Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
NAME = ""
COLLABORATORS = ""
```

CSE 30 Spring 2022 - Homework 15

Sudoku

Instructions

Please disregard the YOUR NAME and COLLABORATORS above. They are put there atomatically by the grading tool. You can find instructions on how to work on a homework on Canvas. Here is a short summary:

Submitting your work

To submit your work:

- First, click on "Runtime > Restart and run all", and check that you get no errors. This enables you to catch any error you might have introduced, and not noticed, due to your running cells out of order.
- Second, download the notebook in .ipynb format (File > Download .ipynb) and upload the .ipynb file to this form. This homework is due at **11:59pm on Tuesday, 31 May 2022**.

You can submit multiple times; the last submission before the deadline is the one that counts.

Homework format

For each question in this notebook, there is:

- A text description of the problem.
- One or more places where you have to insert your solution. You need to complete every place marked:

```
# YOUR CODE HERE
```

and you should not modify any other place.

- One or more test cells. Each cell is worth some number of points, marked at the top. You should not modify these tests cells. The tests pass if no error is printed out: when there is a statement that says, for instance:

  ```
  assert x == 2
  ```

  then the test passes if `x` has value 2, and fails otherwise. You can insert a `print(x)` (for this case!) somewhere if you want to debug your work; it is up to you.

Notes:

- Your code will be tested both according to the tests you can see (the `assert` statements you can see), *and* additional tests. This prevents you from hard-coding the answer to the particular questions posed. Your code should solve the *general* intended case, not hard-code the particular answer for the values used in the tests.

- **Please do not delete or add cells!** The test is autograded, and if you modify the test by adding or deleting cells, even if you re-add cells you delete, you may not receive credit.

- **Please do not import modules that are not part of the standard library.** You do not need any, and they will likely not available in the grading environment, leading your code to fail.

- **If you are inactive too long, your notebook might get disconnected from the back-end.** Your work is never lost, but you have to re-run all the cells before you continue.

- You can write out print statements in your code, to help you test/debug it. But remember: the code is graded on the basis of what it outputs or returns, not on the basis of what it prints.

- **TAs and tutors have access to this notebook,** so if you let them know you need their help, they can look at your work and give you advice.

Grading

Each cell where there are tests is worth a certain number of points. You get the points allocated to a cell only if you pass *all* the tests in the cell.

The tests in a cell include both the tests you can see, and other, similar, tests that are used for grading only. Therefore, you cannot hard-code the solutions: you really have to solve the essence of the problem, to receive the points in a cell.

Code of Conduct

- Work on the test yourself, alone.
- You can search documentation on the web, on sites such as the Python documentation sites, Stackoverflow, and similar, and you can use the results.
- You cannot share your work with others or solicit their help.

Let us write a Sudoku solver. We want to get as input a Sudoku with some cells filled with values, and we want to get as output a solution, if one exists, and otherwise a notice that the input Sudoku puzzle has no solutions.

You will wonder, why spend so much time on Sudoku?

For two reasons.

First, the way we go about solving Sudoku is prototypical of a very large number of problems in computer science. In these problems, the solution is attained through a mix of search (we attempt to fill a square with a number and see if it works out), and constraint propagation (if we fill a square with, say, a 1, then there can be no 1's in the same row, column, and 3x3 square).

Second, and related, the way we go about solving Sudoku puzzles is closely related to how SAT solvers work. So closely related, in fact, that while *we* describe for you how a Sudoku solver works, *you* will have to write a SAT solver as exercise.

Sudoku representation

First, let us do some grunt work and define a representation for a Sudoku problem.

One initial idea would be to represent a Sudoku problem via a 9×9 matrix, where each entry can be either a digit from 1 to 9, or 0 to signify "blank". This would work in some sense, but it would not be a very useful representation. If you have solved Sudoku by hand (and if you have not, please go and solve a couple; it will teach you a lot about what we need to do), you will know that the following strategy works:

Repeat:

- Look at all blank spaces. Can you find one where only one digit fits? If so, write the digit there.
- If you cannot find any blank space as above, try to find one where only a couple or so digits can fit. Try putting in one of those digits, and see if you can solve the puzzle with that choice. If not, backtrack, and try another digit.

Thus, it will be very useful to us to remember not only the known digits, but also, which digits can fit into any blank space. Hence, we represent a Sudoku problem via a 9×9 matrix of *sets*: each set contains the digits that can fit in a given space. Of course, a known digit is just a set containing only one element. We will solve a Sudoku problem by progressively "shrinking" these sets of possibilities, until they all contain exactly one element.

Let us write some code that enables us to define a Sudoku problem, and display it for us; this will be very useful both for our fun and for debugging.

First, though, let's write a tiny helper function that returns the only element from a singleton set.

In [2]:
```
def getel(s):
    """Returns the unique element in a singleton set (or list)."""
    assert len(s) == 1
    return list(s)[0]
```

In [3]:
```
import json

class Sudoku(object):

    def __init__(self, elements):
        """Elements can be one of:
        Case 1: a list of 9 strings of length 9 each.
        Each string represents a row of the initial Sudoku puzzle,
        with either a digit 1..9 in it, or with a blank or _ to signify
        a blank cell.
        Case 2: an instance of Sudoku.  In that case, we initialize an
        object to be equal (a copy) of the one in elements.
        Case 3: a list of list of sets, used to initialize the problem."""
        if isinstance(elements, Sudoku):
            # We let self.m consist of copies of each set in elements.m
            self.m = [[x.copy() for x in row] for row in elements.m]
        else:
            assert len(elements) == 9
            for s in elements:
                assert len(s) == 9
            # We let self.m be our Sudoku problem, a 9x9 matrix of sets.
            self.m = []
            for s in elements:
                row = []
                for c in s:
                    if isinstance(c, str):
                        if c.isdigit():
                            row.append({int(c)})
                        else:
                            row.append({1, 2, 3, 4, 5, 6, 7, 8, 9})
                    else:
                        assert isinstance(c, set)
                        row.append(c)
                self.m.append(row)


    def show(self, details=False):
        """Prints out the Sudoku matrix.  If details=False, we print out
        the digits only for cells that have singleton sets (where only
        one digit can fit).  If details=True, for each cell, we display the
```

```
        sets associated with the cell."""
        if details:
            print("+-----------------------------+-----------------------------+---
            for i in range(9):
                r = '|'
                for j in range(9):
                    # We represent the set {2, 3, 5} via _23_5____
                    s = ''
                    for k in range(1, 10):
                        s += str(k) if k in self.m[i][j] else '_'
                    r += s
                    r += '|' if (j + 1) % 3 == 0 else ' '
                print(r)
                if (i + 1) % 3 == 0:
                    print("+-----------------------------+------------------------
        else:
            print("+---+---+---+")
            for i in range(9):
                r = '|'
                for j in range(9):
                    if len(self.m[i][j]) == 1:
                        r += str(getel(self.m[i][j]))
                    else:
                        r += "."
                    if (j + 1) % 3 == 0:
                        r += "|"
                print(r)
                if (i + 1) % 3 == 0:
                    print("+---+---+---+")


    def to_string(self):
        """This method is useful for producing a representation that
        can be used in testing."""
        as_lists = [[list(self.m[i][j]) for j in range(9)] for i in range(9)]
        return json.dumps(as_lists)


    @staticmethod
    def from_string(s):
        """Inverse of above."""
        as_lists = json.loads(s)
        as_sets = [[set(el) for el in row] for row in as_lists]
        return Sudoku(as_sets)


    def __eq__(self, other):
        """Useful for testing."""
        return self.m == other.m
```

Let us input a problem (the Sudoku example found on this Wikipedia page) and check that our serialization and deserialization works.

In [4]:
```
# Let us ensure that nose is installed.
try:
```

```
    from nose.tools import assert_equal, assert_true
    from nose.tools import assert_false, assert_almost_equal
except:
    !pip install nose
    from nose.tools import assert_equal, assert_true
    from nose.tools import assert_false, assert_almost_equal
```

In [5]:
```
from nose.tools import assert_equal

sd = Sudoku([
    '53__7____',
    '6__195___',
    '_98____6_',
    '8___6___3',
    '4__8_3__1',
    '7___2___6',
    '_6____28_',
    '___419__5',
    '____8__79'
])
sd.show()
sd.show(details=True)
s = sd.to_string()
sdd = Sudoku.from_string(s)
sdd.show(details=True)
assert_equal(sd, sdd)
```

```
+---+---+---+
|53.|.7.|...|
|6..|195|...|
|.98|...|.6.|
+---+---+---+
|8..|.6.|..3|
|4..|8.3|..1|
|7..|.2.|..6|
+---+---+---+
|.6.|...|28.|
|...|419|..5|
|...|.8.|.79|
+---+---+---+
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ __3______ 123456789|123456789 ______7__ 123456789|123456789 123456789 123
456789|
|_____6___ 123456789 123456789|1________ ________9 ____5____|123456789 123456789 123
456789|
|123456789 ________9 _______8_|123456789 123456789 123456789|123456789 _____6___ 123
456789|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ 123456789 123456789|123456789 _____6___ 123456789|123456789 123456789 __3
______|
|___4_____ 123456789 123456789|_______8_ 123456789 __3______|123456789 123456789 1__
______|
|______7__ 123456789 123456789|123456789 _2_______ 123456789|123456789 123456789 ___
__6___|
+-----------------------------+-----------------------------+-----------------------
------+
|123456789 _____6___ 123456789|123456789 123456789 123456789|_2_______ _______8_ 123
456789|
|123456789 123456789 123456789|___4_____ 1________ ________9|123456789 123456789 ___
_5____|
|123456789 123456789 123456789|123456789 _______8_ 123456789|123456789 ______7__ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ __3______ 123456789|123456789 ______7__ 123456789|123456789 123456789 123
456789|
|_____6___ 123456789 123456789|1________ ________9 ____5____|123456789 123456789 123
456789|
|123456789 ________9 _______8_|123456789 123456789 123456789|123456789 _____6___ 123
456789|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ 123456789 123456789|123456789 _____6___ 123456789|123456789 123456789 __3
______|
|___4_____ 123456789 123456789|_______8_ 123456789 __3______|123456789 123456789 1__
______|
|______7__ 123456789 123456789|123456789 _2_______ 123456789|123456789 123456789 ___
__6___|
+-----------------------------+-----------------------------+-----------------------
```

```
------+
|123456789 _____6___ 123456789|123456789 123456789 123456789|_2_______ _______8_ 123
456789|
|123456789 123456789 123456789|___4_____ 1________ ________9|123456789 123456789 ___
_5____|
|123456789 123456789 123456789|123456789 _______8_ 123456789|123456789 ______7__ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
```

Let's test our constructor statement when passed a Sudoku instance.

In [6]:
```
sd1 = Sudoku(sd)
assert_equal(sd, sd1)
```

Constraint propagation

When the set in a Sudoku cell contains only one element, this means that the digit at that cell is known. We can then propagate the knowledge, ruling out that digit in the same row, in the same column, and in the same 3x3 cell.

We first write a method that propagates the constraint from a single cell. The method will return the list of newly-determined cells, that is, the list of cells who also now (but not before) are associated with a 1-element set. This is useful, because we can then propagate the constraints from those cells in turn. Further, if an empty set is ever generated, we raise the exception Unsolvable: this means that there is no solution to the proposed Sudoku puzzle.

We don't want to steal all the fun from you; thus, we will give you the main pieces of the implemenetation, but we ask you to fill in the blanks. We provide tests so you can catch any errors right away.

Question 1: Propagating a single cell

In [7]:
```
class Unsolvable(Exception):
    pass


def sudoku_ruleout(self, i, j, x):
    """The input consists in a cell (i, j), and a value x.
    The function removes x from the set self.m[i][j] at the cell, if present, and:
    - if the result is empty, raises Unsolvable;
    - if the cell used to be a non-singleton cell and is now a singleton
      cell, then returns the set {(i, j)};
    - otherwise, returns the empty set."""
    c = self.m[i][j]
    n = len(c)
    c.discard(x)
    self.m[i][j] = c
```

```
    if len(c) == 0:
        raise Unsolvable()
    return {(i, j)} if 1 == len(c) < n else set()

Sudoku.ruleout = sudoku_ruleout
```

The method `propagate_cell(ij)` takes as input a pair `ij` of coordinates. If the set of possible digits `self.m[i][j]` for cell i,j contains more than one digit, then no propagation is done. If the set contains a single digit `x`, then we:

- Remove `x` from the sets of all other cells on the same row, column, and 3x3 block.
- Collect all the newly singleton cells that are formed, due to the digit `x` being removed, and we return them as a set.

We give you an implementation that takes care of removing `x` from the same row, and we ask you to complete the implementation to take care of the column and 3x3 block as well.

In [8]:
```
### Exercise: define cell propagation

def sudoku_propagate_cell(self, ij):
    """Propagates the singleton value at cell (i, j), returning the list
    of newly-singleton cells."""
    i, j = ij
    if len(self.m[i][j]) > 1:
        # Nothing to propagate from cell (i,j).
        return set()
    # We keep track of the newly-singleton cells.
    newly_singleton = set()
    x = getel(self.m[i][j]) # Value at (i, j).
    # Same row.
    for jj in range(9):
        if jj != j: # Do not propagate to the element itself.
            newly_singleton.update(self.ruleout(i, jj, x))
    # Same column.
    # YOUR CODE HERE
    for ii in range(9):
        if ii != i:
            newly_singleton.update(self.ruleout(ii, j, x))
    # Same block of 3x3 cells.
    # YOUR CODE HERE
    r = 3 * (i//3)
    c = 3 * (j//3)
    for l in range(r, r+3):
        for h in range(c, c+3):
            if not l == i and not h == j:
                newly_singleton.update(self.ruleout(l, h, x))
    # Returns the list of newly-singleton cells.
    return newly_singleton

Sudoku.propagate_cell = sudoku_propagate_cell
```

In [9]:
```
# Here you can write your own tests if you like.
```

```
# YOUR CODE HERE
```

In [10]:
```
### 10 points: Tests for cell propagation

tsd = Sudoku.from_string('[[[5], [3], [2], [6], [7], [8], [9], [1, 2, 4], [2]], [[6
tsd.show(details=True)
try:
    tsd.propagate_cell((0, 2))
except Unsolvable:
    print("Good! It was unsolvable.")
else:
    raise Exception("Hey, it was unsolvable")

tsd = Sudoku.from_string('[[[5], [3], [2], [6], [7], [8], [9], [1, 2, 4], [2, 3]],
tsd.show(details=True)
assert_equal(tsd.propagate_cell((0, 2)), {(0, 8), (2, 0)})
```

```
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ __3______ _2_______|_____6___ ______7__ _______8_|________9 12_4_____ _2_
______|
|_____6___ ______7__ 12_4__7__|123______ ________9 ____5____|__3______ 12_4_____ ___
____8_|
|12_______ ________9 _______8_|__3______ ___4_____ 12_______|____5____ _____6___ ___
___7__|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ ____5____ ________9|1_____7_9 _____6___ 1__4__7_9|___4_____ _2_______ __3
______|
|___4_____ _2_______ _____6___|_______8_ ____5____ __3______|______7__ ________9 1__
______|
|______7__ 1________ __3______|________9 _2_______ ___4_____|_______8_ ____5____ ___
__6___|
+-----------------------------+-----------------------------+-----------------------
------+
|1_______9 _____6___ 1___5_7_9|____5_7_9 __3______ ______7_9|_2_______ _______8_ ___
4_____|
|_2______9 _______8_ _2____7_9|___4_____ 1________ _2____7_9|_____6___ __3______ ___
_5____|
|__3______ ___4_____ _2345____|_2__56___ _______8_ _____6___|1________ ______7__ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
Good! It was unsolvable.
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ __3______ _2_______|_____6___ ______7__ _______8_|________9 12_4_____ _23
______|
|_____6___ ______7__ 12_4__7__|123______ ________9 ____5____|__3______ 12_4_____ ___
____8_|
|12_______ ________9 _______8_|__3______ ___4_____ 12_______|____5____ _____6___ ___
___7__|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ ____5____ ________9|1_____7_9 _____6___ 1__4__7_9|___4_____ _2_______ __3
______|
|___4_____ _2_______ _____6___|_______8_ ____5____ __3______|______7__ ________9 1__
______|
|______7__ 1________ __3______|________9 _2_______ ___4_____|_______8_ ____5____ ___
__6___|
+-----------------------------+-----------------------------+-----------------------
------+
|1_______9 _____6___ 1___5_7_9|____5_7_9 __3______ ______7_9|_2_______ _______8_ ___
4_____|
|_2______9 _______8_ _2____7_9|___4_____ 1________ _2____7_9|_____6___ __3______ ___
_5____|
|__3______ ___4_____ _2345____|_2__56___ _______8_ _____6___|1________ ______7__ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
```

Propagating all cells, once

The simplest thing we can do is propagate each cell, once.

In [11]:
```
def sudoku_propagate_all_cells_once(self):
    """This function propagates the constraints from all singletons."""
    for i in range(9):
        for j in range(9):
            self.propagate_cell((i, j))

Sudoku.propagate_all_cells_once = sudoku_propagate_all_cells_once
```

In [12]:
```
sd = Sudoku([
    '53__7____',
    '6__195___',
    '_98____6_',
    '8___6___3',
    '4__8_3__1',
    '7___2___6',
    '_6____28_',
    '___419__5',
    '____8__79'
])
sd.show()
sd.propagate_all_cells_once()
sd.show()
sd.show(details=True)
```

```
+---+---+---+
|53.|.7.|...|
|6..|195|...|
|.98|...|.6.|
+---+---+---+
|8..|.6.|..3|
|4..|8.3|..1|
|7..|.2.|..6|
+---+---+---+
|.6.|...|28.|
|...|419|..5|
|...|.8.|.79|
+---+---+---+
+---+---+---+
|53.|.7.|...|
|6..|195|...|
|.98|...|.6.|
+---+---+---+
|8..|.6.|..3|
|4..|853|..1|
|7..|.2.|..6|
+---+---+---+
|.6.|..7|284|
|...|419|.35|
|...|.8.|.79|
+---+---+---+
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ __3______ 12_4_____|_2___6___ ______7__ _2_4_6_8_|1__4___89 12_4____9 _2_
4___8_|
|_____6___ _2_4__7__ _2_4__7__|1________ ________9 ____5____|__34__78_ _234_____ _2_
4__78_|
|12_______ ________9 _______8_|_23______ __34_____ _2_4_____|1_345_7__ _____6___ _2_
4__7__|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ 12__5____ 12__5___9|____5_7_9 _____6___ 1__4__7__|___45_7_9 _2_45___9 __3
______|
|___4_____ _2__5____ _2__56__9|_______8_ ____5____ __3______|____5_7_9 _2__5___9 1__
______|
|______7__ 1___5____ 1_3_5___9|____5___9 _2_______ 1__4_____|___45__89 ___45___9 ___
__6___|
+-----------------------------+-----------------------------+-----------------------
------+
|1_3_____9 _____6___ 1_345_7_9|__3_5_7__ __3_5____ ______7__|_2_______ _______8_ ___
4_____|
|_23______ _2____78_ _23___7__|___4_____ 1________ ________9|__3__6___ __3______ ___
_5____|
|123______ 12_45____ 12345____|_23_56___ _______8_ _2___6___|1_34_6___ ______7__ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
```

Question 2: Propagating all cells, repeatedly

This is a good beginning, but it's not quite enough. As we propagate the constraints, cells that did not use to be singletons may have become singletons. For eample, in the above example, the center cell has become known to be a 5: we need to make sure that also these singletons are propagated.

This is why we have written propagate_cell so that it returns the set of newly-singleton cells. We need now to write a method `full_propagation` that at the beginning starts with a set of `to_propagate` cells (if it is not specified, then we just take it to consist of all singleton cells). Then, it picks a cell from the to_propagate set, and propagates from it, adding any newly singleton cell to to_propagate. Once there are no more cells to be propagated, the method returns. If this sounds similar to graph reachability, it is ... because it is! It is once again the algorithmic pattern of keeping a list of work to be done, then iteratively picking an element from the list, doing it, possibly updating the list of work to be done with new work that has to be done as a result of what we just did, and continuing in this fashion until there is nothing left to do. We will let you write this function. The portion you have to write can be done in three lines of code.

In [13]:
```
### Exercise: define full propagation

def sudoku_full_propagation(self, to_propagate=None):
    """Iteratively propagates from all singleton cells, and from all
    newly discovered singleton cells, until no more propagation is possible.
    @param to_propagate: sets of cells from where to propagate.  If None, propagate
        from all singleton cells.
    @return: nothing.
    """
    if to_propagate is None:
        to_propagate = {(i, j) for i in range(9) for j in range(9)}
    # This code is the (A) code; will be referenced later.
    # YOUR CODE HERE
    while len(to_propagate) > 0:
        temp = to_propagate.pop()
        newSet = self.propagate_cell(temp)
        if newSet != set():
            for i in range(len(newSet)):
                newList = list(newSet)[i]
                to_propagate.add(newList)


Sudoku.full_propagation = sudoku_full_propagation
```

In [14]:
```
# Here you can write your own tests if you like.

# YOUR CODE HERE
sd = Sudoku([
    '53__7____',
    '6__195___',
    '_98____6_',
    '8___6___3',
    '4__8_3__1',
    '7___2___6',
```

```
    '_6____28_',
    '___419__5',
    '____8__79'
])
sd.full_propagation()
```

In [15]:

```
### 10 points: Tests for full propagation

sd = Sudoku([
    '53__7____',
    '6__195___',
    '_98____6_',
    '8___6___3',
    '4__8_3__1',
    '7___2___6',
    '_6____28_',
    '___419__5',
    '____8__79'
])
sd.full_propagation()
sd.show(details=True)
sdd = Sudoku.from_string('[[[5], [3], [4], [6], [7], [8], [9], [1], [2]], [[6], [7]
assert_equal(sd, sdd)
```

```
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ __3______ ___4_____|_____6___ ______7__ _______8_|________9 1________ _2_
______|
|_____6___ ______7__ _2_______|1________ ________9 ____5____|__3______ ___4_____ ___
____8_|
|1________ ________9 _______8_|__3______ ___4_____ _2_______|____5____ _____6___ ___
___7__|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ ____5____ ________9|______7__ _____6___ 1________|___4_____ _2_______ __3
______|
|___4_____ _2_______ _____6___|_______8_ ____5____ __3______|______7__ ________9 1__
______|
|______7__ 1________ __3______|________9 _2_______ ___4_____|_______8_ ____5____ ___
__6___|
+-----------------------------+-----------------------------+-----------------------
------+
|________9 _____6___ 1________|____5____ __3______ ______7__|_2_______ _______8_ ___
4_____|
|_2_______ _______8_ ______7__|___4_____ 1________ ________9|_____6___ __3______ ___
_5____|
|__3______ ___4_____ ____5____|_2_______ _______8_ _____6___|1________ ______7__ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
```

We solved our example problem! Constraint propagation, iterated, led us to the solution!

Searching for a solution

Many Sudoku problems can be solved entirely by constraint propagation.
They are designed to be so: they are designed to be relatively easy, so that humans can solve them while on a lounge chair at the beach -- I know this from personal experience!

But it is by no means necessary that this is true. If we create more complex problems, or less determined problems, constraint propagation no longer suffices. As a simple example, let's just blank some cells in the previous problem, and run full propagation again:

In [16]:
```
sd = Sudoku([
    '53__7____',
    '6___95___',
    '_98____6_',
    '8___6___3',
    '4__8_3__1',
    '7___2___6',
    '_6____28_',
    '___41___5',
    '____8__79'
])
sd.show()
sd.full_propagation()
sd.show(details=True)
```

```
+---+---+---+
|53.|.7.|...|
|6..|.95|...|
|.98|...|.6.|
+---+---+---+
|8..|.6.|..3|
|4..|8.3|..1|
|7..|.2.|..6|
+---+---+---+
|.6.|...|28.|
|...|41.|..5|
|...|.8.|.79|
+---+---+---+
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ __3______ 12_4_____|12___6___ ______7__ 12___6_8_|___4___89 12_4_____ _2_
____8_|
|_____6___ 1__4__7__ 12_4__7__|123______ ________9 ____5____|__34___8_ 12_4_____ _2_
___78_|
|12_______ ________9 _______8_|123______ ___4_____ 12_______|__3_5____ _____6___ _2_
___7__|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ 1___5____ 1___5___9|1_____7_9 _____6___ 1__4__7_9|___45____ _2_45____ __3
______|
|___4_____ _2_______ _____6___|_______8_ ____5____ __3______|______7__ ________9 1__
______|
|______7__ 1___5____ 1_3_5___9|1_______9 _2_______ 1__4____9|___45__8_ ___45____ ___
__6___|
+-----------------------------+-----------------------------+-----------------------
------+
|1_______9 _____6___ 1___5_7_9|____5_7_9 __3______ ______7_9|_2_______ _______8_ ___
4_____|
|_2______9 ______78_ _2____7_9|___4_____ 1________ _2____7_9|_____6___ __3______ ___
_5____|
|_23______ ___45____ _2345____|_2__56___ _______8_ _2___6___|1________ ______7__ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
```

As we see, there are still undetermined values. We can peek into the detailed state of the solution:

In [17]:
```python
sd.show(details=True)
# Let's save this Sudoku for later.
sd_partially_solved = Sudoku(sd)
```

```
+-----------------------------+-----------------------------+-----------------------------+
|____5____ __3______ 12_4_____|12___6___ ______7__ 12___6_8_|___4___89 12_4_____ _2_____8_|
|_____6___ 1__4__7__ 12_4__7__|123______ ________9 ____5____|__34___8_ 12_4_____ _2____78_|
|12_______ ________9 _______8_|123______ ___4_____ 12_______|__3_5____ _____6___ _2____7__|
+-----------------------------+-----------------------------+-----------------------------+
|_______8_ 1___5____ 1___5___9|1_____7_9 _____6___ 1__4__7_9|___45____ _2_45____ __3______|
|___4_____ _2_______ _____6___|_______8_ ____5____ __3______|______7__ ________9 1________|
|______7__ 1___5____ 1_3_5___9|1_______9 _2_______ 1__4____9|___45__8_ ___45____ _____6___|
+-----------------------------+-----------------------------+-----------------------------+
|1_______9 _____6___ 1___5_7_9|____5_7_9 __3______ ______7_9|_2_______ _______8_ ___4_____|
|_2______9 ______78_ _2____7_9|___4_____ 1________ _2____7_9|_____6___ __3______ ____5____|
|_23______ ___45____ _2345____|_2__56___ _______8_ _2___6___|1________ ______7__ ________9|
+-----------------------------+-----------------------------+-----------------------------+
```

What can we do when constraint propagation fails? The only thing we can do is make a guess. We can take one of the cells whose set contains multiple digits, such as cell (2, 0) (starting counting at 0, as in Python), which contains $\{1, 2\}$, and try one of the values, for instance 1.
We can see whether assigning to the cell the singleton set $\{1\}$ leads to the solution. If not, we try the value $\{2\}$ instead. If the Sudoku problem has a solution, one of these two values must work.

Classically, this way of searching for a solution has been called search with *backtracking*. The backtracking is because we can choose a value, say 1, and then do a lot of work, propagating the new constraint, making further guesses, and so on and so forth. If that does not pan out, we must "backtrack" and return to our guess, choosing (in our example) 2 instead.

Let us implement search with backtracking. What we need to do is something like this:

search():

1. propagate constraints.
2. if solved, hoorrayy!
3. if impossible, raise Unsolvable()
4. if not fully solved, pick a cell with multiple digits possible, and iteratively:

- Assign one of the possible values to the cell.
- Call search() with that value for the cell.

- If Unsolvable is raised by the search() call, move on to the next value.
- If all values returned Unsolvable (if we tried them all), then we raise Unsolvable.

So we see that search() is a recursive function.
From the pseudo-code above, we see that it might be better to pick a cell with few values possible at step 4 above, so as to make our chances of success as good as possible. For instance, it is much better to choose a cell with set $\{1, 2\}$ than one with set $\{1, 3, 5, 6, 7, 9\}$, as the probability of success is $1/2$ in the first case and $1/6$ in the second case. Of course, it may be possible to come up with much better heuristics to guide our search, but this will have to do so far.

One fine point with the search above is the following. So far, an object has a self.m matrix, which contains the status of the Sudoku solution. We cannot simply pass self.m recursively to search(), because in the course of the search and constraint propagation, self.m will be modified, and there is no easy way to keep track of these modifications. Rather, we will write search() as a method, and when we call it, we will:

- First, create a copy of the current object via the Sudoku constructor, so we have a copy we can modify.
- Second, we assign one of the values to the cell, as above;
- Third, we will call the search() method of that object.

Furthermore, when a solution is found, as in the hoorraay! above, we need to somehow return the solution. There are two ways of doing this: via standard returns, or by raising an exception.

In [18]:
```
def sudoku_done(self):
    """Checks whether an instance of Sudoku is solved."""
    for i in range(9):
        for j in range(9):
            if len(self.m[i][j]) > 1:
                return False
    return True

Sudoku.done = sudoku_done


def sudoku_search(self, new_cell=None):
    """Tries to solve a Sudoku instance."""
    to_propagate = None if new_cell is None else {new_cell}
    self.full_propagation(to_propagate=to_propagate)
    if self.done():
        return self # We are a solution
    # We need to search.  Picks a cell with as few candidates as possible.
    candidates = [(len(self.m[i][j]), i, j)
                  for i in range(9) for j in range(9) if len(self.m[i][j]) > 1]
    _, i, j = min(candidates)
    values = self.m[i][j]
    # values contains the list of values we need to try for cell i, j.
    # print("Searching values", values, "for cell", i, j)
```

```
    for x in values:
        # print("Trying value", x)
        sd = Sudoku(self)
        sd.m[i][j] = {x}
        try:
            # If we find a solution, we return it.
            return sd.search(new_cell=(i, j))
        except Unsolvable:
            # Go to next value.
            pass
    # All values have been tried, apparently with no success.
    raise Unsolvable()

Sudoku.search = sudoku_search


def sudoku_solve(self, do_print=True):
    """Wrapper function, calls self and shows the solution if any."""
    try:
        r = self.search()
        if do_print:
            print("We found a solution:")
            r.show()
            return r
    except Unsolvable:
        if do_print:
            print("The problem has no solutions")

Sudoku.solve = sudoku_solve
```

Let us try this on our previous Sudoku problem that was not solvable via constraint propagation alone.

In [19]:
```
sd = Sudoku([
    '53__7____',
    '6__195___',
    '_98____6_',
    '8___6___3',
    '4__8_3__1',
    '7___2___6',
    '_6____28_',
    '___419__5',
    '____8__79'
])
_ = sd.solve()
```

```
We found a solution:
+---+---+---+
|531|678|942|
|674|295|318|
|298|341|567|
+---+---+---+
|859|167|423|
|426|853|791|
|713|924|856|
+---+---+---+
|165|739|284|
|987|412|635|
|342|586|179|
+---+---+---+
```

It works, search with constraint propagation solved the Sudoku puzzle!

The choice - constraint propagation - recursion paradigm.

We have learned a general strategy for solving difficult problems. The strategy can be summarized thus: **choice - constraint propagation - recursion.**

In the *choice* step, we make one guess from a set of possible guesses. If we want our search for a solution to be exhaustive, as in the above Sudoku example, we ensure that we try iteratively all choices from a set of choices chosen so that at least one of them must succeed. In the above example, we know that at least one of the digit values must be the true one, hence our search is exhaustive. In other cases, we can trade off exhaustiveness for efficiency, and we may try only a few choices, guided perhaps by an heuristic.

The *constraint propagation* step propagates the consequences of the choice to the problem. Each choice thus gives rise to a new problem, which is a little bit simpler than the original one as some of the possible choices, that is, some of its complexity, has been removed. In the Sudoku case, the new problem has less indetermination, as at least one more of its cells has a known digit in it.

The problems resulting from *constraint propagation*, while simpler, may not be solved yet. Hence, we *recur*, calling the solution procedure on them as well. As these problems are simpler (they contain fewer choices), eventually the recursion must reach a point where no more choice is possible, and whether constraint propagation should yield a completely defined problem, one of which it is possible to say whether it is solvable or not with a trivial test. This forms the base case for the recursion.

This solution strategy applies very generally, to problems well beyond Sudoku.

Part 2: Digits must go somewhere

If you have played Sudoku before, you might have found the way we solved Sudoku puzzles a bit odd. The constraint we encoded is:

> If a digit appears in a cell, it cannot appear anywhere else on the same row, column, or 3x3 block as the cell.

This *is* a rule of Sudoku. Normally, however, we hear Sudoku described in a different way:

> Every column, row, and 3x3 block should contain all the 1...9 digits exactly once.

There are two questions. The first is: are the two definitions equivalent? Well, no; the first definition does not say what the digits are (e.g., does not rule out 0). But in our Sudoku representation, we *start* by saying that every cell can contain only one of 1...9. If every row (or column, or 3x3 block) cannot contain more than one repetition of each digit, and if there are 9 digits and 9 cells in the row (or column, or block), then clearly every digit must appear exactly once in the row (or column, or block). So once the set of digits is specified, the two definitions are equivalent.

The second question is: but still, what happens to the method we usually employ to solve Sudoku? I generally don't solve Sudoku puzzles by focusing on one cell at a time, and thinking: is it the case that this call can contain only one digit? This is the strategy employed by the solver above. But it is not the strategy I normally use. I generally solve Sudoku puzzles by looking at a block (or row, or column), and thinking: let's consider the digit k ($1 \leq k \leq 9$). Where can it go in the block? And if I find that the digit can go in one block cell only, I write it there.
Does the solver work even without this "where can it go" strategy? And can we make it follow it?

The solver works even without the "where can it go" strategy because it exaustively tries all possibilities. This means the solver works without the strategy; it does not say that the solver works *well* without the strategy.

We can certainly implement the *where can it go* strategy, as part of constraint propagation; it would make our solver more efficient.

Question 3: A better `full_propagation` method

Not a real question; just copy some previous code into a new method.

There is a subtle point in applying the *where can it go* heuristics.

Before, when our only constraint was the uniqueness in each row, column, and block, we needed to propagate only from cells that hold a singleton value. If a cell held a non-singleton set of digits, such as $\{2, 5\}$, no values could be ruled out as a consequence of this on the same row, column, or block.

The *where can it go* heuristic, instead, benefits from knowing that in a cell, the set of values went for instance from $\{2, 3, 5\}$ to $\{2, 5\}$: by ruling out the possibility of a 3 in this cell, it may be possibe to deduct that the digit 3 can appear in only one (other) place in the block, and place it there.

Thus, we modify the `full_propagation` method. The method does:

- Repeat:
  - first does propagation as before, based on singletons;
  - then, it applies the *where can it go* heuristic on the whole Sudoku board.
- until there is nothing more that can be propagated.

Thus, we replace the `full_propagation` method previously defined with this new one, where the (A) block of code is what you previously wrote in `full_propagation`. You don't need to write new code here: just copy your solution for `full_propagation` into the (A) block below.

In [20]:
```
### Exercise: define full propagation with where can it go

def sudoku_full_propagation_with_where_can_it_go(self, to_propagate=None):
    """Iteratively propagates from all singleton cells, and from all
    newly discovered singleton cells, until no more propagation is possible."""
    if to_propagate is None:
        to_propagate = {(i, j) for i in range(9) for j in range(9)}
    while len(to_propagate) > 0:
        # Here is your previous solution code from (A) in full_propagation.
        # Please copy it below. No change is required.
        # YOUR CODE HERE
        temp = to_propagate.pop()
        newSet = self.propagate_cell(temp)
        if newSet != set():
            for i in range(len(newSet)):
                newList = list(newSet)[i]
                to_propagate.add(newList)
        # Now we check whether there is any other propagation that we can
        # get from the where can it go rule.
        to_propagate = self.where_can_it_go()
```

Question 4: Implement the `occurs_once_in_sets` helper

To implement the `where_can_it_go` method, let us write a helper function, or better, let's have you write it. Given a sequence of sets $S_1, S_2, \ldots, S_n$, we want to obtain the set of elements that appear in *exactly one* of the sets (that is, they appear in one set, and *only* in one set). Mathematically, we can write this as

$$(S_1 \setminus (S_2 \cup \cdots \cup S_n)) \cup (S_2 \setminus (S_1 \cup S_3 \cup \cdots \cup S_n)) \cup \cdots \cup (S_n \setminus (S_1 \cup \cdots \cup S_{n-1}))$$

even though that's certainly not the easiest way to compute it! The problem can be solved with the help of defaultdict to count the occurrences, and is 5 lines long. Of course, other solutions are possible as well.

In [21]:
```
### Exercise: define helper function to check once-only occurrence

from collections import defaultdict

def occurs_once_in_sets(set_sequence):
    """Returns the elements that occur only once in the sequence of sets set_sequen
    The elements are returned as a set."""
    # YOUR CODE HERE
    sort = {}
    elements = set()
    for i in range(len(set_sequence)):
        for j in range(len(list(set_sequence)[i])):
            try:
                sort[list(list(set_sequence)[i])[j]] += 1
            except:
                sort.setdefault(list(list(set_sequence)[i])[j], 1)
    for k in sort:
        if sort[k] == 1:
            elements.add(k)
    return elements
```

In [22]:
```
# Here you can write your own tests if you like.

# YOUR CODE HERE
```

Let us test it.

In [23]:
```
### 10 points: Tests for once-only

from nose.tools import assert_equal

assert_equal(occurs_once_in_sets([{1, 2}, {2, 3}]), {1, 3})
assert_equal(occurs_once_in_sets([]), set())
assert_equal(occurs_once_in_sets([{2, 3, 4}]), {2, 3, 4})
assert_equal(occurs_once_in_sets([set()]), set())
assert_equal(occurs_once_in_sets([{2, 3, 4, 5, 6}, {5, 6, 7, 8}, {5, 6, 7}, {4, 6,
```

Question 5: Implement *where can it go*.

We are now ready to write -- or better, to have you write -- the *where can it go* method.
The method is global: it examines all rows, all columns, and all blocks.
If it finds that in a row (or column, or block), a value can fit in only one cell, and that cell is not currently a singleton (for otherwise there is nothing to be done), it sets the value in the

cell, and it adds the cell to the newly_singleton set that is returned, just as in propagate_cell. The portion of method that you need to write is about two dozen lines of code long.

In [24]:
```
### Exercise: write where_can_it_go

def sudoku_where_can_it_go(self):
    """Sets some cell values according to the where can it go
    heuristics, by examining all rows, colums, and blocks."""
    newly_singleton = set()

    # YOUR CODE HERE
    for i, row in enumerate(self.m):
      once = occurs_once_in_sets(row)
      for j, val in enumerate(row):
        if len(once.intersection(val)) > 0 and len(val) != 1:
          self.m[i][j] = once.intersection(val)
          newly_singleton.add((i,j))

    for j in range(9):
      col = []
      for row in self.m:
        col.append(row[j])
      once = occurs_once_in_sets(col)
      for i, val in enumerate(col):
        if len(once.intersection(val)) > 0 and len(val) != 1:
          self.m[i][j] = once.intersection(val)
          newly_singleton.add((i,j))

    for x in range(9):
      block = []
      ii = 3 * (x//3)
      jj = 3 * (x%3)
      for i in range(ii, ii+3):
        for j in range(jj, jj+3):
          block.append(self.m[i][j])
      once = occurs_once_in_sets(block)
      for k, val in enumerate(block):
        if len(once.intersection(val)) > 0 and len(val) != 1:
          i = ii + k//3
          j = jj + k%3
          self.m[i][j] = once.intersection(val)
          newly_singleton.add((i,j))


    # Returns the list of newly-singleton cells.
    return newly_singleton

Sudoku.where_can_it_go = sudoku_where_can_it_go
```

In [25]:
```
# Here you can write your own tests if you like.

# YOUR CODE HERE
sd = Sudoku.from_string('[[[5], [3], [1, 2, 4], [1, 2, 6], [7], [1, 2, 6, 8], [4, 8
print("Original:")
sd.show(details=True)
```

```
new_singletons = set()
new_s = sd.where_can_it_go()
new_s = sd.where_can_it_go()
```

```
Original:
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ __3______ 12_4_____|12___6___ ______7__ 12___6_8_|___4___89 12_4_____ _2_
____8_|
|_____6___ 1__4__7__ 12_4__7__|123______ ________9 ____5____|__34___8_ 12_4_____ _2_
___78_|
|12_______ ________9 _______8_|123______ ___4_____ 12_______|__3_5____ _____6___ _2_
___7__|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ 1___5____ 1___5___9|1_____7_9 _____6___ 1__4__7_9|___45____ _2_45____ __3
______|
|___4_____ _2_______ _____6___|_______8_ ____5____ __3______|______7__ ________9 1__
______|
|______7__ 1___5____ 1_3_5___9|1_______9 _2_______ 1__4____9|___45__8_ ___45____ ___
__6___|
+-----------------------------+-----------------------------+-----------------------
------+
|1_______9 _____6___ 1___5_7_9|____5_7_9 __3______ ______7_9|_2_______ _______8_ ___
4_____|
|_2______9 ______78_ _2____7_9|___4_____ 1________ _2____7_9|_____6___ __3______ ___
_5____|
|_23______ ___45____ _2345____|_2__56___ _______8_ _2___6___|1________ ______7__ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
```

Let us test it. We cannot test this code in one iteration only, since its result may depend on the order in which you apply the method to rows and columns. Rather, we apply the method until it can determine no more cell values.

In [ ]:

```
### Tests for where can it go

sd = Sudoku.from_string('[[[5], [3], [1, 2, 4], [1, 2, 6], [7], [1, 2, 6, 8], [4, 8
print("Original:")
sd.show(details=True)
new_singletons = set()
while True:
    new_s = sd.where_can_it_go()
    if len(new_s) == 0:
        break
    new_singletons |= new_s
assert_equal(new_singletons,
             {(3, 2), (2, 6), (7, 1), (5, 6), (2, 8), (8, 0), (0, 5), (1, 6),
              (2, 3), (3, 7), (0, 3), (5, 1), (0, 8), (8, 5), (5, 3), (5, 5),
              (8, 1), (5, 7), (3, 1), (0, 6), (1, 8), (3, 6), (5, 2), (1, 1)})
print("After where can it go:")
sd.show(details=True)
sdd = Sudoku.from_string('[[[5], [3], [1, 2, 4], [6], [7], [8], [9], [1, 2, 4], [2]
print("The above should be equal to:")
```

```
sdd.show(details=True)
assert_equal(sd, sdd)

sd = Sudoku([
    '___26_7_1',
    '68__7____',
    '1____45__',
    '82_1___4_',
    '__46_2___',
    '_5___3_28',
    '___3___74',
    '_4__5__36',
    '7_3_18___'
])
print("Another Original:")
sd.show(details=True)
print("Propagate once:")
sd.propagate_all_cells_once()
# sd.show(details=True)
new_singletons = set()
while True:
    new_s = sd.where_can_it_go()
    if len(new_s) == 0:
        break
    new_singletons |= new_s
print("After where can it go:")
sd.show(details=True)
sdd = Sudoku.from_string('[[[4], [3], [5], [2], [6], [9], [7], [8], [1]], [[6], [8]
print("The above should be equal to:")
sdd.show(details=True)
assert_equal(sd, sdd)
```

```
Original:
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ __3______ 12_4_____|12___6___ ______7__ 12___6_8_|___4___89 12_4_____ _2_
____8_|
|_____6___ 1__4__7__ 12_4__7__|123______ ________9 ____5____|__34___8_ 12_4_____ _2_
___78_|
|12_______ ________9 _______8_|123______ ___4_____ 12_______|__3_5____ _____6___ _2_
___7__|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ 1___5____ 1___5___9|1_____7_9 _____6___ 1__4__7_9|___45____ _2_45____ __3
______|
|___4_____ _2_______ _____6___|_______8_ ____5____ __3______|______7__ ________9 1__
______|
|______7__ 1___5____ 1_3_5___9|1_______9 _2_______ 1__4____9|___45__8_ ___45____ ___
__6___|
+-----------------------------+-----------------------------+-----------------------
------+
|1_______9 _____6___ 1___5_7_9|____5_7_9 __3______ ______7_9|_2_______ _______8_ ___
4_____|
|_2______9 ______78_ _2____7_9|___4_____ 1________ _2____7_9|_____6___ __3______ ___
_5____|
|_23______ ___45____ _2345____|_2__56___ _______8_ _2___6___|1________ ______7__ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
After where can it go:
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ __3______ 12_4_____|_____6___ ______7__ _______8_|________9 12_4_____ _2_
______|
|_____6___ ______7__ 12_4__7__|123______ ________9 ____5____|__3______ 12_4_____ ___
____8_|
|12_______ ________9 _______8_|__3______ ___4_____ 12_______|____5____ _____6___ ___
___7__|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ ____5____ ________9|1_____7_9 _____6___ 1__4__7_9|___4_____ _2_______ __3
______|
|___4_____ _2_______ _____6___|_______8_ ____5____ __3______|______7__ ________9 1__
______|
|______7__ 1________ __3______|________9 _2_______ ___4_____|_______8_ ____5____ ___
__6___|
+-----------------------------+-----------------------------+-----------------------
------+
|1_______9 _____6___ 1___5_7_9|____5_7_9 __3______ ______7_9|_2_______ _______8_ ___
4_____|
|_2______9 _______8_ _2____7_9|___4_____ 1________ _2____7_9|_____6___ __3______ ___
_5____|
|__3______ ___4_____ _2345____|_2__56___ _______8_ _____6___|1________ ______7__ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
The above should be equal to:
+-----------------------------+-----------------------------+-----------------------
```

```
------+
|____5____ __3______ 12_4_____|_____6___ ______7__ _______8_|________9 12_4_____ _2_______|
|_____6___ ______7__ 12_4__7__|123______ ________9 ____5____|__3______ 12_4_____ _______8_|
|12_______ ________9 _______8_|__3______ ___4_____ 12_______|____5____ _____6___ ______7__|
+-----------------------------+-----------------------------+-----------------------------+
|_______8_ ____5____ ________9|1_____7_9 _____6___ 1__4__7_9|___4_____ _2_______ __3______|
|___4_____ _2_______ _____6___|_______8_ ____5____ __3______|______7__ ________9 1________|
|______7__ 1________ __3______|________9 _2_______ ___4_____|_______8_ ____5____ _____6___|
+-----------------------------+-----------------------------+-----------------------------+
|1_______9 _____6___ 1___5_7_9|____5_7_9 __3______ ______7_9|_2_______ _______8_ ___4_____|
|_2______9 _______8_ _2____7_9|___4_____ 1________ _2____7_9|_____6___ __3______ ____5____|
|__3______ ___4_____ _2345____|_2__56___ _______8_ _____6___|1________ ______7__ ________9|
+-----------------------------+-----------------------------+-----------------------------+
Another Original:
+-----------------------------+-----------------------------+-----------------------------+
|123456789 123456789 123456789|_2_______ _____6___ 123456789|______7__ 123456789 1________|
|_____6___ _______8_ 123456789|123456789 ______7__ 123456789|123456789 123456789 123456789|
|1________ 123456789 123456789|123456789 123456789 ___4_____|____5____ 123456789 123456789|
+-----------------------------+-----------------------------+-----------------------------+
|_______8_ _2_______ 123456789|1________ 123456789 123456789|123456789 ___4_____ 123456789|
|123456789 123456789 ___4_____|_____6___ 123456789 _2_______|123456789 123456789 123456789|
|123456789 ____5____ 123456789|123456789 123456789 __3______|123456789 _2_______ _______8_|
+-----------------------------+-----------------------------+-----------------------------+
|123456789 123456789 123456789|__3______ 123456789 123456789|123456789 ______7__ ___4_____|
|123456789 ___4_____ 123456789|123456789 ____5____ 123456789|123456789 __3______ _____6___|
|______7__ 123456789 __3______|123456789 1________ _______8_|123456789 123456789 123456789|
+-----------------------------+-----------------------------+-----------------------------+
Propagate once:
After where can it go:
+-----------------------------+-----------------------------+-----------------------------+
```

```
|___4_____ __3______ ____5____|_2_______ _____6___ ________9|______7__ _______8_ 1__
______|
|_____6___ _______8_ _2_______|____5____ ______7__ 1________|___4_____ ________9 __3
______|
|1________ ________9 ______7__|_______8_ __3______ ___4_____|____5____ _____6___ _2_
______|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ _2_______ _____6___|1________ ________9 ____5____|__3______ ___4_____ ___
___7__|
|__3______ ______7__ ___4_____|_____6___ _______8_ _2_______|________9 1________ ___
_5____|
|________9 ____5____ 1________|______7__ ___4_____ __3______|_____6___ _2_______ ___
____8_|
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ 1________ 12__56_89|__3______ _2_______ _____6___|12_____89 ______7__ ___
4_____|
|_2_______ ___4_____ 12_____89|________9 ____5____ ______7__|12_____89 __3______ ___
__6___|
|______7__ _____6___ __3______|___4_____ 1________ _______8_|_2_______ ____5____ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
The above should be equal to:
+-----------------------------+-----------------------------+-----------------------
------+
|___4_____ __3______ ____5____|_2_______ _____6___ ________9|______7__ _______8_ 1__
______|
|_____6___ _______8_ _2_______|____5____ ______7__ 1________|___4_____ ________9 __3
______|
|1________ ________9 ______7__|_______8_ __3______ ___4_____|____5____ _____6___ _2_
______|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ _2_______ _____6___|1________ ________9 ____5____|__3______ ___4_____ ___
___7__|
|__3______ ______7__ ___4_____|_____6___ _______8_ _2_______|________9 1________ ___
_5____|
|________9 ____5____ 1________|______7__ ___4_____ __3______|_____6___ _2_______ ___
____8_|
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ 1________ 12__56_89|__3______ _2_______ _____6___|12_____89 ______7__ ___
4_____|
|_2_______ ___4_____ 12_____89|________9 ____5____ ______7__|12_____89 __3______ ___
__6___|
|______7__ _____6___ __3______|___4_____ 1________ _______8_|_2_______ ____5____ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
```

Let us try it now on a real probem. Note from before that this Sudoku instance could not be solved via propagate_cells alone:

In [27]:
```python
sd = Sudoku(sd_partially_solved)
newly_singleton = sd.where_can_it_go()
print("Newly singleton:", newly_singleton)
print("Resulting Sudoku:")
sd.show(details=True)
```

```
Newly singleton: {(3, 2), (2, 6), (7, 1), (5, 6), (2, 8), (8, 0), (5, 7), (0, 6),
(0, 5), (1, 6), (3, 6), (3, 7), (0, 3), (5, 2), (1, 1)}
Resulting Sudoku:
+-----------------------------+-----------------------------+-----------------------
------+
|____5____ __3______ 12_4_____|_____6___ ______7__ _______8_|________9 12_4_____ _2_
____8_|
|_____6___ ______7__ 12_4__7__|123______ ________9 ____5____|__3______ 12_4_____ _2_
___78_|
|12_______ ________9 _______8_|123______ ___4_____ 12_______|____5____ _____6___ ___
___7__|
+-----------------------------+-----------------------------+-----------------------
------+
|_______8_ 1___5____ ________9|1_____7_9 _____6___ 1__4__7_9|___4_____ _2_______ __3
______|
|___4_____ _2_______ _____6___|_______8_ ____5____ __3______|______7__ ________9 1__
______|
|______7__ 1___5____ __3______|1_______9 _2_______ 1__4____9|_______8_ ____5____ ___
__6___|
+-----------------------------+-----------------------------+-----------------------
------+
|1_______9 _____6___ 1___5_7_9|____5_7_9 __3______ ______7_9|_2_______ _______8_ ___
4_____|
|_2______9 _______8_ _2____7_9|___4_____ 1________ _2____7_9|_____6___ __3______ ___
_5____|
|__3______ ___45____ _2345____|_2__56___ _______8_ _2___6___|1________ ______7__ ___
_____9|
+-----------------------------+-----------------------------+-----------------------
------+
```

As we can see, the heuristics led to substantial progress. Let us incorporate it in the Sudoku solver.

In [28]:
```python
Sudoku.full_propagation = sudoku_full_propagation_with_where_can_it_go
```

Let us try again to solve a Sudoku example which, as we saw before, could not be solved by constrain propagation only (without using the *where can it go* heuristics). Can we solve it now via constraint propagation?

In [29]:
```python
sd = Sudoku([
    '53__7____',
    '6___95___',
    '_98____6_',
    '8___6___3',
    '4__8_3__1',
    '7___2___6',
    '_6____28_',
    '___41___5',
```

```
    '____8__79'
])
print("Initial:")
sd.show()
sd.full_propagation()
print("After full propagation with where can it go:")
sd.show()
```

```
Initial:
+---+---+---+
|53.|.7.|...|
|6..|.95|...|
|.98|...|.6.|
+---+---+---+
|8..|.6.|..3|
|4..|8.3|..1|
|7..|.2.|..6|
+---+---+---+
|.6.|...|28.|
|...|41.|..5|
|...|.8.|.79|
+---+---+---+
After full propagation with where can it go:
+---+---+---+
|53.|.7.|...|
|6..|.95|...|
|.98|...|.6.|
+---+---+---+
|8..|.6.|..3|
|4..|8.3|..1|
|7..|.2.|..6|
+---+---+---+
|.6.|...|28.|
|...|41.|..5|
|...|.8.|.79|
+---+---+---+
```

No! We still cannot! But if we compare the above with the previous attempt, we see that the heuristic led to much more progress; very few positions still remain to be determined via search.

Question 6: Solving some problems from example sites

Let us see how long it takes us to solve examples found around the Web. We consider a few from this site. You should be able to complete all of these tests in a short amount of time.

In [30]:
```
import time
```

Daily Telegraph January 19th "Diabolical"

In [31]:
```
# 5 points: You need to do this in less than 5.

sd = Sudoku([
    '_2_6_8___',
    '58___97__',
    '____4____',
    '37____5__',
    '6_______4',
    '__8____13',
    '____2____',
    '__98___36',
    '___3_6_9_'
])
t = time.time()
sd.solve()
elapsed = time.time() - t
print("Solved in", elapsed, "seconds")

assert elapsed < 5
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-21-8845f30832de> in occurs_once_in_sets(set_sequence)
     13             try:
---> 14                 sort[list(list(set_sequence)[i])[j]] += 1
     15             except:

KeyError: 8

During handling of the above exception, another exception occurred:

KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-31-78d01961da67> in <module>()
     13 ])
     14 t = time.time()
---> 15 sd.solve()
     16 elapsed = time.time() - t
     17 print("Solved in", elapsed, "seconds")

<ipython-input-18-48dd18f7f207> in sudoku_solve(self, do_print)
     42     """Wrapper function, calls self and shows the solution if any."""
     43     try:
---> 44         r = self.search()
     45         if do_print:
     46             print("We found a solution:")

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
```

```
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     13     """Tries to solve a Sudoku instance."""
```

```
     14     to_propagate = None if new_cell is None else {new_cell}
---> 15     self.full_propagation(to_propagate=to_propagate)
     16     if self.done():
     17         return self # We are a solution

<ipython-input-20-57e734969a01> in sudoku_full_propagation_with_where_can_it_go(sel
f, to_propagate)
     18         # Now we check whether there is any other propagation that we can
     19         # get from the where can it go rule.
---> 20         to_propagate = self.where_can_it_go()
     21 

<ipython-input-24-b91495a114be> in sudoku_where_can_it_go(self)
     18       for row in self.m:
     19         col.append(row[j])
---> 20       once = occurs_once_in_sets(col)
     21       for i, val in enumerate(col):
     22         if len(once.intersection(val)) > 0 and len(val) != 1:

<ipython-input-21-8845f30832de> in occurs_once_in_sets(set_sequence)
     14                 sort[list(list(set_sequence)[i])[j]] += 1
     15             except:
---> 16                 sort.setdefault(list(list(set_sequence)[i])[j], 1)
     17     for k in sort:
     18         if sort[k] == 1:

KeyboardInterrupt: 
```

Vegard Hanssen puzzle 2155141

In [32]:

```python
# 5 points: you need to do this in less than 5 seconds.

sd = Sudoku([
    '___6__4__',
    '7____36__',
    '____91_8_',
    '_________',
    '_5_18___3',
    '___3_6_45',
    '_4_2___6_',
    '9_3______',
    '_2____1__'
])
t = time.time()
sd.solve()
elapsed = time.time() - t
print("Solved in", elapsed, "seconds")
assert elapsed < 5
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-21-8845f30832de> in occurs_once_in_sets(set_sequence)
     13             try:
---> 14                 sort[list(list(set_sequence)[i])[j]] += 1
     15             except:

KeyError: 6

During handling of the above exception, another exception occurred:

KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-32-037ecf691fd5> in <module>()
     13 ])
     14 t = time.time()
---> 15 sd.solve()
     16 elapsed = time.time() - t
     17 print("Solved in", elapsed, "seconds")

<ipython-input-18-48dd18f7f207> in sudoku_solve(self, do_print)
     42     """Wrapper function, calls self and shows the solution if any."""
     43     try:
---> 44         r = self.search()
     45         if do_print:
     46             print("We found a solution:")

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
```

```
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
```

```
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     29         try:
     30             # If we find a solution, we return it.
---> 31             return sd.search(new_cell=(i, j))
     32         except Unsolvable:
     33             # Go to next value.

<ipython-input-18-48dd18f7f207> in sudoku_search(self, new_cell)
     13     """Tries to solve a Sudoku instance."""
```

```
     14     to_propagate = None if new_cell is None else {new_cell}
---> 15     self.full_propagation(to_propagate=to_propagate)
     16     if self.done():
     17         return self # We are a solution

<ipython-input-20-57e734969a01> in sudoku_full_propagation_with_where_can_it_go(sel
f, to_propagate)
     18         # Now we check whether there is any other propagation that we can
     19         # get from the where can it go rule.
---> 20         to_propagate = self.where_can_it_go()
     21

<ipython-input-24-b91495a114be> in sudoku_where_can_it_go(self)
     18       for row in self.m:
     19         col.append(row[j])
---> 20       once = occurs_once_in_sets(col)
     21       for i, val in enumerate(col):
     22         if len(once.intersection(val)) > 0 and len(val) != 1:

<ipython-input-21-8845f30832de> in occurs_once_in_sets(set_sequence)
     14                 sort[list(list(set_sequence)[i])[j]] += 1
     15             except:
---> 16                 sort.setdefault(list(list(set_sequence)[i])[j], 1)
     17     for k in sort:
     18         if sort[k] == 1:

KeyboardInterrupt:
```

A supposedly even harder one

source

In [ ]:
```
# 5 points: you need to do this in less than 10 seconds.

sd = Sudoku([
    '6____894_',
    '9____61__',
    '_7__4____',
    '2__61____',
    '______2__',
    '_89__2___',
    '____6___5',
    '_______3_',
    '8____16__'
])
t = time.time()
sd.solve()
elapsed = time.time() - t
print("Solved in", elapsed, "seconds")
assert elapsed < 10
```

Trying puzzles in bulk

Let us try the puzzles found at https://raw.githubusercontent.com/shadaj/sudoku/master/sudoku17.txt; apparently lines 517 and 6361 are very hard).

In [ ]:
```
import requests

r = requests.get("https://raw.githubusercontent.com/shadaj/sudoku/master/sudoku17.t
puzzles = r.text.split()
```

Let us convert these puzzles to our format.

In [ ]:
```
def convert_to_our_format(s):
    t = s.replace('0', '_')
    r = []
    for i in range(9):
        r.append(t[i * 9: (i + 1) * 9])
    return r
```

You need to solve these tests efficiently.

In [ ]:
```
# 5 points: you need to solve the first 1000 Sudokus in less than 30 seconds.

t = 0
max_d = 0.
max_i = None
t = time.time()
for i, s in enumerate(puzzles[:1000]):
    p = convert_to_our_format(puzzles[i])
    sd = Sudoku(p)
    sd.solve(do_print=False)
elapsed = time.time() - t
print("It took you", elapsed, "to solve the first 1000 Sudokus.")
assert elapsed < 30
```