Gearball simulator

Usage

invoke the gear ball randomizer with

```
python run.py [number]
```

where [number] is the number of moves you want the randomizer to take. Each moven taken will print the steps taken (row moved, direction moved in, row held) and the state of the gear ball after the move.

If you wish to make a single move on the gearball invoke `run.py` via

```
python run.py [row to move] [direction to move in] [row to hold] [q to quit]
```

For example, to execute a left twist on the top row while moving the bottom row in the opposite direction (the center row does not move), you would use

```
python run.py top left center q
```

If you want to continue to make moves on the gear ball after your initial move do not append 'q' to the end of your arguments. Continue to move the gear ball via the same command structure, `[row] [direction] [hold]`. Type `quit` or `q` to exit the gearball program.

The state of the gearball will be output to the screen after each move.

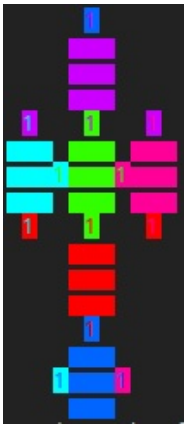
GUI

The gearball output is colored with 6 colors, one for each face of the gearball;

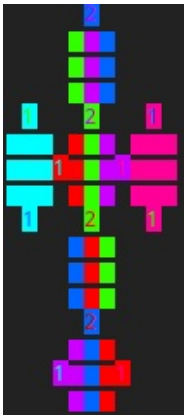
```
purple
cyan
green
magenta
red
blue
```

The central face (the one at the center of the "cross" formed by unfolding the gearball) is the face on which operations are executed upon, ie one row of the central face (left, right, top, or bottom) will always stay central.

A solved (or untouched) gearball will look like this:



and a gearball with one move executed (a left move up while holding center) will look like this:



Data Structure

A 'Gearball is an object defined in the `ball` class found in `ball.py`. A gearball has a collection of functions to print the state of the current gearball to the screen (textify*rows, output*ball, etc), and dicts of the other objects included within the ball. These are `rows` (the string version of the ball, only used to display the gearball state), `faces` (6, numbered, the six faces of a gearball) and `gears` (12, the gear pieces in between each face. Each faces and gears dict consists of keys corresponding to the location of the object on the gearball in string form, and the reference to that object as the value.

Faces and gears are python objects as well. Gears have an orientation and two colors, the two colors coming from the two faces the gear borders, and the orientation referencing which cog is pointing to the right. This orientation is numbered 1 to 6, with 1 being the 'solved' state of the gear (the state it begins in). Both colors are one of the possible 6 (purple, cyan, etc), given at gear creation. The string representation of the gear has the orientation colored as color 1 (one of the bordered faces) and the background as color 2 (the other bordered face). Gears never change color, they only change location on the ball (kept track of at the `ball` level object) and their orientation is updated as the adjacent rows slide past.

Faces have a dict of `piece` objects, another python object, and some functions used to output the state of the face. Color is handled at the `piece` level, faces only keep track of which pieces they own.

The `piece` object knows its own color and possesses a function to change its color to a new color. The location of each piece is kept track of at the `face` level, through the `pieces` dict.

As move operations are performed on the central face, rows in adjacent faces have their pieces change color accordingly. Pieces never move position only color, and always belong to their original face. Gears will change position and orientation as needed.

Randomizer

The `randomize()` function takes a number of moves as an argument, creates a gearball object, and scrambles it by generating sequential moves (up to the number passed to the function). This is accomplished by generating 3 random numbers, first to decide on a random row to move, then to decide a random direction, and finally a random row to hold. There are 16 "possible moves" (four rows, two directions, two holds, 4 * 2 * 2) but 8 of these moves are equivalent (`top left center` is equivalent to `bottom right center`) so 4 moves are removed from the possible moveset to avoid duplicates.

The `randomize()` function also will not produce an "undo" move, moves that either directly return to the n-2 state (a 'top, right, center' move will not be immediately preceded by a 'top, left, center' move) or moves that would constitute a 90 or 180 degree turn (a 'top, right, bottom' and 'bottom, right, center' would simply result in the gearball being turned 90 degrees to the right). This is accomplished by storing the previous states of the gearball just before each move is taken and persisting this into the next iteration of the loop that chooses moves. A move is chosen (via generating random numbers for th row, direction, and hold) and a temporary gear ball created from the state of the current gearball is moved according to the randomly chosen move. This temporary gearball's state (the string representation) is compared to the previous gearball's state and if they are equivalent, a new random move is generated. Simmilarly, before a move is executed the rotation states are saved (6 rotation states, 2 180 degree rotations on the x and y axis, 4 90 degree rotations towards 'up', 'left', 'right', 'down') and next iteration the possible move is compared against these possible rotations states, and a new random move is chosen if one is equal.

Heuristic

An 'edge' piece (the inside pieces that are neither corners nor gears) or 'corner' piece can be at most 8 steps away from its correct position, 6 moves to the opposite face of the gearball, two moves to flip to the other side of that face (a 'right' to a 'left', a 'top' to a 'bottom'). There are 4 edges and corners per face and 6 faces for total Manhattan Distance of 8 * 4 * 6 = 192. However, as any edge or corner moves it also moves the 4 edges and corners within its row around the other faces, and rotates the 4 edges and corners on the slice of gearball being moved, and moves the 8 edge pieces and corners on the center row or the 8 edge pieces on the reverse side of the gearball, so the true Manhattan Distance would be 192/16 = 12. The gears can be at most 6 moves away from solved, as they require 12 moves from solved to return back to solved. The 4 central gears move with every move but the gears on each edge being moved do not, so the true Manhattan Distance for gears is 12/4 = 3.

Statement

I had never heard of or considered A* algorithms prior to this assignment, although I did know and had used search trees and algorithms for traversing them. Learning A* and the considerations taken for how to construct reliable datastructures for the priority queue, how to pick a search algorithm, and how to find a heuristic gave me a more concrete understanding of the applications for the various algorithms and data structures I had learned previously. While in classes we would be tested on DFS and BFS I didn't really have a direct need to use them in a program.

Generally, I also learned better strategies for starting to model something relatively complex. Whenever I started to work on the gearball representation after trying to "think through it" or come up with what I needed in an abstract sense I would have trouble actually implementing it through code. Instead, I drew up some very simple initial data structures and began coding behavior through many repetetive 'if' statements, and then later after understanding the patterns I was writing out over and over I could go back and collapse many lines of individual logic into general utility functions.