```
# Sudoku program
# Author: Ronald L. Rivest
# Version 1.0
# Last modified: January 14, 2006

# This program is "public domain": you may copy, modify, run,
# publish, distribute, embed, and/or use this program in any way
# whatsoever without prior permission and/or giving me any
# credit or payment whatsoever.
# (Indeed, please don't bother me to ask such permission!)
# It comes "as is" with no warranties or guarantees of any kind.
# It may contain bugs and/or be inconsistent with its internal
# or usage documentation.
# Have fun with it!
# (Of course, this really needs a nice GUI, rather than this
# console interface!)

usage_string = """\

Usage: sudoku.py <processing_options> <source_options>

   <processing_options> may include one or more of:
       (note that printing out puzzle always happens)
       -f         to save this puzzle (only useful with -g option)
                  filename is random number seed used to generate puzzle
                  (puzzle is saved in format suitable for reading in later,
                  in the "puzzles" subdirectory, as a ".txt" file)
       -s         to solve the puzzle, and print out solution (if possible)
                  and indicate whether there are no solutions, whether
                  there is a unique solution, or whether there are multiple
                  solutions.  If a solution exists, one will be printed.
                  (Solving puzzle is optional, so as not to spoil fun for puzzle
                  you only want rated.)
       -d         to turn on "debug printing"
                  This will explain in detail how a particular
                  puzzle can be solved, step by step, with reasons for
                  each step, or how a puzzle was generated.
       -r         to rate the puzzle's difficulty
                  by solving it 20 times
                  and returning the median "score" (measured as
                    moves_made + 10*guesses_made + 50*guess_depth
                  a somewhat arbitrary measure).
                  A score equal to the number of unfilled squares means
                  that the puzzle requires no backtracking to solve.
       If no processing options are given, defaults to -s -r (solve and rate).
       Processing options don't really need to be first; they may be
       anywhere in argument list, but are treated as if they were first.

   <source_options> specifies one or more puzzles:
       -gxxx      to generate a puzzle
                  using arbitrary length string xxx as random number seed
                  This allows you to regenerate the same
                  puzzle again easily later.
       -g         to generate a random puzzle
                  This uses clock as random number seed.
       -i         to read puzzle from standard input
       filename   to read puzzle from file with given filename
                  from the "puzzles" subdirectory (.txt file)

       Some "automation" you may find convenient:
       The -gxxx and filename options may include, at the end,
       a "+r" extension, where r is a decimal integer > 0 which
       specifies how many extra similar generations or filenames
       to process.  This is like a macro which expands by counting:
```

```
          -gabc+3   is the same as saying -gabc -gabc1 -gabc2 -gabc3
          ttt+5     is the same as saying ttt ttt1 ttt2 ttt3 ttt5 ttt5
          -g+6      is the same as saying -g -g -g -g -g -g -g
       If you have a directory of puzzles, with filenames indicating page
       numbers, say, as in page1 page2 ... page100, you can solve them all
       by specifying page1+99 .

Each puzzle input file represents "empty" by ".", otherwise uses 1--9.
If a line contains a sharp sign (#), the # and all following characters
on that line are ignored.  Characters other than . or 1-9, including blanks,
are also ignored.  Here is an example input file:
    # page 1 of puzzle book
    5.7...3..
    ...2...5.
    2.3.9.1..
    .2.75.8..
    .9...4.6.
    1...8....
    3..5....1
    .....3...
    4....19.5
At the moment, only 9x9 puzzles are supported.
All puzzles are read/saved in the "puzzles" subdirectory of the current directory.

Examples:
    sudoku.py -g             # generate, solve, and rate a random puzzle
    sudoku.py -g -f          # generate a random puzzle and save it
    suodku.py -g -s          # generate a random puzzle and solve it
    sudoku.py -g -r          # generate a random puzzle and rate it
    sudoku.py -ga12          # generate and print random puzzle with seed a12
    sudoku.py -ga12 -s -r    # solve and rate random puzzle generated from seed a12
    sudoku.py -r pa pb       # rate puzzles in files "pa" and "pb"
    sudoku.py -s pa pb       # solve puzzles in files "pa" and "pb"
    sudoku.py pa -d -s       # solve and detail reasoning for solving puzzle "pa"
    sudoku.py -s pa1+3       # solve puzzles pa1 pa2 pa3 pa4
    sudoku.py -g+10          # generate, rate, and solve 10 puzzles
"""

# n is the size of the square and the number of digits in the sudoku alphabet
# a block has nr rows and nc columns, where nr*nc = n
# row indices are in [1,...,n]
# col indices are in [1,...,n]
# blk indices are in [1,...,n] (across rows first, then down)
# values are in [1,...,n] or UNDEFINED (0)

import profile
import random
import string
import sys
import time

# Some global constants
UNDEFINED = 0               # undefined value represented by 0
rating_times = 20           # number of times to solve puzzle to get rating

class Pos:
    """
    Implements a sudoku position in the grid, i.e. an individual square.
    """

    def __init__(self,r,c):
        """
        Construct/initialize position in row r and col c, 1<=r,c<=n.
        """
        # STATIC attributes defining general structure of problem
```

```
        self.r = r              # row index 1<=r<=n
        self.c = c              # col index 1<=c<=n
        # self.row              # will contain pointer to row object (group)
        # self.col              # will contain pointer to col object (group)
        # self.blk              # will contain pointer to blk object (group)
        # self.grps             # will contain [row,col,blk]

        # DYNAMIC attributes that change as puzzle is worked on.
        # These two attributes are the ONLY values in this program
        # that are dynamic in this sense, so they are the only values
        # that need to be saved/restored for backtracking.
        self.val = UNDEFINED
        self.possibilities = range(1,n+1)

class Group:
    """
    Implements a group (i.e. row, col, or block);
    a set of n positions constrained to have each value exactly once.
    """

    def __init__(self,gt,i):
        """
        Construct and initialize a group.

        gt = "row","col", or "blk", respectively (group_type)
        i = 1 to n, inclusive, giving index of group within group_type
        """

        self.group_type = gt  # "row", "col", or "blk"
        self.index = i        # 1 to n
        self.posns = []       # positions in this group,
                              # to be filled in later cross-linking


# Some global variables
#
# T is (n+1)x(n+1) array (0-th col and 0-th row unused)
# T[i][j] is Pos for row i, column j
#
# sudoku_alphabet[i] is the printed version of the i-th symbol
#
# rows = list of all rows
# cols = list of all cols
# blks = list of all blocks
# grps = list of all groups (rows, cols, or blocks)
# posns = list of all positions

def initialize(n0,nr0,nc0):
    """
    initialize(n) -- set up for table of size n with nr x nc blocks
    """
    global n, nr, nc
    global posns, rows, cols, blks, grps
    global T, sudoku_alphabet

    # save and check n, nr, nc
    n = n0
    nr = nr0
    nc = nc0
    assert n >= 1
    assert nr >= 1
    assert nc >= 1
    assert n == nr * nc

    assert n < 36
```

```
    sudoku_alphabet = ".123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"[:n+1]

    # create all positions
    # array of posns T is actualy (n+1) x (n+1)
    posns = []
    T = [ [0]*(n+1) for r in range(n+1) ]
    for r in range(1,n+1):
        for c in range(1,n+1):
            p = Pos(r,c)
            T[r][c] = p
            posns.append(p)

    # create all groups
    rows = [Group("row",i) for i in range(1,n+1)]
    cols = [Group("col",i) for i in range(1,n+1)]
    blks = [Group("blk",i) for i in range(1,n+1)]
    grps = rows+cols+blks

    # now cross-link positions and groups
    for row in rows:
        r = row.index
        for col in cols:
            c = col.index
            p = T[r][c]
            # link pos p to groups containing it
            p.row = row
            p.col = col
            blk = blks[((r-1)/nr)*nr + (c-1)/nc]
            p.blk = blk
            p.grps = [p.row,p.col,p.blk]
            # and reverse
            row.posns.append(p)
            col.posns.append(p)
            blk.posns.append(p)

def print_puzzle(file = sys.stdout):
    """
    Print out the current puzzle configuration, nicely.
    """
    global T
    print_block_lines = True
    for r in range(1,n+1):
        line = ""
        for c in range(1,n+1):
            line = line + " " + sudoku_alphabet[T[r][c].val] + " "
            if print_block_lines and c<n and c%nc == 0:
                line = line + "|"
        print >>file,line
        if print_block_lines and r<n and r%nr == 0:
            print >>file,"-"*(3*n+n/nc-1)
    file.flush()

def set_pos(r,c,v):
    """
    Set position at row r col c to have value v.
    Value v may be UNDEFINED (0) or in [1,...,n].
    """
    assert 1<=r<=n
    assert 1<=c<=n
    assert 0<=v<=n
    global T, debug_print
    p = T[r][c]
    p.val = v
    # if not setting p to UNDEFINED,
    # set p's possibility list to just [v]
```

```
    # and remove v from possibilities for other positions q in same groups g as p
    if v == UNDEFINED:
        return
    p.possibilities = [v]
    for g in p.grps:
        for q in g.posns:
            if p != q and v in q.possibilities:
                q.possibilities = q.possibilities[:]
                q.possibilities.remove(v)
                # print "remove", q.r, q.c, v, "due to", p.r, p.c, v
                # CHECK FOR UNSOLVABILITY:
                if len(q.possibilities)==0:
                    if debug_print:
                        print "UNSOLVABLE!"
                    raise "UNSOLVABLE"

def trim(x,alphabet):
    """
    Return input string x with all characters not in alphabet removed.
    """
    s = ""
    for a in x:
        if a in alphabet:
            s += a
    return s

def initialize_from_rows(s):
    """
    s is an array of strings, one per row.
    """
    global n
    initialize(9,3,3)                 # assumes 9x9 for now
    s = [ trim(string.split(x,"#")[0],sudoku_alphabet) for x in s ]  # eliminate comments
    s = [ x for x in s if len(x)>0 ]
    if len(s) != n:
        print "Input does not have %d rows"%n
        print s
        assert len(s) == n
    for r in range(1,n+1):
        row = s[r-1]
        for c in range(1,n+1):
            while len(row)>0 and row[0] not in sudoku_alphabet:
                row = row[1:]
            assert len(row)>0
            val = row[0]
            val = sudoku_alphabet.find(val)
            assert val>=0
            set_pos(r,c,val)
            row = row[1:]

def initialize_from_string(s):
    """
    Input s is a string defining input sudoku puzzle.
    Each line describes one row using digits 1--9 and "." for empty
    """
    s = string.split(s)
    initialize_from_rows(s)

def initialize_from_file(puzzle_filename):
    """
    Input puzzle from file.
    Leading and trailing blanks on each line are ignored.
    Comment lines whose first nonblank is a sharp are ignored.
    Example file:
        # page 1 of puzzle book
```

```
        5.7...3..
        ...2...5.
        2.3.9.1..
        .2.75.8..
        .9...4.6.
        1...8....
        3..5....1
        .....3...
        4....19.5
    Assumes puzzle is in directory "puzzles".
    """
    # First remove any illegal characters from filename
    filename_alphabet = string.letters + string.digits + "-_+~."
    puzzle_filename = trim(puzzle_filename,filename_alphabet)
    assert len(puzzle_filename)>0
    try:
        puzzle_file = open("puzzles/"+puzzle_filename+".txt","r")
    except:
        print "Puzzle file '"+puzzle_filename+"' doesn't exist!"
        raise "puzzle filename error"
        return
    lines = puzzle_file.readlines()
    initialize_from_rows(lines)
    puzzle_file.close()

def write_to_file(puzzle_filename,header_string):
    """
    Write current puzzle to the specified file,
    with header_string on first line, commented out.
    """
    try:
        puzzle_file = open("puzzles/"+puzzle_filename+".txt","w")
    except:
        print "Can't open puzzle file",puzzle_filename,"for writing!"
        return
    print >>puzzle_file,"#",header_string
    print_puzzle(puzzle_file)
    puzzle_file.close()
    return

def save_state():
    """
    Return current state of puzzle solving;
    that is, return list of triples:  p, p's value, p's possibilities
    """
    global posns
    return [(p,p.val,p.possibilities[:]) for p in posns]

def restore_state(old_state):
    """
    Restore puzzle state to old_state.
    """
    global posns
    for (p,p_val,p_possibilities) in old_state:
        p.val = p_val
        p.possibilities = p_possibilities[:]

def number_of_undefined_posns():
    """
    Return the number of unfilled (undefined) positions.
    """
    global posns
    return len( [p for p in posns if p.val == UNDEFINED] )

def compute_possibilities():
```

```
    """
    Compute possible values for each position.
    These are based ONLY on excluding values already
    established within the same group (row, col, or blk).
    This is a "recompute from scratch" operation.
    """
    global posns, debug_print
    for p in posns:
        if p.val == UNDEFINED:
            p.possibilities = range(1,n+1)
            for g in p.grps:
                for q in g.posns:
                    if q.val in p.possibilities:
                        p.possibilities = p.possibilities[:]
                        p.possibilities.remove(q.val)
                        # CHECK FOR UNSOLVABILITY:
                        if len(p.possibilities)==0:
                            if debug_print:
                                print "UNSOLVABLE!"
                            raise "UNSOLVABLE"
        else:
            p.possibilities = [p.val]

def S1():
    """
    Strategy 1:
    If a position has only one possibility, choose it.
    Returns True if something was forced, else returns False
    Stops after forcing the first forcable position.
    """
    global debug_print
    for p in posns:
        if p.val == UNDEFINED and len(p.possibilities) == 1:
            set_pos(p.r,p.c,p.possibilities[0])
            if debug_print:
                print
                print "S1", "Row", p.r, "Col", p.c, "can only have a", p.possibilities[0]
                print_puzzle()
            return True
    return False

def S2():
    """
    Strategy 2:
    If for some group there is a value that can only go in
    one place in that group, then chose it.
    Returns True if something was forced, else returns False
    """
    global debug_print
    for g in grps:
        for val in range(1,n+1):
            cango = [p for p in g.posns if p.val == UNDEFINED and val in p.possibilities]
            if len(cango) == 1:
                p = cango[0]
                set_pos(p.r,p.c,val)
                if debug_print:
                    print
                    print "S2","In",g.group_type,g.index,",",val,"must go in
row",p.r,"col",p.c
                    print_puzzle()
                return True
    return False

def S3():
    """
```

```
    Strategy 3:
    If for some group g1 there is a value whose only possibilities
    within g1 also lie in some other group g2, then that val may not
    go in positions in g2 that lie outside of g1.
    Returns True if something was changed, else returns False
    """
    global debug_print
    something_changed = False
    return False
    ### BASED ON PROFILING STUDIES, IT SEEMS THAT
    ### S3 IS NOT WORTH ALL THE TIME IT TAKES... SO S3 IS DISABLED.
    ### OMIT PREVIOUS LINE TO RESTORE S3 TO FUNCTIONING.
    for g1 in grps:
        for val in range(1,n+1):
            cango = [p for p in g1.posns if p.val == UNDEFINED and val in p.possibilities]
            if len(cango)>1:
                for g2 in cango[0].grps:
                    g2OK = True
                    for pos in cango:
                        if g2 not in pos.grps:
                            g2OK = False
                    if g2OK:
                        # now val can go several places within g1
                        # but those places are all within g2 as well
                        # so remove val from places within g2 outside of g1
                        for pos in g2.posns:
                            if not g1 in pos.grps and val in pos.possibilities:
                                pos.possibilities = pos.possibilities[:]
                                pos.possibilities.remove(val)
                                if debug_print:
                                    print
                                    print "S3","row",pos.r,"col",pos.c, "may not be",val
                                    print "because in",g1.group_type,g1.index
                                    print "all possibilities for",val, "lie
within",g2.group_type,g2.index
                                    print_puzzle()
                                something_changed = True
                                # CHECK FOR UNSOLVABILITY HERE:
                                if len(pos.possibilities)==0:
                                    if debug_print:
                                        print "UNSOLVABLE!"
                                    raise "UNSOLVABLE"
    return something_changed


def S4():
    """
    Strategy 4:
    If for some group there are two positions that have the
    exact same two possible values, then those values can not go
    other places within that group
    Returns True if something was changed, else returns False
    """
    global debug_print
    something_changed = False
    for g in grps:
        for p in g.posns:
            if len(p.possibilities) ==2:
                for q in g.posns:
                    if q!=p and q.possibilities == p.possibilities:
                        for val in p.possibilities:
                            for pos in g.posns:
                                if pos!=p and pos!=q and val in pos.possibilities:
                                    pos.possibilities = pos.possibilities[:]
                                    pos.possibilities.remove(val)
```

```
                                    if debug_print:
                                        print "S4","row",pos.r,"col",pos.c,"may not be",val
                                        print "because in",g.group_type,g.index
                                        print "positions at
row",p.r,"col",p.c,"and","row",q.r,"col",q.c
                                        print "have the same possibilities:",p.possibilities
                                        print_puzzle()
                                    something_changed = True
                                    # CHECK FOR UNSOLVABILITY HERE
                                    if len(pos.possibilities)==0:
                                        if debug_print:
                                            print "UNSOLVABLE!"
                                        raise "UNSOLVABLE"
    return something_changed


def make_forced_move():
    """
    Find a forced move and make it. (at most one move made)
    Return True if a forced move was found, else return False.
    Strategies S1 and S2 make forced moves;
    strategies S3 and S4 are only used if necessary,
       to update possibilities; they don't actually make a move,
       but facilitate S1 and S2's operations.
    All these strategies return True if they make progress.
    """
    keep_going = True
    while keep_going:
        if S1() or S2():
            return True
        keep_going = S3() or S4()
    return False

def solve(numwanted, depth = 0, solns = []):
    """
    Solve currently established puzzle.
    return list of length 0,...,numwanted giving up to numwanted solutions found
    (typically numwanted will be 1 or 2)
    depth is recursion (guess) depth level so far;
    initial top-level call supplies depth = 0
    solns gives solutions found so far in other parts of search tree
    """
    global debug_print, moves_made, guesses_made, guess_depth

    assert len(solns)<numwanted

    if depth == 0:  # initial call
        moves_made = 0
        guesses_made = 0
        guess_depth = 0

    guess_depth = max(depth,guess_depth)

    # first solve puzzle if possible using only forced moves
    while make_forced_move():
        moves_made += 1

    if number_of_undefined_posns() == 0:
        if debug_print:
            print "Puzzle solved. Solution:"
            print_puzzle()
        solns = solns + [save_state()]
        return solns

    if debug_print:
```

```
        print depth, "Puzzle only partially solved. Current configuration:"
        print_puzzle()

    # guess and recurse; find a random position p with fewest possibilities
    # Note that we are sorting triples with p.r and p.c rather than
    # pairs with just p, so that sort always returns same answer for same input
    # (otherwise sort order may depend on internal memory layout, etc.)
    L = [(len(p.possibilities),p.r,p.c) for p in posns if len(p.possibilities)>1]
    L.sort()
    numpos = L[0][0]
    L = [x for x in L if x[0] == numpos]
    L = shuffle(L)
    r = L[0][1]
    c = L[0][2]
    p = T[r][c]
    # guess and recurse
    state = save_state()
    for val in shuffle(p.possibilities):
        # print depth,"TRYING by guessing",val,"(from",p.possibilities,") for
row",p.r,"column",p.c
        if debug_print:
            print depth,"TRYING by guessing",val,"(from",p.possibilities,") for
row",p.r,"column",p.c
        set_pos(p.r,p.c,val)
        moves_made += 1
        guesses_made += 1
        try:
            solns = solve(numwanted,depth+1,solns)
            if len(solns) == numwanted:
                return solns
        except:
            pass
        restore_state(state)
        if debug_print:
            print depth,"FINISHED guessing",val,"(from",p.possibilities,") for
row",p.r,"column",p.c
    return solns

def shuffle(L):
    """
    Return a random permutation of a copy of the list L.
    """
    L = L[:]
    random.shuffle(L)
    return L

# Some puzzles from books that have been solved with this program:
# Pocket Sudoku (ps): ps1, ps13, ps50, ps76, ps101, ps127, ps147, ps148
# The Ultimate Sudoku Challenge (tusc): tusc23, tusc88, tusc90, tusc190
# The Book of Sudoku, No 2 (bos2p): bos2p132, bos2p21

def simplify_puzzle():
    """
    "Simplify" puzzle by removing assignments to positions
    if this preserves the unique puzzle solution.
    (The puzzle is "simpler" only in that it is now less filled in;
    it is probably harder to solve.)
    """
    for p in posns:
        if p.val != UNDEFINED:
            state = save_state()
            val = p.val
            set_pos(p.r,p.c,UNDEFINED)
            compute_possibilities()
            nsolns = len(solve(2))
```

```
            if nsolns == 1:
                if debug_print:
                    print "Removing:",p.r,p.c,val, "OK:"
            restore_state(state)
            if nsolns == 1:
                set_pos(p.r,p.c,UNDEFINED)
                compute_possibilities()
                if debug_print:
                    print "Simplified puzzle:"
                    print_puzzle()


def generate_puzzle():
    """
    Make up a sudoku puzzle.
    """
    global posns, guess_depth, debug_print, moves_made, guesses_made
    initialize_from_string(".........\n"*9)
    # pick and fill in random positions that have more than one possibility
    for p in [p for p in shuffle(posns) if len(p.possibilities)>1]:
        state = save_state()
        if debug_print:
            print "working on position",p.r,p.c
            print "p.possibilities = ",p.possibilities
            print_puzzle()
        # Find a value v that is results in at least one soln at position p
        for v in shuffle(p.possibilities):
            restore_state(state)
            try:
                set_pos(p.r,p.c,v)
                if debug_print:
                    print "Setting:",p.r, p.c, v
                    print_puzzle()
                nsolns = len(solve(2))
                if debug_print:
                    print "Setting:",p.r, p.c, v, "-->", nsolns, "solutions"
            except:
                if debug_print:
                    print "this puzzle state is unsolvable... exception raised."
                nsolns = 0
            if nsolns == 0:
                # this value of v didn't work out; keep looking...
                continue
            elif nsolns==1:
                if debug_print:
                    print "Here is puzzle:"
                restore_state(state)
                set_pos(p.r,p.c,v)
                simplify_puzzle()
                if debug_print:
                    print "Final (simplified) puzzle:"
                    print_puzzle()
                return
            else:
                # more than one solution exists;
                # set v at p and keep going with next p
                restore_state(state)
                set_pos(p.r,p.c,v)
                break


def score(moves_made,guesses_made,guess_depth):
    """
    Compute a score for a puzzle, based on parameters from solving it.
    """
```

```
    return moves_made + 10*guesses_made + 50 * guess_depth

def parse_arg(s):
    """
    Here input s is a string of the form tu+r or tu or just t
    where
       t is a possibly empty string not ending in a digit
       u is a maximal length (possibly zero length) string of digits
       r is a (possibly zero length) string of decimal digits
    Return (t,int(u),int(r))  (string, int, int)
    if u or r is missing then a zero value is return for those components.
    Example: "x3y4+5" returns ("x3y",4,5)
    """
    t = ""
    u = ""
    r = ""
    tu_r = s.split("+")
    tu = tu_r[0]
    if len(tu_r)>1:
        r = tu_r[1]
    try:
        intr = int("0"+r)
        assert 0<=intr<=1000
    except:
        print "Illegal repetition count",r,"ignored."
        intr = 0
    for c in tu:
        u = u + c
        if not c.isdigit():
            t = t + u
            u = ""
    return (t,int("0"+u),intr)

def main():
    """
    main routine to interpret command-line arguments.
    """
    global debug_print, guesses_made, guess_depth, moves_made
    if len(sys.argv)==1:
        print usage_string
        return

    # first collect processing options, wherever they are
    processing_options_specified = False
    solve_puzzle = False
    rate_puzzle = False
    save_puzzle = False
    debug_print = False
    for arg in sys.argv[1:]:
        if arg[:2] == "-d":
            debug_print = True
        elif arg[:2] == "-s":
            solve_puzzle = True
            processing_options_specified = True
        elif arg[:2] == "-r":
            rate_puzzle = True
            processing_options_specified = True
        elif arg[:2] == "-f":
            save_puzzle = True
            processing_options_specified = True
    if not processing_options_specified:
        solve_puzzle = True
        rate_puzzle = True

    # now process each puzzle specified
```

```
header_string = ""                # file header line for file output option
arglist = sys.argv[1:]
while len(arglist)>0:
    arg = arglist[0]
    arglist = arglist[1:]
    if arg[0] == "-" and arg[:2] not in ["-g","-i"]:
        # processing options have already been handled; skip here
        continue
    print                         # blank line on output starts each puzzle
    if arg[:2] == "-g":
        # -gxxx+yy set seeds to xxx and generates yy additional puzzles
        # e.g. -gxx5+3 ==> uses seeds   xx5 xx6 xx7 xx8
        # e.g. -gxx+4 ==> uses seeds    xx xx1 xx2 xx3 xx4
        (t,u,r) = parse_arg(arg[2:])
        if t == "" and u == 0:
            t = "%d"%int(time.time())
        if u>0:
            seed_string = "%s%d"%(t,u)
        else:
            seed_string = t
        random.seed(seed_string)
        generate_puzzle()
        header_string = "Puzzle generated from seed '" + seed_string + "' :"
        print header_string
        print_puzzle()
        state = save_state()
        filename = seed_string
        # now prepare for iteration, by stuffing new arg at front of
        # arglist that parses to (t,u+1,r-1)
        if r>0:
            new_arg = "-g%s%d+%d"%(t,u+1,r-1)
            arglist = [new_arg]+arglist
    elif arg[:2] == "-i":
        print "Enter puzzle, followed by a blank line:"
        s = ""
        while True:
            line = sys.stdin.readline()
            line = string.strip(line)
            if line == "":
                break
            s = s + line + "\n"
        initialize_from_string(s)
        state = save_state()
        filename = "I%d"%random.randrange(1,999999999)
        header_string = "Puzzle read from input:"
        print header_string
        print_puzzle()
    else:
        # abc+2 generates abc abc1 abc2
        (t,u,r) = parse_arg(arg)
        if t == "" and u == 0:
            print "argument", arg, "illegal; ignored."
            continue
        if u>0:
            filename = "%s%d"%(t,u)
        else:
            filename = arg
        try:
            initialize_from_file(filename)
        except:
            continue
        header_string = "Puzzle read from file: " + filename
        print header_string
        print_puzzle()
        state = save_state()
```

```
            # now prepare for iteration, by stuffing new arg at front of
            # arglist that parses to (t,u+1,r-1)
            # e.g. abc3+7 generates filename abc3 and puts abc4+6 at front of arglist
            if r>0:
                new_arg = "%s%d+%d"%(t,u+1,r-1)
                arglist = [new_arg]+arglist
        if solve_puzzle:
            solns = solve(2)    # ask for up to 2 solns, to check for uniqueness
            if len(solns)==0:
                print "NO SOLUTION EXISTS"
            elif len(solns)==1:
                print "Puzzle solution:"
                restore_state(solns[0])
                print_puzzle()
                print "SOLUTION IS UNIQUE"
            else:
                print "MULTIPLE SOLUTIONS EXIST; HERE IS ONE SOLUTION:"
                restore_state(solns[0])
                print_puzzle()
        if rate_puzzle:
            score_list = []
            random.seed(1) # so rating is deterministic function of puzzle
            for i in range(rating_times):
                restore_state(state)
                solve(1)
                score_list += [score(moves_made,guesses_made,guess_depth)]
            score_list.sort()
            rating = score_list[rating_times/2]
            print "Puzzle rating:",rating,
            restore_state(state)
            if rating == number_of_undefined_posns():
                print "(easy)"
            elif rating < score(n*n,5,1):
                print "(moderate)"
            else:
                print "(hard)"
        if save_puzzle:
            if arg[0] != "-":
                print "Puzzle read from file '"+arg+"' will not be re-saved."
            else:
                write_to_file(filename,header_string)
                print "Puzzle saved to file: ",filename

# profile.run("main()")
main()
```