a string ->
  if self.givenPuzzle is a sudoku puzzle ->
    self.__board := a list of BOARD_L integers
```

```
else -> ... ]
```

representing that puzzle

7.3. Trace table: `SudokuSolver.get()`

Cases:

1. The row index x is not in the interval $[0, \text{MAT_L}]$. See Section 7.3.1, “Case 1: Row index out of range” (p. 31).
2. The row index x is in $[0, \text{MAT_L}]$ but the column index y is not in $[0, \text{MAT_L}]$. See Section 7.3.2, “Case 2: Valid row index, invalid column index” (p. 31).
3. The row and column indices are both valid. See Section 7.3.3, “Case 3: Valid row and column indices” (p. 32).

7.3.1. Case 1: Row index out of range

The case assumption is that the row index r is not in the interval $[0, \text{MAT_L}]$, that is, it is not a valid row index.

```
#-- 1 --
if not ( 0 <= r < MAT_L ):
    raise KeyError, ( "SudokuSolver.get(): Bad row "
                    "index, %d." % r )
```

This raises **KeyError**, satisfying the goal state:

```
[ r and c are integers ->
  if (0<=r<MAT_L) and (0<=r<MAT_L) ->
  ...
  else -> raise KeyError
```

7.3.2. Case 2: Valid row index, invalid column index

The case assumption here is that r is in the interval $[0, \text{MAT_L}]$ but c is not.

```
#-- 1 --
# [ if r is in [0, MAT_L] ->
#   I
#   else -> raise KeyError ]
```

By case assumption, this does nothing (the identity transform “**I**”).

```
#-- 2 --
# [ if c is in [0, MAT_L] ->
#   I
#   else -> raise KeyError ]
```

This raises **KeyError**, satisfying the goal state:

```
[ r and c are integers ->
  if (0<=r<MAT_L) and (0<=r<MAT_L) ->
```

```
...
else -> raise KeyError
```

7.3.3. Case 3: Valid row and column indices

The case assumption here is that both **r** and **c** fall in the interval **[0, MAT_L)**.

```
#-- 1 --
# [ if r is in [0, MAT_L) ->
#     I
# else -> raise KeyError ]
```

Does nothing.

```
#-- 2 --
# [ if c is in [0, MAT_L) ->
#     I
# else -> raise KeyError ]
```

Does nothing.

```
#-- 3 --
# [ x := index in self.__board corresponding to row r,
#     column c ]
```

x	Index in self.__board corresponding to row r , column c
----------	--

```
#-- 4 --
return self.__board[x]
```

This returns the element of **self.__board** corresponding to row **r** and column **c**, satisfying the goal state:

```
[ r and c are integers ->
  if (0<=r<MAT_L) and (0<=c<MAT_L) ->
    return the state of the cell at row r and column c
    as 0 for empty, or an integer in [1,9] if set
  else -> ... ]
```

Note the invariant on **self.__board** has matching verbiage:

```
.__board:
[ the current state of the puzzle as a list of integers,
  0 for empty, or in [1,9] if set ]
```

7.4. Inspection: **SudokuSolver.write()**

Verification of this method is aided by the vague wording of its intended function:

```
[ outFile is a writable file ->
  outFile += a representation of self's state in
  input-file format ]
```

The output format here is pretty simple, and assumes that it will be viewed in a monospaced font. Each row is written with a space between columns, with two extra spaces added between columns 2 and 3 and again between columns 5 and 6. We also print a blank line between rows 2 and 3, and again between rows 5 and 6. Here's a sample output:

```

7 2 .   . . 5   1 . .
. 1 5   9 8 .   4 . .
. 8 .   . . 1   . . 6

. 4 .   . 3 .   6 . 5
3 . .   6 . 2   . . 7
9 . 1   . 7 .   . 2 .

5 . .   7 . .   . 9 .
. . 2   . 9 6   3 5 .
. . 4   2 . .   . 6 8

```

Note the symmetry of the row and column presentation. We'll discuss the way it's implemented for the blank lines between rows; the same logic works for the extra spaces between columns.

We can test for the extra blank line either before writing each row of data, or after. It's pretty arbitrary, so we choose to test before writing the data. Below is a truth table for when to write the blank line. The author added a column for the index's remainder modulo **SUBMAT_L**. Why did the author do this? Well, when a behavior repeats cyclically, it suggests to the author that a modulo function is involved.

row	extra line?	row % SUBMAT_L
0	0	0
1	0	1
2	0	2
3	1	0
4	0	1
5	0	2
6	1	0
7	0	1
8	0	2

This table suggests that we could write the extra blank line whenever the **row % SUBMAT_L** is zero, but this logic would write an extra blank line initially. Therefore, the correct logic is: write an extra blank line before every row **r** for which ((**r**>0) and ((**r % SUBMAT_L**)==0)). Here is the actual code:

```

#-- 1 --
for rowx in range(MAT_L):
    #-- 1 body --
    # [ rowx is in [0,MAT_L) ->
    #     outFile += a representation of row rowx of self ]

    #-- 1.1 --
    # [ if ( ( rowx > 0 ) and
    #       ( rowx % SUBMAT_L ) == 0 ) ) ->
    #     outFile += an empty line
    #     else -> I ]

```

```
if ( ( rowx > 0 ) and
      ( ( rowx % SUBMAT_L ) == 0 ) ):
    print >> outFile

#-- 1.2 --
# [ outFile += a representation of row rowx of self ]
self.writeRow ( outFile, rowx )
```

7.5. Inspection: `SudokuSolver.writeRow()`

This method prints one row of the puzzle, translating zero values to “.” and nonzero values to the corresponding digit, printing an extra space between each column.

It also adds an extra two spaces between columns 2 and 3, and again between columns 5 and 6. The logic that dictates these extra columns is exactly the same as the logic for blank lines between rows in Section 7.4, “Inspection: `SudokuSolver.write()`” (p. 32).