# CSC 110 Honors Project Final Writeup

## Overview

My program allows a user to play through simple “Einstein’s Riddle” style, grid-based logic puzzles. This style of puzzle typically involves three categories, in my case one subject and two adjectives, with grids that represent the relationship between any given pair of these categories. To solve the puzzle, a list of logic-based, textual clues is provided that can be solved through deductive reasoning. For example, if a subject was a person, say, John, who had blue eyes and brown hair you would mark the intersection of the columns for John and blue eyes, John and brown hair, and blue eyes and brown hair as solutions. For more information about this type of puzzle, visit this website.

## Functionality

When this program is started, it selects two random adjective categories that could describe a person, such as ‘eye color’ and ‘language spoken’, and chooses four adjectives from within each selected category randomly from a file, which could be (‘Amber’, ‘Blue’, ‘Gray’, ‘Green’) and (‘Arabic’, ‘English’, ‘French’, and ‘Spanish’’) in this example. After this, it selects four random names from a file to act as the subjects that are described by these adjectives, like (‘Brody’, ‘Candice’, ‘Keaton’, ‘Talia’).

After finalizing the choice of the aforementioned adjective categories and subjects, the program begins to generate all possible permutations of basic clues using these categories, encoding each clue as both human-readable text and computer-understandable logic, in JSON format. A possible clue that could be generated in this case is {text: “The person with Green eyes is Brody.”, logic: [0,0,3,0]}.

After generating all possible clues, two constraint-satisfaction problem solvers are initialized, one for each adjective. One at a time, random clues are added or removed as constraints to these solvers, until exactly one unique solution is found for each adjective-subject pair. This means that, given this set of clues, the puzzle is fully defined, and a human should be able to solve it without issue.

However, there are two problems with this raw set of clues. For one, they only involve basic logical statements and relationships, such as “Subject A has X Y”, “Subject B does not have X Y”. Secondly, there may be redundant clues that are not required to solve the problem. To address these issues, the program executes two functions back-to-back. The first is what I call the consolidation step, which adds random compound clues that match the solution set, such as “The subject with X Y has U V”. The second function reduces this new set of clues to the least number possible required to solve the puzzle. It accomplishes this by removing one clue at a time from the set and checking if the solution space is identical to its previous state. If it is, the clue is redundant and removed from the set. At this point, the set of clues is ready to be presented to the user. A function prints out an ordered list of the textual representation of the clues, wishing the user good luck.

After this, a series of steps are taken by the program in preparation for the start of the game. The first thing that happens is the game-board/grid is initialized as a 3D array, with each index set to 0, which represents a blank, unmarked space. Then, two threads are created and started. One runs the graphical user interface, and the other runs the command line interface. Each of these runs parallel to each other, so the user can interact with both at the same time, except for during conditions outlined later on.

The graphical user interface in this program utilizes a slightly modified version of ‘graphic.py’ used in class. During each draw loop, the various parts of the gameboard are drawn on the screen. This includes the titles of each category, the grid lines (both bolded and standard), and the player’s marks. To create a mark, the user must click on the square in the grid they want to place their mark on, which is ‘O’ on a right-click, which represents the solution (like a flag in minesweeper), and ‘X’ on a left-click, which represents an index that cannot be part of the solution set (like clearing tiles in minesweeper). Either of these symbols can be toggled back to blank through a left-click action as well. The way the program accomplishes this is through the 3D grid array mentioned earlier. When an ‘O’ is placed, the corresponding index in the grid array is set to 2, and for an ‘X’, it is set to 1. At the start of each graphics loop, this grid array is referenced to figure out what to draw at each grid location.

The command-line interface provides the user with several useful tools when running the game. When a user enters text into the command window and submits it (presses enter), the program parses and sanitizes the user’s input and attempts to determine the user's intended command. The accepted commands are ‘help’, ‘clear’, ‘check’, ‘show’, and ‘exit’. When the user selects ‘help’ the list of allowed commands prints to the screen. For ‘clear’, the 3D grid array is set to 0 at all indices, which subsequently clears the game board. For ‘check’, the current state of the gameboard is read and if there are any discrepancies/errors between it and the solution space, the number of errors is given to the user. If everything is correct, the program displays a 'win' message to the user, and the program closes. For ‘show’, the user is asked to confirm if they want to see the solution to the puzzle and if yes, the solutions are shown on the board, with the program ending afterward. Finally, ‘exit’ closes the program.

To prevent/mitigate race conditions, anytime a shared resource is planned to be modified by either thread, such as in the command line interface when the grid array is modified during the ‘clear’, ‘check’, and ‘show’ commands, a lock is applied on one of the threads. Once the operation is complete, the lock is removed and execution can continue as normal.

## Going Beyond the 110 Topics

This program dealt heavily with many of the topics we touched on in this course, such as branching statements, file input/output, graphics, string operations, and set operations, at a larger scope/scale. In addition, this program utilized concepts not explored in the course, such as constraint problems, threading, the JSON format, race conditions, and recursion.

## Conclusion

At the start of this project, I identified multiple challenges I’d have to overcome in order to be successful, such as finding a way to procedurally generate entire puzzles, with sets of written clues, for the user to solve, varying in difficulty, and being able to modify, display, check, and store the gameboard in an efficient way. I believe that I was able to accomplish both of these challenges over the past few months, and deliver a fully-functional deliverable that met the original specifications outlined at the start of this project.

Karson's Logic Puzzle Player

| | Brody | Candice | Keaton | Talia | Arabic | English | French | Spanish |
|---|---|---|---|---|---|---|---|---|
| Amber | | | | | | | | |
| Blue | | | | | | | | |
| Gray | | | | | | | | |
| Green | | | | | | | | |
| Arabic | | | | | | | | |
| English | | | | | | | | |
| French | | | | | | | | |
| Spanish | | | | | | | | |

C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe

```
Generating clues...
Consolidating clues...
Reducing clues...

Here are your clues. Have fun!
 - [1] The person with Gray eyes is neither Talia nor Keaton.
 - [2] The person with Green eyes is Brody.
 - [3] The person with Arabic as their language is either Talia or Brody.
 - [4] The person with Gray eyes has French as their language.
 - [5] The person with Spanish as their language has Green eyes.
 - [6] The person with Amber eyes is Keaton.
 - [7] The eye color of Brody is Green.

Type 'help' for a list of commands.
>
```

```python
from graphics import graphics
from constraint import *
import threading
import random
import json

threadLock = [1, 1]

def generate_clues(sub, adj1, adj2):
    # Append every possible clue combination for given categories to JSON array
    data = {}
    data['clues'] = []
    adj = [adj1, adj2]

    # Append basic clue type (is/is not)
    for i in range(0,4):
        for j in range(0,4):
            for k in range(0,2):
                # x is y
                data['clues'].append({
                    'text': "The " + adj[k]["keyword"] + " of " + sub["list"][i] + " is " + adj[k]["list"][j] + ".",
                    'logic': [0, i, j, k]
                })
                # x is not y
                data['clues'].append({
                    'text': "The " + adj[k]["keyword"] + " of " + sub["list"][i] + " is not " + adj[k]["list"][j] + ".",
                    'logic': [1, i, j, k]
                })
                # y is x
                data['clues'].append({
                    'text': "The " + sub["type"] + " with " + adj[k]["list"][j] + " " + adj[k]["type"] + " is " + sub["list"][i] + ".",
                    'logic': [0, i, j, k]
                })
                # y is not x
                data['clues'].append({
                    'text': "The " + sub["type"] + " with " + adj[k]["list"][j] + " " + adj[k]["type"] + " is not " + sub["list"][i] + ".",
                    'logic': [1, i, j, k]
                })

    # Append neither/either clue types
    pairs = [[0,1],[0,2],[0,3],[1,2],[1,3],[2,3]]
    for pair in pairs:
        for j in range(0,4):
            for k in range(0,2):
                # x is either a or b
                data['clues'].append({
                    'text': "The " + sub["type"] + " with " + adj[k]["list"][j] + " " + adj[k]["type"] + " is either " + sub["list"][pair[0]] + " or " + sub[
                    "list"][pair[1]] + ".",
                    'logic': [2, pair, j, k]
                })
                data['clues'].append({
                    'text': "The " + sub["type"] + " with " + adj[k]["list"][j] + " " + adj[k]["type"] + " is either " + sub["list"][pair[1]] + " or " + sub[
                    "list"][pair[0]] + ".",
                    'logic': [2, pair, j, k]
                })
                # x is neither a nor b
                data['clues'].append({
                    'text': "The " + sub["type"] + " with " + adj[k]["list"][j] + " " + adj[k]["type"] + " is neither " + sub["list"][pair[0]] + " nor " +
                    sub["list"][pair[1]] + ".",
                    'logic': [3, pair, j, k]
                })
                data['clues'].append({
                    'text': "The " + sub["type"] + " with " + adj[k]["list"][j] + " " + adj[k]["type"] + " is neither " + sub["list"][pair[1]] + " nor " +
                    sub["list"][pair[0]] + ".",
                    'logic': [3, pair, j, k]
                })

    return data

def generate_categories():
    # List of possible adjectives and their keywords/types
    adj_keywords = ["age", "city", "eye color", "hair color", "height",
                    "language", "shirt size", "state"]
    adj_type = ["years of age", "as their city", "eyes", "hair",
                "as their height", "as their language",
                "as their shirt size", "as their state"]
```

```
    # Select two adjectives from list randomly
    adj1_rand, adj2_rand = random.sample(range(0,len(adj_keywords)), 2)

    # Read file corresponding to adjective one
    adj1_file = open("categories/" + adj_keywords[adj1_rand] + ".txt", 'r')
    adj1_lines = adj1_file.readlines()
    adj1_file.close()

    # Read file corresponding to adjective 2
    adj2_file = open("categories/" + adj_keywords[adj2_rand] + ".txt", 'r')
    adj2_lines = adj2_file.readlines()
    adj2_file.close()

    # Read file corresponding to subject (person)
    subj_file = open("subjects/" + "name.txt", 'r')
    subj_lines = subj_file.readlines()
    subj_file.close()

    # Randomly pick adjective and subject names
    adj1_list = []
    for i in random.sample(range(0,len(adj1_lines)), 4):
        adj1_list.append(adj1_lines[i].strip('\n'))
    adj2_list = []
    for i in random.sample(range(0,len(adj2_lines)), 4):
        adj2_list.append(adj2_lines[i].strip('\n'))
    subj_list = []
    for i in random.sample(range(0,len(subj_lines)), 4):
        subj_list.append(subj_lines[i].strip('\n'))

    # Create dictionary files of our random categories
    sub = {
        "keyword" : "name",
        "list" : sorted(subj_list),
        "type" : "person"
        }
    adj1 = {
        "keyword" : adj_keywords[adj1_rand],
        "list" : sorted(adj1_list),
        "type" : adj_type[adj1_rand]
        }
    adj2 = {
        "keyword" : adj_keywords[adj2_rand],
        "list" : sorted(adj2_list),
        "type" : adj_type[adj2_rand]
        }
    return sub, adj1, adj2

def pick_clues(data, sub, adj1, adj2):
    # Create two problem solver objects
    solver1 = Problem()
    solver2 = Problem()

    # Create two lists to store solutions
    sol1 = []
    sol2 = []

    # Create two lists to store clues
    clues = []
    clues_old = []

    # Initialize solvers
    reset_solver([solver1, solver2])

    # Generate initial list of clues that creates one unique solution
    print("Generating clues...")
    while (len(sol1) != 1) or (len(sol2) != 1):
        # Store backup of clues before adding new one
        clues_old = list(clues)

        # Get random generated clue
        clue = get_random_clue(data)

        # Add clue logic as constraint to solver
        add_constraint(clue['logic'], solver1, solver2)

        # Append clue to clue list
        clues.append([clue['text'], clue['logic']])
```

```
        # Get solution space of solver after new clue added
        sol1 = solver1.getSolutions()
        sol2 = solver2.getSolutions()

        # If solution space is 0 (inconsistent)
        if len(sol1) < 1 or len(sol2) < 1:
            # Clear solver and clues
            reset_solver([solver1, solver2])
            clues.clear()
            # Reset clues/solver to before last clue was added
            for i in range(len(clues_old)):
                add_constraint(clues_old[i][1], solver1, solver2)
                clues.append(clues_old[i])

    # Add certain compound clues to increase puzzle difficulty
    print("Consolidating clues...")
    clues = consolidate_clues(clues, sol1, sol2, sub, adj1, adj2)

    # Procedurally remove clues that have no effect on solution space (redundant)
    print("Reducing clues...")
    for a in range(len(clues)):
        for i in range(len(clues)):
            # Reset solver
            reset_solver([solver1, solver2])
            for j in range(len(clues)):
                # If clue not identical, add to solver
                if clues[i][1] != clues[j][1]:
                    add_constraint(clues[j][1], solver1, solver2)
            # If adding clue had no effect on solution space, remove clue
            if (len(solver1.getSolutions()) == 1) and (len(solver2.getSolutions()) == 1):
                del clues[i]
                break

    return clues, sol1, sol2

def consolidate_clues(clues, sol1, sol2, sub, adj1, adj2):
    adj = [adj1, adj2]

    # Iterate through every item in the solution
    for key in ['A','B','C','D']:
        # Convert the solution item to subj, adj1, adj2
        i = ord(key) - ord('A')
        j = sol1[0][key]
        k = sol2[0][key]

        # Small chance that a compound clue will be inserted that matches solution
        if random.randint(0,2) == 0:
            logic = [4, i, j, k]
            # Two variations of clue text to choose from randomly
            if random.randint(0,1):
                text = "The " + sub["type"] + " with " + adj[0]["list"][j] + " " + adj[0]["type"] + " has " + adj[1]["list"][k] + " " + adj[1]["type"] +
                 "."
            else:
                text = "The " + sub["type"] + " with " + adj[1]["list"][k] + " " + adj[1]["type"] + " has " + adj[0]["list"][j] + " " + adj[0]["type"] +
                 "."
            # Append new clue to list
            clues.append([text, logic])

    return clues

def reset_solver(solvers):
    # Reset all solvers to default state
    for solver in solvers:
        solver.reset()
        solver.addVariable("A", [0, 1, 2, 3])
        solver.addVariable("B", [0, 1, 2, 3])
        solver.addVariable("C", [0, 1, 2, 3])
        solver.addVariable("D", [0, 1, 2, 3])
        solver.addConstraint(AllDifferentConstraint())

def get_random_clue(data):
    # Get random clue from JSON list
    clue = data['clues'][random.randint(0,len(data['clues'])-1)]
    return clue

def add_constraint(logic, solve_adj1, solve_adj2):
```

```
    # Constraint logic for x == y
    if logic[0] == 0:
        for i in range(0,4):
            if i != logic[1]:
                add_constraint([1, i, logic[2], logic[3]], solve_adj1, solve_adj2)

    # Constraint logic x != y
    if logic[0] == 1:
        subject = chr(ord('A') + int(logic[1]))
        value = logic[2]
        adjective = logic[3] + 1
        for i in ['A', 'B', 'C', 'D']:
            if adjective == 1 and subject[0] == i:
                solve_adj1.addConstraint((lambda a: a != value), [subject[0]])
            elif adjective == 2 and subject[0] == i:
                solve_adj2.addConstraint((lambda a: a != value), [subject[0]])

    # Constraint logic a || b == y
    elif logic[0] == 2:
        for i in range(0,4):
            if i not in {logic[1][0], logic[1][1]}:
                add_constraint([1, i, logic[2], logic[3]], solve_adj1, solve_adj2)

    # Constraint logic a || b != y
    elif logic[0] == 3:
        add_constraint([1,logic[1][0], logic[2], logic[3]], solve_adj1, solve_adj2)
        add_constraint([1,logic[1][1], logic[2], logic[3]], solve_adj1, solve_adj2)

    # Compound constraint logic, type 1
    elif logic[0] == 4:
        add_constraint([0, logic[1], logic[3], 1], solve_adj1, solve_adj2)

def print_clues(clues):
    # Convert list of clues to set, to remove duplicates
    set_clues = set()
    for clue in clues:
        set_clues.add(clue[0])

    # Print clues in ordered list
    i = 0
    for clue in set_clues:
        print(" - [" + str(i + 1) + "] " + clue)
        i = i + 1

def gui(grid, sub, adj1, adj2):
    global threadLock

    # Window size
    xsize = 600
    ysize = 600

    # Window object created
    window = graphics(xsize, ysize, "Karson's Logic Puzzle Player")

    # Window mouse actions set
    window.set_left_click_action(grid_place, [grid, xsize, ysize, "l"])
    window.set_right_click_action(grid_place, [grid, xsize, ysize, "r"])

    # Graphics loop
    while threadLock[0] != 0:
        window.clear()
        draw_gridlines(window, xsize, ysize)
        draw_titles(window, sub, adj1, adj2, xsize, ysize)
        draw_marks(window, xsize, ysize, grid)
        window.update_frame(60)
        # Pause execution if locked
        while threadLock[0] == -1:
            threadLock[1] = 1

def instantiate_grid(grid):
    # Clear grid
    grid.clear()

    # Create 3D array of correct size, initialized to 0
    for i in range(0,3):
        grid.append([])
        for j in range(0,4):
```

```
        grid[i].append([])
        for k in range(0,4):
            grid[i][j].append([])
            grid[i][j][k] = 0

def draw_gridlines(window, xsize, ysize):
    # Calculate size of grid from window size
    gridx = xsize / 9
    gridy = ysize / 9

    # Draw all horizontal/vertical lines in grid
    for i in range(1,10):
        window.line(i * gridx, 5 * gridy, i * gridx, 0, 'black', 1)
    for i in range(1,6):
        window.line(i * gridx, 5 * gridy, i * gridx, ysize, 'black', 1)
        window.line(0, i * gridy, xsize, i * gridy, 'black', 1)
    for i in range(6,10):
        window.line(0, i * gridy, 5 * gridx, i * gridy, 'black', 1)

    # Draw bolded lines between titles and play area
    window.line(0, gridy, xsize, gridy, 'black', 2)
    window.line(0, 5 * gridy, xsize, 5 * gridy, 'black', 2)
    window.line(gridx, ysize, gridx, 0, 'black', 2)
    window.line(5 * gridx, ysize, 5 * gridx, 0, 'black', 2)

def draw_titles(window, sub, adj1, adj2, xsize, ysize):
    # Calculate grid size from window size
    gridx = xsize / 9
    gridy = ysize / 9

    # Draw category titles onto the screen
    for i in range(0,4):
        fontsize = 10
        drawx = gridx * (i+1.05)
        drawy = 0.05 * gridy
        window.text(drawx, drawy, sub['list'][i][0:9], 'black', fontsize)
        window.text(drawx, drawy + (0.2 * gridy), sub['list'][i][9::], 'black', fontsize)
        drawx = 0.05 * gridx
        drawy = gridy * (i+1.05)
        window.text(drawx, drawy, adj1['list'][i][0:9], 'black', fontsize)
        window.text(drawx, drawy + (0.2 * gridy), adj1['list'][i][9::], 'black', fontsize)
        drawx = gridx * (i+5.05)
        drawy = 0.05 * gridy
        window.text(drawx, drawy, adj2['list'][i][0:9], 'black', fontsize)
        window.text(drawx, drawy + (0.2 * gridy), adj2['list'][i][9::], 'black', fontsize)
        drawx = 0.05 * gridx
        drawy = gridy * (i+5.05)
        window.text(drawx, drawy, adj2['list'][i][0:9], 'black', fontsize)
        window.text(drawx, drawy + (0.2 * gridy), adj2['list'][i][9::], 'black', fontsize)

def grid_place(window, xpos, ypos, args):
    # Calculate size of grid from input arguments
    gridx = int(xpos / (args[1] / 9))
    gridy = int(ypos / (args[2] / 9))

    # Calculate which subgrid mouse click is on, if any
    subgrid = -1
    if gridx in range(1,5) and gridy in range(1,5):
        subgrid = 0
    elif gridx in range(5,9) and gridy in range(1,5):
        subgrid = 1
    elif gridx in range(1,5) and gridy in range(5,9):
        subgrid = 2

    # Convert click location to subgrid array indexes
    xindex = (gridx - 1) % 4
    yindex = (gridy - 1) % 4

    # Set index in grid array to appropiate value, based on l/r click
    if subgrid != -1:
        if args[3] == "l":
            if args[0][subgrid][xindex][yindex] == 0:
                args[0][subgrid][xindex][yindex] = 1
            else:
                args[0][subgrid][xindex][yindex] = 0
        else:
            for i in range(0,4):
```

```
            args[0][subgrid][i][yindex] = 1
            args[0][subgrid][xindex][i] = 1
        args[0][subgrid][xindex][yindex] = 2

def draw_marks(window, xsize, ysize, grid):
    # Set size of grids based on window size
    gridx = xsize / 9
    gridy = ysize / 9

    # Iterate through all tiles of grid and place appropiate marking
    for i in range(0,3):
        for j in range(0,4):
            for k in range(0,4):
                # Calculate initial drawing position for each tile
                drawx = (j + 0.5) * gridx
                drawy = (k + 0.5) * gridy
                if i == 0:
                    drawx += gridx
                    drawy += gridy
                elif i == 1:
                    drawx += 5 * gridx
                    drawy += gridy
                else:
                    drawx += gridx
                    drawy += 5 * gridy
                # Draw an X at the tile
                if grid[i][j][k] == 1:
                    window.line(drawx - 0.45 * gridx, drawy - 0.45 * gridy, drawx + 0.45 * gridx, drawy + 0.45 * gridy, 'black')
                    window.line(drawx - 0.45 * gridx, drawy + 0.45 * gridy, drawx + 0.45 * gridx, drawy - 0.45 * gridy, 'black')
                # Draw an O at the tile
                if grid[i][j][k] == 2:
                    window.ellipse(drawx, drawy, 0.9 * gridx, 0.9 * gridy, 'black')
                    window.ellipse(drawx, drawy, 0.9 * gridx - 4, 0.9 * gridy - 4, 'white')

def cli(grid, sol1, sol2):
    print("\nType 'help' for a list of commands.")

    # Command line interface loop, only breaks when done == TRUE
    done = False
    while not done:
        # Get sanitized command input from user
        command = input("> ")
        command = sanitize(command)

        # Print list of commands
        if "help" in command:
            print_help()
        # Clear board of all markings
        elif "clear" in command:
            lockGUI()
            instantiate_grid(grid)
            unlockGUI()
            print("Cleared board.")
        # Check how correct the current board is
        elif "check" in command:
            lockGUI()
            if check_solution(grid, sol1, sol2):
                unlockGUI()
                done = True
                input("\nPress enter when you are ready to exit...\n")
                closeGUI()
            else:
                unlockGUI()
        # Show solution
        elif "show" in command:
            sure = False
            # Ask if user is sure, as this will end the game
            while not sure:
                print("Are you sure? This will end the game. (y/n)", end='')
                command = input(" ")
                command = sanitize(command)
                # Show solution
                if 'y' == command:
                    sure = True
                    done = True
                    lockGUI()
                    show_solution(grid, sol1, sol2)
```

```
                unlockGUI()
                input("\nPress enter when you are ready to exit...\n")
                closeGUI()
            # Continue as normal
            elif 'n' == command:
                sure = True
        # Exit program
        elif "exit" in command:
            print("Goodbye!")
            done = True
            closeGUI()
        # Command not found
        else:
            print("Invalid command. Type 'help' for a list of available commands.")

def lockGUI():
    global threadLock

    # Send signal to stop execution of GUI
    threadLock[0] = -1
    threadLock[1] = 0

    # Wait until GUI has stopped
    while threadLock[1] != 1:
        threadLock[0] = -1

def unlockGUI():
    global threadLock

    # Send signal to start execution of GUI
    threadLock[0] = 1

def closeGUI():
    global threadLock

    # Send signal to close GUI
    threadLock[0] = 0

def print_help():
    # Print help command
    print("Available commands are:")
    print("\t'help' - list available commands")
    print("\t'clear' - clear board")
    print("\t'check' - check solution")
    print("\t'show' - show solution")
    print("\t'exit' - exit program")

def check_solution(grid, sol1, sol2):
    # Variable to keep track of differences between solution and board
    num_error = 0

    # If board does not match given solution, increment check_solution
    for i in sol1[0]:
        j = ord(i) - ord('A')
        if not grid[0][j][sol1[0][i]] == 2:
            num_error += 1
    for i in sol2[0]:
        j = ord(i) - ord('A')
        if not grid[2][j][sol2[0][i]] == 2:
            num_error += 1
    for i in range(0,4):
        j = chr(ord('A') + i)
        if not grid[1][sol2[0][j]][sol1[0][j]] == 2:
            num_error += 1

    # Print result to user
    if num_error > 0:
        print("You have " + str(num_error) + " errors.")
        return False
    else:
        print("Congratulations! You beat the puzzle.")
        return True

def show_solution(grid, sol1, sol2):
    # Set all indexes to 1 (X)
    for i in range(0,3):
        for j in range(0,4):
```

```
            for k in range(0,4):
                grid[i][j][k] = 1

    # Set indexes of solutions to 2 (O)
    for i in sol1[0]:
        j = ord(i) - ord('A')
        grid[0][j][sol1[0][i]] = 2
    for i in sol2[0]:
        j = ord(i) - ord('A')
        grid[2][j][sol2[0][i]] = 2
    for i in range(0,4):
        j = chr(ord('A') + i)
        grid[1][sol2[0][j]][sol1[0][j]] = 2

def sanitize(string):
    # Set maximum string length
    if len(string) > 6:
        string = string[0:6]

    # Only allow ASCII characters in string
    string = string.encode("ascii", errors="ignore").decode()

    # Remove invalid characters
    invalid_chars = {'<','>',':','"','/','\\','|','?','*','_','\n'}
    for char in invalid_chars:
        string = string.replace(char, '')

    # Set string to lowercase letters
    string = string.lower()

    return string

def main():
    # Generate the randomized categories for this round
    sub, adj1, adj2 = generate_categories()

    # Generate all possible basic clues for the given categories
    all_clues = generate_clues(sub, adj1, adj2)

    # Pick assortment of clues to make a solvable puzzle
    puzzle_clues, sol1, sol2 = pick_clues(all_clues, sub, adj1, adj2)

    # Show clues to player
    print("\nHere are your clues. Have fun!")
    print_clues(puzzle_clues)

    # Create gameboard grid
    grid = []
    instantiate_grid(grid)

    # Start threads for the CLI and GUI
    t = threading.Thread(target=gui, args=(grid, sub, adj1, adj2))
    u = threading.Thread(target=cli, args=(grid, sol1, sol2))
    t.start()
    u.start()

main()
```