Make sure you fill in any place that says `YOUR CODE HERE`.

Homework 12

This is a Python Notebook homework. It consists of various types of cells:

- Text: you can read them :-)
- Code: you should run them, as they may set up the problems that you are asked to solve.
- **Solution:** These are cells where you should enter a solution. You will see a marker in these cells that indicates where your work should be inserted.

```
# YOUR CODE HERE
```

- Test: These cells contains some tests, and are worth some points. You should run the cells as a way to debug your code, and to see if you understood the question, and whether the output of your code is produced in the correct format. The notebook contains both the tests you see, and some secret ones that you cannot see. This prevents you from using the simple trick of hard-coding the desired output.

Questions

There are several questions in this notebook; all of them contain the `YOUR CODE HERE` placeholder.

There are other pieces of text called "exercises", but you only have to do those that are explicitly marked with `YOUR CODE HERE`.

Working on Your Notebook

You can work directly on this notebook. The notebook that is shared with you is shared also with the TAs in case you need help.

Submitting Your Notebook

Submit your work as follows:

- Download the notebook from Colab, clicking on "File > Download .ipynb".
- Upload the resulting file to this Google form.
- **Deadline: see home page**

You can submit multiple times, and the last submittion before the deadline will be used to assign you a grade.

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.

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

```
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.

```
# 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
```

```
Collecting nose
  Downloading https://files.pythonhosted.org/packages/15/d8/dd071918c040f50fa1cf
     |████████████████████████████████| 163kB 4.5MB/s
Installing collected packages: nose
Successfully installed nose-1.3.7
```

```
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
|______6___ 123456789 123456789|1_________ _________9 ____5____|123456789 123456789
|123456789 _________9 ________8_|123456789 123456789 123456789|123456789 ______6___
+-------------------------------+-------------------------------+-------------------
|________8_ 123456789 123456789|123456789 ______6___ 123456789|123456789 123456789
|___4______ 123456789 123456789|________8_ 123456789 __3_______|123456789 123456789
|_______7__ 123456789 123456789|123456789 _2________ 123456789|123456789 123456789
+-------------------------------+-------------------------------+-------------------
```

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

```
|123456789 123456789 123456789|123456789 ________8_ 123456789|123456789 _______7__
```

```
1 sd1 = Sudoku(sd)
2 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.

Propagating a single cell

```
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
```

```
### 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 {}
    # 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
    s,e = (i - (i%3), j-(j%3))
    for ii in range(s,s+3):
            for jj in range(e,e+3):
                if  ii!=i or jj!=j:
                    newly_singleton.update(self.ruleout(ii,jj,x))
    # Returns the list of newly-singleton cells.
    return newly_singleton

Sudoku.propagate_cell = sudoku_propagate_cell
```

```
### Tests for cell propagation

tsd = Sudoku.from_string('[[[5], [3], [2], [6], [7], [8], [9], [1, 2, 4], [2]], [[6
tsd.show(details=False)
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)})

```

```
+---+---+---+
|532|678|9.2|
|67.|.95|3.8|
|.98|34.|567|
+---+---+---+
|859|.6.|423|
|426|853|791|
|713|924|856|
+---+---+---+
|.6.|.3.|284|
|.8.|41.|635|
|34.|.86|179|
+---+---+---+
+-----------------------------+-----------------------------+-----------------------------+
|____5____ __3_______ _2_______|_____6___ _______7__ _______8_|_________9 12_4_____ _____6___|
|_____6___ _______7__ 12_4__7__|123_______ _________9 ____5____|__3_______ 12_4_____ _____6___|
|12_______ _________9 _______8_|__3_______ ___4_____ 12_______|____5____ _____6___ _____7___|
+-----------------------------+-----------------------------+-----------------------------+
```

Propagating all cells, once

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

```
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):
            #print ( "All Cells " , i,j)
            self.propagate_cell((i, j))

Sudoku.propagate_all_cells_once = sudoku_propagate_all_cells_once
```

```
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
|______6___ _2_4__7__ _2_4__7__|1_________ _________9 _____5_____|__34__78_ _234______
|12_________ _________9 ________8_|_23_______ __34______ _2_4______|1_345_7__ ______6___
+-----------------------------+-----------------------------+-----------------------------+
|________8_ 12__5_____ 12__5___9|_____5_7_9 ______6___ 1__4__7__|___45_7_9 _2_45___9
|___4______ _2__5_____ _2__56__9|________8_ _____5_____ __3_______|_____5_7_9 _2__5___9
|_______7__ 1___5_____ 1_3_5___9|_____5___9 _2________ 1__4______|___45__89 ___45___9
+-----------------------------+-----------------------------+-----------------------------+
```

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

```
### 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."""
    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:
        x = to_propagate.pop()
        to_propagate.update(sudoku_propagate_cell(self,x))

Sudoku.full_propagation = sudoku_full_propagation
```

```
### 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.show()
sd.full_propagation()
sd.show()
sdd = Sudoku.from_string('[[[5], [3], [4], [6], [7], [8], [9], [1], [2]], [[6], [7]
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|
```

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

```
|534|678|912|
```

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:

```
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()
```

```
+---+---+---+
|53.|.7.|...|
|6..|.95|...|
|.98|...|.6.|
+---+---+---+
|8..|.6.|..3|
|4..|8.3|..1|
|7..|.2.|..6|
+---+---+---+
|.6.|...|28.|
|...|41.|..5|
|...|.8.|.79|
+---+---+---+
+---+---+---+
|53.|.7.|...|
|6..|.95|...|
```

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

```
|8..|.6.|..3|
```

```
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_____
|_____6___ 1__4__7__ 12_4__7__|123_______ _________9 ____5____|__34___8_ 12_4_____
|12_______ _________9 ________8_|123_______ ___4_____ 12_______|__3_5____ _____6___
+-----------------------------+-----------------------------+-----------------------------
|________8_ 1___5____ 1___5___9|1_____7_9 _____6___ 1__4__7_9|___45____ _2_45____
|___4_____ _2_______ _____6___|________8_ ____5____ __3_______|______7__ _________9
|______7__ 1___5____ 1_3_5___9|1________9 _2_______ 1__4____9|___45__8_ ___45____
+-----------------------------+-----------------------------+-----------------------------
|1________9 _____6___ 1___5_7_9|____5_7_9 __3_______ _______7_9|_2_______ ________8_
|_2_______9 _______78_ _2____7_9|___4_____ 1_________ _2____7_9|_____6___ __3_______
|_23______ ___45____ _2345____|_2__56___ ________8_ _2___6___|1_________ _______7__
+-----------------------------+-----------------------------+-----------------------------
```

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 guess 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.

```
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()
    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.

```
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.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.

Adding the where can it go heuristics

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 may be tempted to rewrite the code, and include in the *to_propagate* list of cells all cells whose set of possible values has shrunk. This may lead, however, to an inefficient implementation. When we modify one cell, there are up to 8 + 8 + 4 = 20 other cells whose values might have changed. We believe it is more efficient to first do propagation as before, based on singletons, and then apply the *where can it go* heuristics on the whole Sudoku board. The *where can it go* heuristic will return a (possibly empty) set of cells which have become singletons, and we can then propagate these.

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*

```
### 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.
        # YOUR CODE HERE
        while len(to_propagate) > 0:
            x = to_propagate.pop()
            to_propagate.update(sudoku_propagate_cell(self,x))
        # 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()

```

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 list 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.

```
### Exercise: define helper function to check once-only occurrence

from collections import defaultdict

def occurs_once_in_sets(set_sequence):
```

```
def occurs_once_in_sets(set_sequence):
    """Returns the elements that occur only once in the sequence of sets set_sequer
    The elements are returned as a set."""
    # YOUR CODE HERE
    C = defaultdict(int)
    for i in set_sequence:
        for j in i:
            C[j]+=1
    return {k for k,v in C.items() if v==1}
```

Double-click (or enter) to edit

Let us test it.

```
### Tests for once-only

from nose.tools import assert_equal

assert_equal(occurs_once_in_sets([{1, 2}, {2, 3}]), {1, 3})

```

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.

```
### 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()
```

```
    newly_singleton = set()
    # YOUR CODE HERE
    #All Rows
    T = []
    for i in range(9):
        for j in range(9):
            #List of all elements in rows
            T.append(self.m[i][j])
            #removes singelton values
            V = occurs_once_in_sets(T)-{v for k in T for v in k if len(k)==1}
            for k in V:
                if k in self.m[i][j]:
                    self.m[i][j]=={k}
                    #adds new singletons
                    newly_singleton|={(i,j)}
    #All Columns
    T = []
    for i in range(9):
        for j in range(9):
            T.append(self.m[j][i])
            V = occurs_once_in_sets(T)-{v for k in T for v in k if len(k)==1}
            for k in V:
                if k in self.m[j][i]:
                    self.m[j][i]=={k}
                    newly_singleton|={(j,i)}
    #All Boxes
    T = []
    for x in range(3):
        xx = x*3
        for y in range(3):
            yy = y*3
        for i in range(3):
            for j in range(3):
                T.append(self.m[xx+i][yy+j])
        V= occurs_once_in_sets(T)-{v for k in T for v in k if len(k)==1}
        for i in V:
            for j in range(3):
                for k in range(3):
                    if i in self.m[xx+j][yy+k]:
                        self.m[xx+j][yy+k]={i}
                        newly_singleton |={(xx+j, yy+k)}

    #returns the list of newly_singleton cells.
    return newly_singleton

Sudoku.where_can_it_go = sudoku_where_can_it_go
```

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.

```
### 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=False)
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)
```

```
assert_equal(sd, sdd)
```

```
Original:
+---+---+---+
|53.|.7.|...|
|6..|.95|...|
|.98|.4.|.6.|
+---+---+---+
|8..|.6.|..3|
|426|853|791|
|7..|.2.|..6|
+---+---+---+
|.6.|.3.|284|
|...|41.|635|
|...|.8.|179|
+---+---+---+
+-----------------------------+-----------------------------+-----------------------------+
|____5____ __3_______ 12_4_____|12___6___ _______7__ 12___6_8_|___4___89 12_4_____|
|_____6___ 1__4__7__ 12_4__7__|123______ _________9 ____5____|__34___8_ 12_4_____|
|12_______ _________9 ________8_|123______ ___4_____ 12_______|__3_5____ _____6___|
+-----------------------------+-----------------------------+-----------------------------+
|________8_ 1___5____ 1___5___9|1_____7_9 _____6___ 1__4__7_9|___45____ _2_45____|
|___4_____ _2_______ _____6___|________8_ ____5____ __3_____|______7__ _________9|
|______7__ 1___5____ 1_3_5___9|1________9 _2_______ 1__4____9|___45__8_ ___45____|
+-----------------------------+-----------------------------+-----------------------------+
|1________9 _____6___ 1___5_7_9|____5_7_9 __3_____ ______7_9|_2_______ _______8_|
|_2______9 _______78_ _2____7_9|___4_____ 1_______ _2____7_9|_____6___ __3_____|
|_23_____ ___45____ _2345____|_2__56___ _______8_ _2___6___|1_______ ______7__|
+-----------------------------+-----------------------------+-----------------------------+
```