**Show name: Hue Show**

Short Description: A visually stunning performance that demonstrates the capabilities of the lights. This program runs four sub-programs that ripple colors across the lights with varying degrees of order.

Long Description: This program has four parts that are customizable to run at different speeds. Part 1: simultaneous\_hue\_shift. All LEDs light at the same time to the same color. Color iterates through RGB spectrum. Part 2: serialized\_hue\_shift. Rather than all LEDs displaying the same color, each successive LED in the chain will display the next color in the spectrum. Each LED iterates through, making a rippling effect. Part 3: non\_serialized\_hue\_shift. The LEDs change in successive order, so the colors still appear to move along the strand, but each successive color is random. Like the lights on the trim of a movie theater. Part 4: random\_light\_display. Each color and each placement of the color is random. Precursor to rave.py.

Options:

* Wait1, wait2, wait3, wait4 – Float between 0 and 5. Default: 0. Each variable corresponds to the sub-function of the same number. This is the time in seconds between each change of color. In reality, 0 is not actually 0 seconds.
* Speed1, speed2, speed3, speed4 – Float between 1 and 256. Default: 1.75. Determines how gradual the change is between each color. 1 means you see every color, 3 means you see every third color, etc.

**Show name: Snake**

Short Description: This program plays the classic game of snake, but adds a few elements. The snake moves itself automatically around a grid. It actively looks for nom\_noms which make it longer. It can hit hidden landmines that decrease its length. Caffeine pills will make it move rapidly.

Long Description: This program plays the classic game of snake, but adds a few elements. The snake moves itself automatically around a grid. It looks for nom\_noms which make it longer. There is a weighting system implemented to help the snake decide where to move based on the proximity of nom\_noms. Though it can see nom\_noms and navigate towards them, it cannot see bombs, which make it shorter. When a nom\_nom or bomb is hit, the appropriate effect is applied and the item respawns somewhere else on the board. There are also caffeine pills which make the snake move faster, though this is only for visual effect. When a bomb, nom\_nom, or caffeine pill is hit, the non-programmable lights will flash. If the snake gets caught in a loop, it will execute a random move to escape the loop. In text output, each node of the snake is numbered, with 1 being the head. 44 represents a bomb, 55 a caffeine pill, and 99 a nom\_nom.

Options:

* Sleep\_tme – Float between 0 and 2. Default: 0.5. The amount of time between each movement of the snake. Lower number equals faster snake.
* Wait\_times – Vector of any length with float elements less than sleep\_time. Default: [0.05, 0.05, 0.05, 0.05, 0.05, 0.1, 0.15, 0.2, 0.2] The sleep times for the moves following the snake hitting a caffeine pill. The default make the moves very fast for five moves then slows down over the next four.
* Number\_of\_nom\_noms – Integer between 1 and 10. Default: 2. The number of nom\_noms on the board at any given time. In practice, lower numbers are better to avoid quick games.
* Number\_of\_caffeine\_pills – Integer between 0 and 10. Default: 5. The number of caffeine pills on the board at any given time.
* Level\_0\_nom\_nom – Integer between 3 and 20. Default: 10. The weighting placed on a level\_o\_nom\_nom, which is classified as a nom\_nom that is adjacent to the head of the snake. Higher values make the snake more likely to turn towards the nom\_nom.
* Level\_1\_nom\_nom – Integer between 2 and 19. Must be less than level\_o\_nom\_nom. Default: 7. Same as above but applies to nom\_noms that are exactly 2 moves away from the head.
* Level\_2\_nom\_nom – Integer between 1 and 18. Must be less than level\_1\_nom\_nom. Default: 4. Same as above but applies to nom\_noms that are exactly 3 moves away from the head.
* Display\_on\_lights\_boolean – Boolean. Determines whether the program is displayed on the lights. If false, only text output is displayed on the screen.

**Show Name: Reversi**

Short Description: This program plays the game of reverse. The two players can be controlled by humans or by the computer, in which case several strategies can be implemented. A basic level of machine learning can be employed to train the computer players. Then, any number of games can be played.

Long Description: Reversi is a game in which two players lay tiles on an 8x8 grid and attempt to flip their opponents tiles by sandwiching them between their own. A piece can only be laid if it will capture (flip) at least one opponent’s piece. If a human is playing, he or she can enter moves directly from the terminal. If the computer is selected as a player, there are several strategies that it can follow, based on raw score increase or on the success of that move in past games. This latter set of strategies can only be implemented if the computer is allowed to train itself by playing many games. It will generate a weighting for each move on the grid, allowing it to later predict the success of each move. This program comes with a lot of text output that can be turned on or off. There are essentially two sets of options: one for when the computer trains and one for when it doesn’t. In each case, the strategy for each player can be set as well as if the text output should be printed and how many games should be played. Each move is printed along with a description of the move and the current score. See REVERSI\_READ\_ME in code repo for an in-depth description of the many sub-functions used in this show. There is also functionality to change the color of each player. This is done by inputting an RGB color code for the black player and for the white player.

Options:

* Strategies
  + First\_max\_score - The computer will evaluate all legal moves for the number of pieces that they will capture, then find the most valuable move. If there is a tie, it will choose the first move it evaluated.
  + Max\_score\_priority\_closest\_to\_cent - Save as first\_max\_score, but ties are resolved by evaluating proximity to center of the board.
  + Max\_score\_priority\_closest\_to\_edge - Same as above, but ties are resolved by evaluating proximity to edge of board.
  + Weight\_grid\_strategy - An 8x8 grid (weight\_grid) of zeros is generated with each value representing a matching square on the board. The computer will train for the number of games specified in train\_iterations. When black wins, any square on the board that has a black piece on it gets incremented by 1 on the weight\_grid. Any square with a white piece on it gets decremented by 1. In the games following training, moves are determined by finding the legal move that is rated highest on the weight\_grid. If there is a tie, the computer chooses the first move that it evaluated. Note that in training, the weight\_grid is based on Black’s moves, but either color can use the data.
  + Weight\_grid\_strategy\_edge\_priority - Same as weight\_grid\_strategy, but ties are resolved by evaluating proximity to edge of the board.
* Options for black\_input and white\_input
  + Starter\_input - Program asks the user for one of the options below (entered in terminal).
  + Human - The user plays by entering coordinates in the terminal.
  + Computer\_weight\_capture - Uses the weight\_capture\_strategy\_edge\_priority 20% of the time, and a random strategy from the list 80% of the time.
  + Computer\_random - Picks a random strategy.
* Train - Boolean. Default: Yes. Determines if the computer trains. Necessary for successful implementation of weight\_grid strategies.
* Train\_iterations - Integer between 1 and 1,000,000. Default: 1,000. Number of games to train for. 1,000 takes about 2 minutes.
* Number\_of\_games – Integer between 1 ad 1,000,000. Default: 1. Number of games to play (after training, if training is used).
* Print\_bool\_no – Boolean. Default: 1. If train=no, determines whether to print text output of games. If train=yes, determines whether to print text output of games after training.
* Black\_input\_no – See options block above. Default: starter\_input. Used when train=no.
* White\_input\_no – See options block above. Default: starter\_input. Used when train=no.
* Print\_bool\_yes – Boolean. Default: 0. If train=yes, determines whether to print text output of the games used for training. For games after training, see print\_bool\_no.
* Black\_input\_yes – See options block above. Default: computer\_weight\_capture. Used when train=yes.
* White\_input\_yes – See options block above. Default: computer\_random. Used when train=yes.
* Reset\_weight\_grid\_boolean – Boolean. Default: 1. Determines whether to reset the weight grid to 0’s at the start of training, or use the grid from last time. The grid is saved after every run.
* Display\_on\_lights\_boolean – Boolean. Default: 0. Determines whether to display the program on the lights.
* Sleep\_time – Float between 0 and 10. Default: 0.5. Time in seconds between each move.
* R\_value\_black – Integer between 0 and 255. Default: 255. R value in black’s RGB color code.
* G\_value\_black – Integer between 0 and 255. Default: 0. G value in black’s RGB color code.
* B\_value\_black – Integer between 0 and 255. Default: 0. B value in black’s RGB color code.
* R\_value\_white – Integer between 0 and 255. Default: 0. R value in white’s RGB color code.
* G\_value\_white – Integer between 0 and 255. Default: 255. G value in white’s RGB color code.
* B\_value\_white – Integer between 0 and 255. Default: 0. B value in white’s RGB color code.

**Show Name: Tic\_tac\_toe**

Short Description: In this program, the computer will play tic tac toe against itself, war games style. There are two phases: a training stage and a playing stage. The computer will train against itself for thousands of games without any output, then it will play games on the lights board. It uses it’s training, so it is expected that most, or all, games are draws. In both phases, the players check for immediate wins, then check for ways to avoid immediate losses. This way, they act more like human players.

Long Description: In the training stage, both players execute random moves. Each of 9 possible positions (3x3 grid) in 5 move numbers (e.g. move #2), gets a weighting. When a particular position in a particular move is played, and the player eventually wins the game, that value is incremented. If the move is played and the game is eventually lost, the move is decremented. The computer will train for thousands of games, with both players using their own weightings. In the playing stage, they simply base their moves on the weightings rather than playing randomly. Due to the weightings, most games in the playing stage should be exactly the same. Thus, it may be more entertaining to turn on the output during the training stage to just see a bunch of random games played out. Also, before you ask us, yes we know there are established flow charts for winning/drawing moves that we could have simply programmed in. But what fun is that?

Options:

* Display\_on\_lights\_boolean – Boolean. Default: 1. Indicates whether the games in the playing stage should be displayed on the lights.
* Display\_on\_lights\_train\_boolean – Boolean. Default: 0. Indicates whether the games in the training stage should be displayed on the lights.
* Summary\_print\_bool – Boolean. Default: True. Indicates whether a summary of the entire game session will be printed to the screen after all games have been played.
* Input\_number\_of\_trains – Integer between 0 and 1,000,000. Default: 10,000. The number of games used to train the players.
* Input\_number\_of\_games – Integer between 1 and 1,000,000. Default: 10. The number of games played after training.
* X\_win\_weighting – Float. Default: 1. How much to increment or decrement the weighting values when X wins or loses. Theoretically, if this is set negative, X will try to lose, though it will still avoid immediately losses and take immediate wins. This has not been tested.
* O\_win\_weighting – Float. Default: 1. Same as x\_win\_weighting, but for player O.
* Sleep\_time – Float between 0 and 10. Default: 0.5. The time to wait between each move in playing stage. Faster makes it more like War Games.
* Sleep\_time\_train – Float between 0 and 10. Default: 0. The time to wait between each move in training stage. Usually 0 because number of training games is usually high.

**Show Name: Clock**

Short Description: Display a clock in either vertical or horizontal orientation. The horizontal orientation offers a programmable timer.

Long Description: The vertical clock is limited by space, so it does not offer a timer. It displays the hour, minutes, and seconds. The horizontal clock can display a timer, which can be set to any value. It will do its best to adjust the timer when the units change (e.g. 1 minute goes to 59 seconds). When the timer finishes, lights flash.

Options:

* Orientation – ‘horizontal’ or ‘vertical’. Default: ‘horizontal’. Indicates the orientation of the board. Vertical is 16x10, horizontal is 10x16.
* Display\_on\_lights\_bool – Boolean. Default: 1. Indicates whether clock is displayed on lights.
* Timer\_value – Float between 0 and 99. Default: 1. The magnitude of the timer. Only available in horizontal orientation.
* Timer\_unit – ‘seconds’, ‘minutes’, or ‘hours’. Default: ‘minutes’. The unit of the timer. Only available in horizontal orientation.

**Show Name: Text\_Scroller**

Short Description: Scrolls text across the lights grid in horizontal orientation. Has two rows of text, so two messages can be displayed at once. The text can be monochromatic or every letter can be a different color. Meant to be used as a member function of other programs. Text can be looped for a specified amount of time.

Long Description: The user passes a string into the program. It parses the string and assigns a grid display to each character. Then, some cool math is used to make the text scroll. There is a loop function that will make the text strings loop for a certain number of iterations. If the input for this feature is set to -1, the string will loop forever. This function could be useful in other programs that need to output text.

Options:

* Input\_string\_top – String. Default: “Go Blue!”. Any string that will be displayed on the upper lane of the scroller. Should accept any character, including special characters.
* Input\_string\_bottom – String. Default: “Go Blue!”. Any string that will be displayed on the lower lane of the scroller. Should accept any character, including special characters.
* Rainbow\_bool – Boolean. Default: False. If set to true, each character in the string will be assigned a different color, creating rainbow text.
* Lights\_bool – Boolean. Default: True. Indicates if text is displayed on the lights.
* Text\_color\_top - “Blue”, “Red”, “Yellow”, “Green”, or “Purple”. Default: “Blue”. Color of the string on the top lane.
* Text\_color\_bottom - “Blue”, “Red”, “Yellow”, “Green”, or “Purple”. Default: “Blue”. Color of the string on the bottom lane.
* Loop\_count – Integer between 1 and 100, or -1. The number of times to loop the text. If -1, loop text forever.

**Show Name: Big\_Text\_Scroller**

Short Description: Same as regular Text\_Scroller, but only one lane, so the text is bigger.

Options:

* Input\_string – String. Default: “Go Blue!”. Any string that will be displayed on the scroller. Should accept any character, including special characters.
* Rainbow\_bool – Boolean. Default: False. If set to true, each character in the string will be assigned a different color, creating rainbow text.
* Lights\_bool – Boolean. Default: True. Indicates if text is displayed on the lights.
* Text\_color - “Blue”, “Red”, “Yellow”, “Green”, or “Purple”. Default: “Blue”. Color of the text.
* Loop\_count – Integer between 1 and 100, or -1. The number of times to loop the text. If -1, loop text forever.