**The working title of the project:**

**Force Up**

**Purpose of the project:**

Create a game application with GUI

**Platform:**

Any platform which can run Python 3.7

**Intended customer/user:**

Anyone who wants to play

**Source of the idea for the project:**

My imagination

**Development environment:**

Atom

**Tools which will be used:**

Python with 3.7 version

I will use standard Python modules:

* cmath
* tkinter
* random
* math

**Estimated numbers of lines of code:**

1400

**Documentation:**

**Force Up** is a two-player board game (or you can play with yourself if you want).

At the start, each player chooses his color and super powers one by one.

Players can’t have the same super powers.

Both players must have the same number of powers(at least 3 but no more than 10):

3-4 powers: each power will have 2 turns cooldown.

5-6 powers: each power will have 3 turns cooldown.

7-8 powers: each power will have 4 turns cooldown.

9-10 powers: each power will have 5 turns cooldown.

There is a list of powers along with their types and images:

C:\HugeLibrary\Projects\Job\OrdersInProcess\Force Up\development\resources\powers\rotate_clockwise.pngRotate

C:\HugeLibrary\Projects\Job\OrdersInProcess\Force Up\development\resources\powers\rotate_anticlockwise.png1 ) Rotate the board by 90 degrees clockwise.

C:\HugeLibrary\Projects\Job\OrdersInProcess\Force Up\development\resources\powers\rotate_180.png2 ) Rotate the board by 90 degrees counterclockwise.

3 ) Rotate the board by 180 degrees.

Pull up

4 ) Pull up all blocks on the selected row to the right.

5 ) Pull up all blocks on the selected row to the left.

Swap

6 ) Swap specified rows.

7 ) Swap specified columns. C:\HugeLibrary\Projects\Job\OrdersInProcess\Force Up\development\resources\powers\swap_cols.png

Intervene

8 ) Destroy last fallen block of the opponent and make him fall again randomly.

(if another player have used control power then this block will fall into the specified column)

9 ) Destroy last fallen block of the player and make him fall again randomly.

(if another player have used control power then this block will fall into the specified column)

Control

C:\HugeLibrary\Projects\Job\OrdersInProcess\Force Up\development\resources\powers\control_opponent.png10) Make the next block of the player to fall in the specified column.

11) Make the next block of the opponent to fall in the specified column.

Pull out (Notice that there is no matter who owns the affected block)

12) Choose a column where the block will be pulled out from the bottom of the column

(if this column contains a block in the bottom).

13) Choose a column where the block will be pulled out from the left edge of the row

(if this row contains a block in the left edge).

14) Choose a column where the block will be pulled out from the right edge of the row

C:\HugeLibrary\Projects\Job\OrdersInProcess\Force Up\development\resources\powers\pull_out_top.png(if this row contains a block in the right edge).

15) Choose a column where the block will be pulled out from the top of the column

(if this column contains a block in the top).

Shuffle

16) Shuffle all blocks in the specified row.

17) Shuffle all blocks in the specified column.

C:\HugeLibrary\Projects\Job\OrdersInProcess\Force Up\development\resources\powers\shuffle_cols.png18) Shuffle all rows of the board.

19) Shuffle all columns of the board.

20) Shuffle all blocks on the whole board.

Before a start, players may specify the size of the board (from 3x3 to 9x9).

At the beginning of each turn on one column will randomly (or not randomly if this fall will be affected by control power) fall one block with the color of the player.

Then the player may finish turn or use as much of his available powers as he wants.

If the player will achieve an appearance of the rectangle with the specified size on board filled up with blocks of his color after the board was affected by gravity then he will win.

The player will win if he will make a turn which will lead both players to the win.

But if on the current turn the board is filled up and there is no winner then this is a draw.

All blocks on the board are affected by gravity.

So they will fall further if they can and GUI have an animation for this.

***The architecture of the application****:*

Here will be frontend and backend parts of the application:

* Graphical User Interface (frontend part)
* Game Mechanics (backend part)

**Structure of the backend part:**

Here will be the main class ***Game*** which will be responsible for creating instances of the game.

This class will contain both computer logic and game mechanics.

The constructor of this class accepts the following arguments:

1. Rows
2. Cols
3. Height
4. Width
5. Turn

All passed arguments will become attributes of the ***Game*** class.

Besides them, the class will have the following attributes:

* mat (contraction from the matrix) attribute which will be a rows x cols array of zeros.

This is the board which can be changed with each move of any player.

* id attribute contains the id of the last fallen block.
* the pos1 attribute contains None or a number of the column where the next block of the first player will fall.
* the pos2 attribute contains None or a number of the column where the next block of the second player will fall.

Some methods of the ***Game*** class may return tips tuple or a dictionary or a number or None.

Tips contain the relations between the new and old position of the blocks if they have changed their position.

Tips will be empty or None if all blocks stayed on their positions.

The purpose of the tips is to send feedback to the GUI, so GUI will be able to understand which blocks should be animated and decide how they are should be animated.

Methods of the ***Game*** class:

* gravity (game)

This method iterates through all columns and set the pos variable to None.

Then it iterates through all rows from down to top and for each row checks value of each cell on the board.

If this cell is empty and pos is None then pos is set to the number of the current row. Else if this cell isn’t empty and pos isn’t None then the cell on this row swaps with the cell on the pos row and increase the absolute value of pos by one (going to next row).

Using this method simulate the gravity on the board.

This method also returns the tips dictionary.

This dictionary will have the following structure:

Id : path

Id represents id of the particular block on the board.

Path tells us how many rows the block was falling until he reached another block or the bottom of the board.

* fall (game)

The purpose of this method is to put one block (which represents game.id) on the top row of the board randomly or by considering whos turn now and that game.pos1 or game.pos2 is None or not.

Also, before putting the block on the board, the game.id is increasing by 1.

Besides, this method returns the value of the tips variable.

If there is no place for a block to fall then tips will be None otherwise it will contain a number of the column where the block has fallen.

* rotate (game, degrees)

The method rotates the board by specified degrees.

Degrees must be 90, 180 or – 90.

Notice that in pseudocode below(not only in this method) SWAP() swaps passed arguments, it’s not the swap (game, type, ind1, ind2) function of ***Game*** class object.

* pull\_up (game, row, side)

The purpose of this method is to pull up all blocks on the specified row to the specified side.

The side can be ‘right’ or ‘left.

Notice, that during transition blocks on the selected row will not fall down before calling gravity function even if during movement will be no blocks under them.

Also, blocks above them also won’t fall before calling gravity function even if under them will be no blocks.

Besides, this method creates a tips dictionary with the same structure as in the gravity method.

If the path will be positive then it means that the block has been moved to the right side of the board.

If the path will be negative then it means that the block has been moved to the left side of the board.

* swap (game, type, ind1, ind2)

If the type is ‘rows’ then this method will swap rows with indexes ind1 and ind2.

Else if the type is ‘cols’, then this method will swap columns with indexes ind1 and ind2.

Besides, this method creates a tips dictionary with the same structure as in the pull\_up method.

But instead of the id of the block, it will contain an index of the row/column.

Also, if the type is ‘rows’ and the path is negative then it means that the row is now closer to the bottom of the board.

Analogically, if the type is ‘rows’ and the path is positive then it means that the row is now closer to the top of the board.

* intervene (game, whos)

If whos is ‘player’ then this method starts to search for a block with id which is equal to the game.id on the whole board.

The block will be removed when he will be found.

Then this method will call the gravity method.

Because other blocks in real life would fall faster than we would even drop a new block to the board.

Next, this method will call the fall method (anticipatorily reducing game.id by 1).

Else if whos is ‘opponent’ then this method starts to search for a block with id which is equal to the game.id – 1 on the whole board.

The block will be removed if he will be found and game.id will be reduced by 2.

Notice, that if last fallen block of the opponent may be not found (if the player has removed him on his turn by using on this block, for example, pull out power) then nothing will happen.

But if the block was found then this method also will call gravity and fall methods.

Then it also will increase the game.id by one (so we will return to the latest block of the player).

Besides, this method creates tips tuple with 4 elements.

The first element of this tuple is an array which contains the id of the blocks which have fallen after destroying of the founded block.

The second element of the tips tuple is the id of the destroyed block.

And the last two elements of the tips tuple tells us about the relative bias of the destroyed block after he has fallen again.

* control (game, whos, col)

The purpose of this method is to make next falling block of the player (specified by the whos attribute) to fall into the specified column by changing game.pos1 or game.pos2 (depending on the whos attribute) to be equal to the col.

* pull\_out (game, ind, side)

The purpose of this method is to remove a block.

If the side is ‘bottom’ or ‘top’ then this method will remove a block from the specified by ind column.

Else if the side is ‘left’ or ‘right’ then this method will remove a block from the specified by ind row.

* shuffle (game, request)

The purpose of this method is to shuffle the board by specified request.

If request is ‘board’ then all block on the board will be randomly shuffled.

Else if request ‘rows’ then all rows on the board will be randomly shuffled.

Else if request ‘cols’ then all columns on the board will be randomly shuffled.

Else if request is (‘col’, ind) then all blocks in the column with index ind will be shuffled.

Else if request is (‘row’, ind) then all blocks in the row with index ind will be shuffled.

* check\_victory (game)

The role of this method is to check whether a victory or a draw situation has been reached. It will take a game object and return the appropriate number:

* + 0 if no one has won yet.
  + 1 if the first player won or made a turn which led both players to the win.
  + 2 if the second player won or made a turn which led both players to the win.
  + 3 if there is a draw.

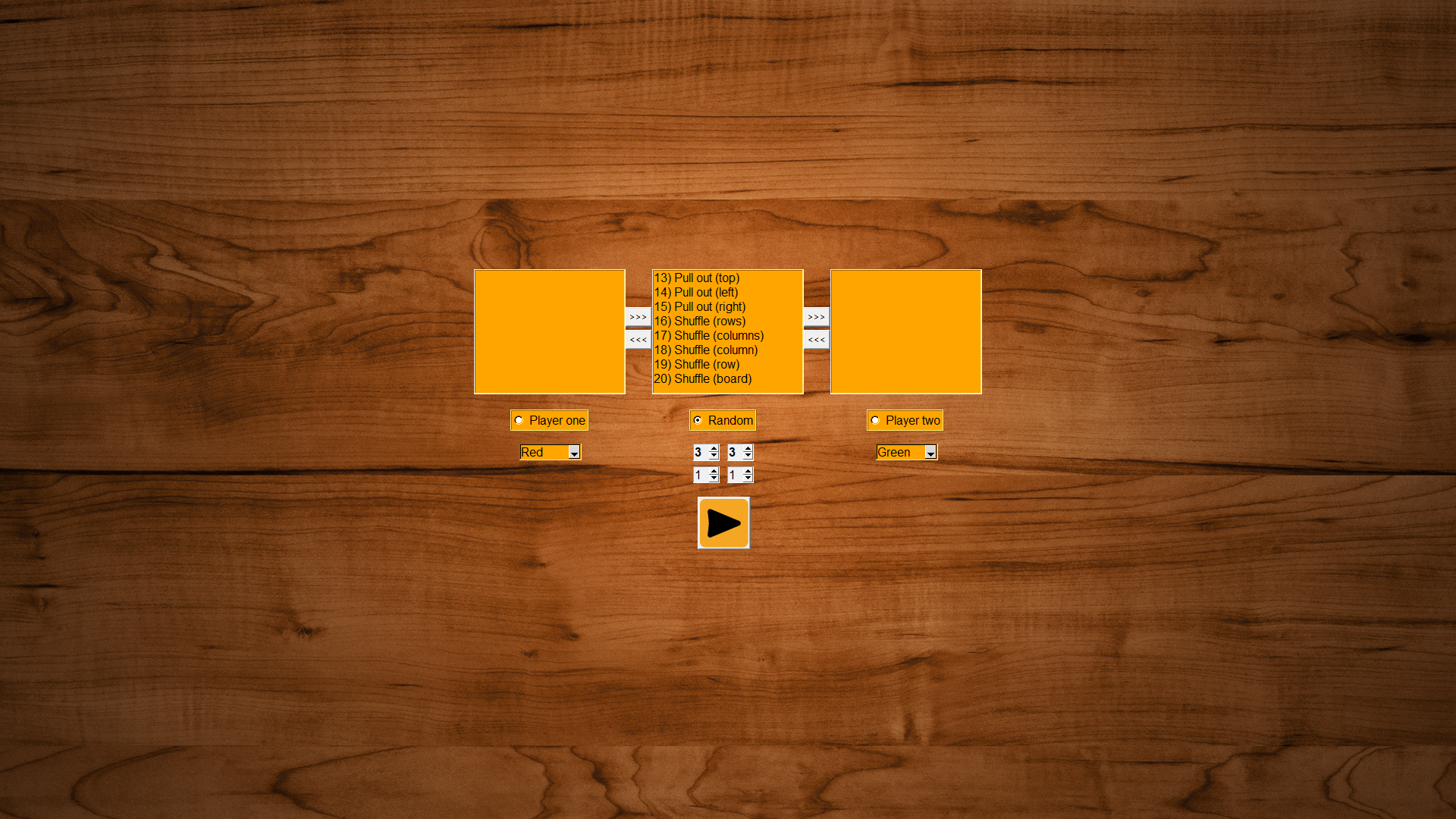
At first, this method will search through all rectangles with the width which is equal to the game.width and height which is equal to the game.height and check whether anyone has reached the win condition.

Then this method will do the same search for all rectangles with the width which is equal to the game.height and with height which is equal to the game.width.

**Structure of the frontend part:**

Graphical User Interface will consist of the root fullscreen window and some nested frames.

Window fullscreen mode can be disabled by the escape button.



Also, root window is responsible, so you can resize it as you wish and it still will look nice, but root also will have the min size of the window so you will always be able to see all important elements on the window.

The root window and all frames will have the same background image (and if you will resize the window images still won’t overlap because this was envisaged during developing).

At the start, will be displayed a start frame which allows you to choose options for a new game.



At the top, you can see 3 list boxes.

The left list box is able to contain super powers of the first player.

Right lis tbox is able to contain super powers of the second player.

And list box in the center contains all super powers.

Buttons between list boxes allow you to transfer elements of one list box to another list box.

On the line below you can see 3 radio buttons which allow you to choose who will take first turn (player one, player two or random).

On the sides of the line below, you can see two combo boxes which allow you to choose the color of each player.

The left one is for the first player and the right one for the second player.

Notice that players can’t have the same color (if you’ll try to start a game with the same color of players by clicking on the play button at the bottom of the frame then nothing will happen)

And in the center of the frame, you can see four spin boxes.

The left spin boxes is responsible for specifying the number of rows.

And the right spin boxes is responsible for specifying the number of columns.

Two above allows you to specify the size of the board (their numbers are bold).

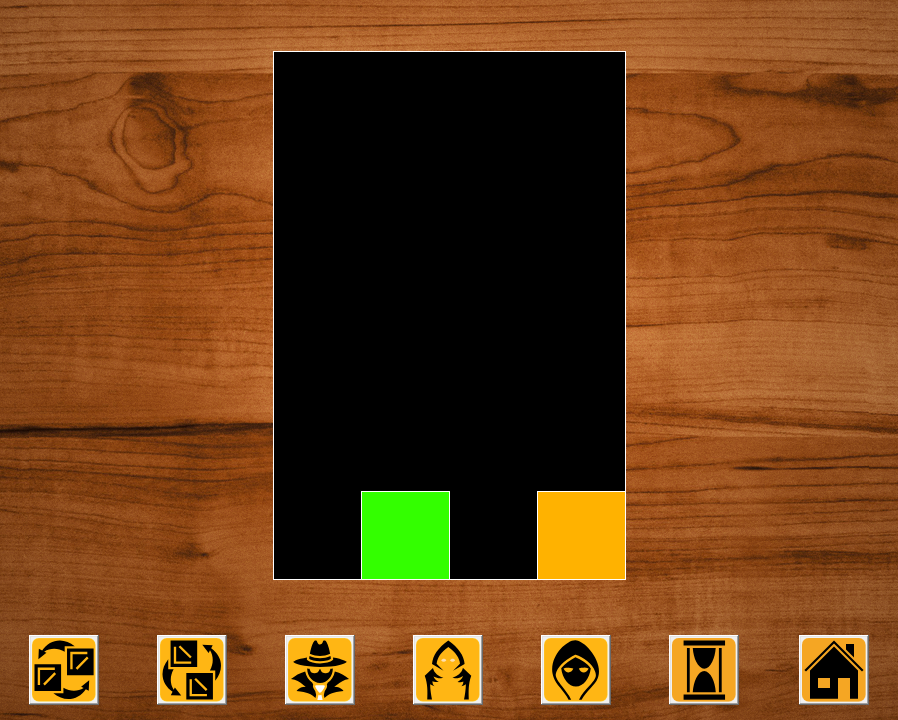
Two below allows you to specify the size of the rectangle which is needed to win.

Notice, that both number of columns and rows of the rectangle which is needed to win must be no more than (x + 1) // 2 where x number of rows of the board if you selecting number of rows of the winning rectangle or x is number of columns of the board if you selecting number of columns of the winning rectangle.

Also, winning rectanglecan’t have the 1x1 size because this would make no sense to play.

If those conditions aren’t met then you will be unable to start a new game (after clicking on the play button nothing will happen).

But if all conditions are met then after clicking on the play button start frame will be hidden and you’ll see the main frame (his size is 840x710).



This frame consists of a canvas element and buttons below him.

No matter which turn now or how many powers each player have you will always see two the same buttons:

 The purpose of the first button is to finish turn.

The purpose of the second button is to go back to the start frame (the current game will be finished).

Now let’s talk about the canvas element (his size is 840x640).

The board will always be centered on the canvas and will always have a nice size (even if it will consist of a few rows and columns) and even if you’ll rotate the board by rotate power then the board will not go beyond the canvas.

This behavior is achieved by using some math.

For example, when you rotate the board it might go beyond the canvas because the diagonal of the board is bigger than any of his side.

So we need to make the size of the board such that his diagonal will be always less than height and width of the board.

But the width of the canvas is bigger than it’s height so if we’ll make the diagonal of the board less than the height of the canvas then condition is met.

So we have: 640 >= where a – the number of rows of the board, b – the number of columns of the board, k – the size of the one block (because the board consists of blocks).

Inequality above is equal to the following: k <= 640 / .

So we can make the following conclusion:

If the size of every single block in the board will be the floor of the equation 640 / then all will be fine.

This conclusion not only making the board to not go beyond the board but also gives to the board a nice size because the size of each block on the board will be computing dynamically.

Also, you might already know that tkinter does not support background transparency.

So there was a problem about how to make all background images do not overlap even if you’ll resize the root window.

It was achieved by anchoring each frame to the center of the root window by binding resize function on each resizes of the root window.

This function moving each frame at the center of the root window.

Notice that each frame has its own constant size and the current minsize of the root window is exactly the size of the current frame on the screen.

Now let’s talk about interaction with buttons.

If power button require to select something (e.g. row, column or a single block on the edge of the board) then after clicking on the button visually nothing will happen but you should know that now selected mode is enabled and you can select.

If while the select mode is enabled you will click on another button then currently selected mode will be canceled so all will be fine.

If power button is on cooldown then you will be unable to click on that button until enough turns will be finished.

Also, during each animation, all buttons except home button will be unable to be clicked on so you won’t break the game.

If you having the intervene button then nothing will happen after clicking on this button if the needed block doesn’t exist.

Shuffle board, Shuffle rows, Shuffle columns buttons and rotate buttons will work even if the board is empty.

But if you’ll try to shuffle empty row or a column, pop out a block from the empty cell of the board, pull up all blocks on the empty row or column, swap empty rows or columns then after clicking on those buttons nothing will happen so those buttons won’t get the cooldown (that will save you from misclick).

Also, there is a feature that on the current turn the swap buttons will remember the last chosen index. So, for example, after clicking on the swap rows button if you will select two empty rows and after it select not empty row then last empty row and currently selected not empty row will be swapped.

Now let’s talk about types of animations which have been implemented:

1. teleport (tips)

This animation momentarily changes the coordinates of blocks with keys of passed tips dictionary using values of the tips dictionary.

1. stenosis (tips)

This animation at first scaling down each block in tips dictionary to their center’s points.

Then this animation using teleport function to move those points to their new coordinates.

Next, this animation using nested dilation animation which is reverse of the stenosis animation (so it’s scaling up each block in tips dictionary to their normal states).

1. fall()

This function is responsible for putting one block into the top of the board.

1. gravitate()

This animation simulates the gravity on the board and making all block fall up to the bottom of the board until they will meet an obstacle.

1. intervene (tips)

This animation flattening the specified block up to down and then using tips dictionary to place specified block on the right position above the board but with the usual size of the block. Next, this animation will call gravity function and after it, the fall animation will be called.

1. pull\_up (side)

This animation is like a gravitate but horizontal and depend on the side argument

1. pull\_out (ind, id, side)

This animation pulling out a block with id on the ind index at the specified side

1. rotate (degrees)

This animation is rotating the board and all blocks on it by multiplying their complex coordinates of each block by a complex angle.

So, for example, when I was implementing the intervene animation I’ve taken into consideration the fact that after rotating the board coordinates of the top corners of each rectangle on the board is changing.

To save the way intervene animation was working (flattening each block up to the down) I had to create an orientation variable.

A lot of functions using this variable.

For example, then you call a fall function it will drop already rotated block and etc.

Besides, in the code, you might see a counter variable.

This is a counter for animations which helps to understand then new animation can be started.

**User training:**

The user has to know the rules of the game and be able to correspond image of the power to the name of the power.

**Delivery/installation plan:**

To start the game you don’t need to install any module