Assignment 2

CSC148:

Assignment 2: Blocky

Due date: Tuesday, March 31, 2020 before noon sharp (not 12:10).

You may complete this assignment individually or with a partner who can be from any section of the course.

Learning goals

By the end of this assignment, you should be able to:

- model hierarchical data using trees
- implement recursive operations on trees (both non-mutating and mutating)
- convert a tree into a flat, two-dimensional structure
- use inheritance to design classes according to a common interface

Coding Guidelines

These guidelines are designed to help you write well-designed code that will adhere to the interfaces we have defined (and thus will be able to pass our test cases).

You must:

- write each method in such a way that the docstrings you have been given in the starter code accurately describe the body of the method.
- avoid writing duplicate code.
- write a docstring for any class, function, or method that lacks one.

You must **NOT**:

- change the parameters, parameter type annotations, or return types in any of the methods or function you have been given in the starter code.
- add or remove any parameters in any of the methods you have been given in the starter code.
- change the type annotations of any public or private attributes you have been given in the starter code.
- create any new public attributes.
- create any new public methods.
- write a method or function that mutates an object if the docstring doesn’t that it will be mutated.

- add any more import statements to your code, except for imports from the `typing` module.

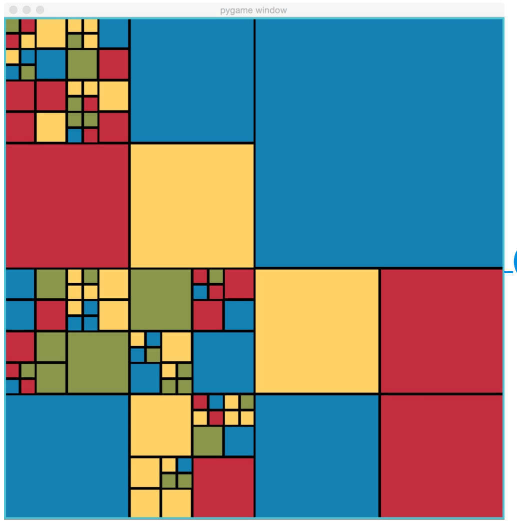
You may find it helpful to:

- create new private helper methods or functions for the classes you have been given.
    - if you create new private methods or functions you must provide type annotations for every parameter and return value. You must also write a full docstring for this method as described in the **function design recipe (https://q.utoronto.ca/courses/130571/files/5593988/download) ***
- create new private attributes for the classes you have been given.
    - if you create new private attributes you must give them a type annotation and include a description of them in the class's docstring as described in the **class design recipe (https://q.utoronto.ca/courses/130571/files/5594041/download) ***
- import more objects from the `typing` module
- override the inherited version of the `__eq__` special method in some cases (this is not the same as creating a new public method).

While writing your code you can assume that all arguments passed to the methods and functions you have been given in the starter code will respect the preconditions and type annotations outlined in the methods' docstrings.

Introduction: the Blocky game

Blocky is a game with simple moves on a simple structure. But, like a Rubik's Cube, it is quite challenging to play. The game is played on a randomly-generated game board made of squares of four different colours, such as this:



(https://q.utoronto.ca/courses/130571/files/6426939/download?wrap=1)

Each player has their own goal that they are working towards, such as creating the largest connected "blob" of blue. After each move, the player sees their score, which is determined by how well they have achieved their goal and which moves they have made. The game continues for a certain number of turns, and the player with the highest score at the end is the winner. Next, let's look in more detail at the rules of the game and the different ways it can be configured for play.

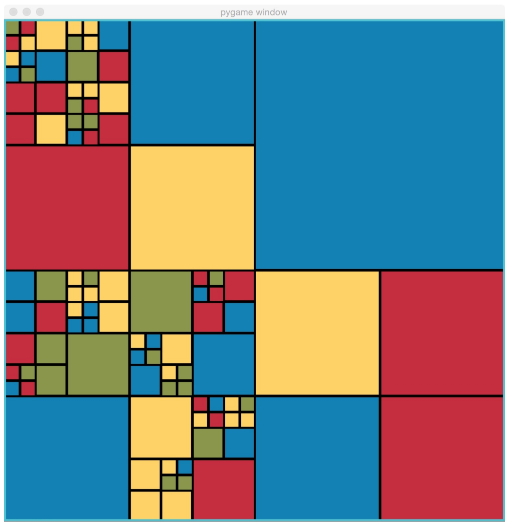
The Blocky board and terminology

We call the game board a 'block', which is best defined recursively. A **block** is either:

- a square of one colour, or
- a square that is subdivided into 4 equal-sized blocks.

The largest block of all, containing the whole structure, is called the **top-level block**. We say that the top-level block is at **level 0**. If the top-level block is subdivided, we say that its four sub-blocks are at level 1. More generally, if a block at level k is subdivided, its four sub-blocks are at level k+1.

A Blocky board has a **maximum allowed depth**, which is the number of levels down it can go. A board with maximum allowed depth 0 would not be fun to play on – it couldn't be subdivided beyond the top level, meaning that it would be of one solid colour. This board was generated with maximum depth of 5:



For scoring, the units of measure are squares the size of the blocks at the maximum allowed depth. We will call these blocks **unit cells**.

Actions and Moves

A number of **actions** can be performed in Blocky. A **move** is an action that is performed on a specific block. The actions are:

1. Rotate clockwise
2. Rotate counterclockwise
3. Swap Horizontally
4. Swap Vertically
5. Smash
6. Paint

7. Combine
8. Pass

The **Smash** action can only be performed on blocks with no children. If a block is smashed, then it is sub-divided into four new, randomly-generated sub-blocks. Smashing a unit cell is also not allowed (it is already at the maximum depth).

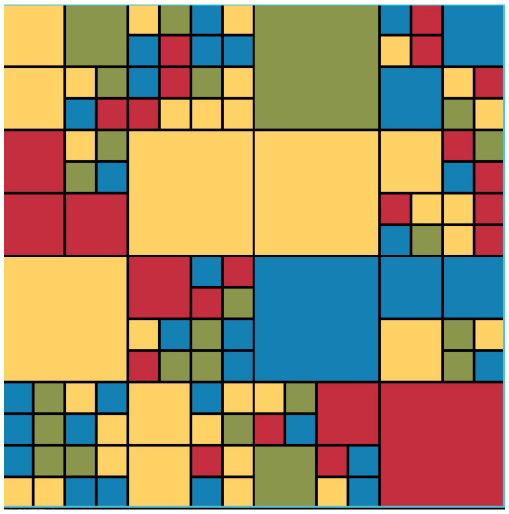
The **Paint** action sets a block's colour to a new, specified colour. It can only be performed on unit cells.

The **Combine** action turns a block into a leaf based on the majority colour of its children. It can only be performed on a block that is subdivided and whose children are at the maximum depth. If there is a majority colour among the four children, then the children are discarded and the block has the majority colour. A majority colour means that one colour is the majority amongst all children; a tie does not constitute a majority.

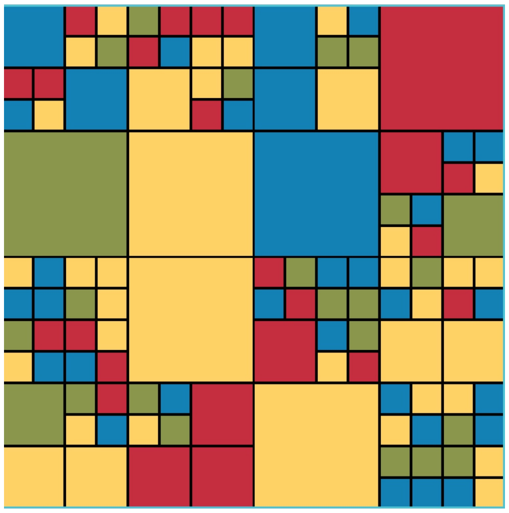
The **Pass** action does not mutate the block. It can be used by a player who wishes to skip their turn.

Choosing a block and levels

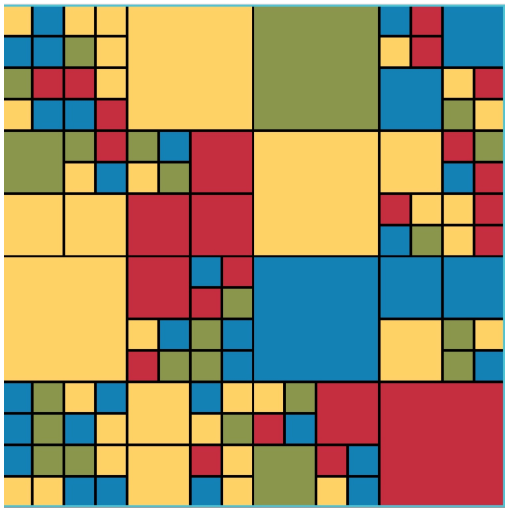
What makes moves interesting is that they can be applied to any block at any level. For example, if the user selects the entire top-level block for this board:



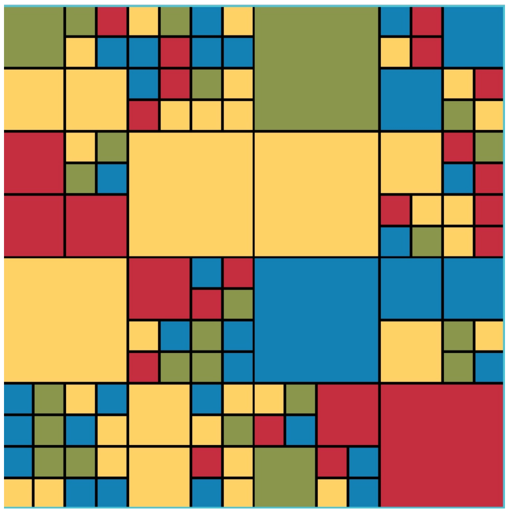
and chooses to rotate it counter-clockwise, the resulting board is this:



But if instead, on the original board, they rotated the block at level 1 (one level down from the top-level block) in the upper left-hand corner, the resulting board is this:



And if instead they were to rotate the block a further level down, still sticking in the upper-left corner, they would get this:



Of course there are many other blocks within the board at various levels that the player could have chosen.

Players

The game can be played solitaire (with a single player) or with two to four players. There are three kinds of players:

1. A **human player** chooses moves based on user input.
2. A **random player** is a computer player that, as the name implies, chooses moves randomly.
3. A **smart player** is a computer player that chooses moves more intelligently: It generates a set of random moves and, for each move, checks what its score would be if it were to make that move. Then it picks the one that yields the best score.

Goals and scoring

At the beginning of the game, each player is assigned a randomly-generated goal. There are two types of goal:

1. **Blob goal.**
   The player must aim for the largest “blob” of a given colour c. A **blob** is a group of connected blocks with the same colour. Two blocks are **connected** if their sides touch; touching corners doesn’t count. The player’s score is the number of unit cells in the largest blob of colour c.
2. **Perimeter goal.**
   The player must aim to put the most possible units of a given colour c on the outer perimeter of the board. The player’s score is the total number of unit cells of colour c that are on the perimeter. There is a premium on corner cells: they count twice towards the score.

Notice that both goals are relative to a particular colour. We will call that the **target colour** for the goal.

In addition to the points gained by the player from achieving their goal, a player can also lose points based on the actions they perform.

- Rotating, Swapping, and Passing cost 0 points.
- Painting and Combining cost 1 point each time they are performed.
- Smashing costs 3 points each time it is performed.

Configurations of the game

A Blocky game can be configured in several ways:

- *Maximum allowed depth.*
  While the specific colour pattern for the board is randomly generated, we control how finely subdivided the squares can be.
- *Number and type of players.*
  There can be any number of players of each type. The “difficulty” of a smart player (how hard it is to play against) can also be configured.

- *Number of moves.*
  A game can be configured to run for any desired number of moves. (A game will end early if any player closes the game window.)

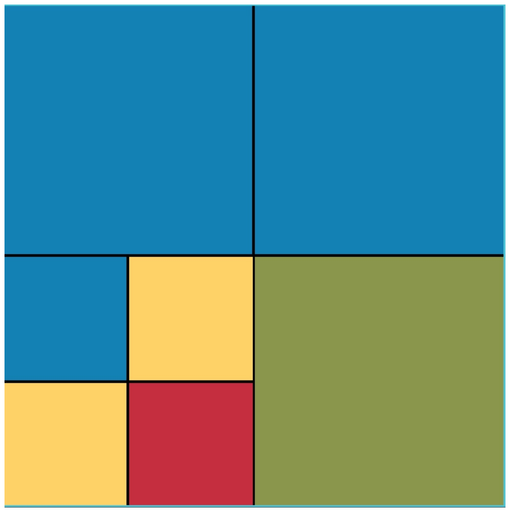
Setup and starter code

1. Download the zip file that contains the starter code here **a2.zip (https://q.utoronto.ca/courses/130571/files/6427246/download?wrap=1)** .
2. Unzip the file and place the contents in pycharm in your `a2` folder (remember to set your `a2` folder as a sources root)
3. You should see the following files and directories:
   - `actions.py`
   - `block.py`
   - `blocky.py`
   - `game.py`
   - `goal.py`
   - `player.py`
   - `renderer.py`
   - `settings.py`
   - `example_tests.py`
   - the `images` directory

Task 1: Understand the Block data structure

Surprise, surprise: we will use a tree to represent the nested structure of a block. Our trees will have some very strong restrictions on their structure and contents, however. For example, a node cannot have 3 children. This is because a block is either solid-coloured or subdivided. If it is solid-coloured, it is represented by a node with no children. If it is subdivided, it is subdivided into exactly four sublocks. Representation invariants document this rule and several other critically important facts.

1. Open `block.py`, and read through the class docstring carefully. A `Block` has quite a few attributes to understand, and the Representation Invariants are critical.
2. Draw the `Block` data structure corresponding to the game board below, assuming the maximum depth was 2 (and notice that it was indeed reached). You can just write a letter for each colour value. Assume that the size of the top-level block is 750.



Did you draw 9 nodes? Do the attribute values of each node satisfy the representation invariants? Note: If you come to office hours, we will ask to see your drawing before answering questions!

Task 2: Initialize Blocks and draw them

File(s): `block.py`, `blocky.py`

With a good understanding of the data structure, you are ready to start implementing some of the key functionality. You should be able to run the game, though it will only show you a blank board. In order to see an actual Blocky board, you must:

1. To make a `Block` drawable, you must implement `_block_to_squares` in `blocky.py`
2. To make a `Block` interesting, you must implement `Block.smash` in `block.py`

For implementing `_block_to_squares`, read the documentation from `Block`, paying special note of the positions and sizes of a `Block` and its children. Here is the strategy to use for implementing `Block.smash`: If a `Block` is not yet at its maximum depth, it can be subdivided; this method will decide whether or not to actually do so. To decide:

- Use function `random.random` to generate a random number in the interval [0, 1).
- Subdivide if the random number is less than `math.exp(-0.25 * level)`, where `level` is the level of the `Block` within the tree.
- If a `Block` is not going to be subdivided, use a random integer to pick a colour for it from the list of colours in `settings.COLOUR_LIST`.

Method `Block.smash` is responsible for giving appropriate values to the attributes of all `Block`s within the `Block` it generates. Notice that the randomly-generated `Block` may not reach its maximum allowed depth. It all depends on what random numbers are generated.

Check your work: We have provided an implementation of `Block.__str__` that allows you to print a `Block` in text form. Use this to confirm that your `Block.__smash__` method works correctly. We have also provided some pytest code for testing the `_blocks_to_squares` function in the `blocky.py` module.

See `example_tests.py` for the test case. If both are working properly, then you should see a correctly initialized game board when you run the game.

Task 3: The goal classes and random goals

File(s): `goal.py`, `settings.py`

We need to have some basic ability to set goals and compute a player's score with respect to their goal. Get familiar with the abstract `Goal` class in `goal.py`. It defines two abstract methods: `score` and `description`. `Goal` has two subclasses: `BlobGoal` and `PerimeterGoal`. The basic skeletons for these classes are provided.

Implement the function `generate_goals` in the `goal.py` module. This function generates a list of random goals. Each goal is of the same type (i.e., `BlobGoal` or `PerimeterGoal`) but has a different colour. The supported colours can be found in the `settings.py` module in a Python list called `COLOUR_LIST`.

Once you have appropriately implemented `generate_goals`, implement the `BlobGoal.description` and `PerimeterGoal.description` methods. You should see these descriptions at the bottom of the window when running the game.

Task 4: The `Player` class and random players

File(s): `player.py`

Get familiar with the abstract `Player` class in `player.py`. A concrete implementation of a human player is given to you; see: `HumanPlayer`. In order for the user to be able to play the game, they must be able to select by hovering over it with the mouse. See the `HumanPlayer.get_selected_block` method. For this to work, you must implement the function `_get_block` in the `player.py` module. Note: Do **not** make changes to the `HumanPlayer` class.

In order to support more than one player, you will need to implement the `create_players` function in `players.py`. This function will generate the right number of human players, random players, and smart players (with the given difficulty levels), in that order. Give the players consecutive player identifiers (IDs), starting at 0. Assign each player a random goal.

Check your work: You should be able to run a game with only human players. Try running function `two_player_game` – you can uncomment out the call to it in the main block of module `game`. To select a block for action, put the cursor anywhere inside it and use the W and S keyboard keys to select the desired level. The area to the right of the game board gives you instructions on the key to press for each action.

So far, no real moves are happening and the score never changes. But, you should see the board and see play pass back and forth between players. Finally, the game should end when the desired number of moves has been reached.

Task 5: The Blocky actions

Let's improve playability by supporting all the actions in Blocky. Each of these actions will mutate `Blocks` in some way.

Before you begin, you should review the representation invariants for class `Block`. They are critical to the correct functioning of the program, and it is the responsibility of all methods in the class to maintain them. When you are done, double check that each of your mutating methods maintains the representation invariants of class `Block`.

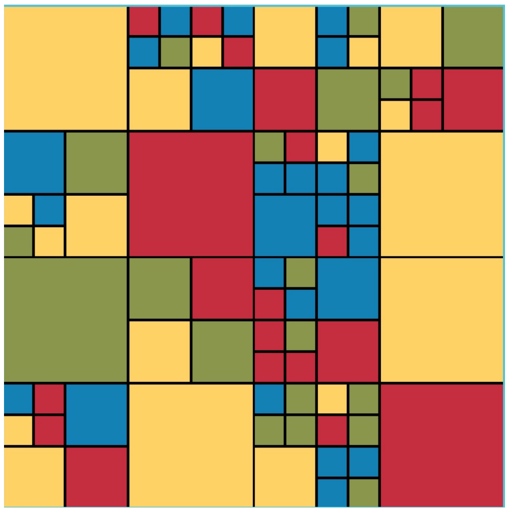
Implement also the `Block._update_children_positions` method and consider it as a handy helper method for other `Block` methods than change the order of children. The (x, y) coordinates for the upper left corner of a child block are different depending on which child it is!

Check your work: Now when you play the game, you should see the board changing. You may find it easiest to use function `solitaire_game` to try out the various moves.

Task 6: Implement scoring for perimeter goals

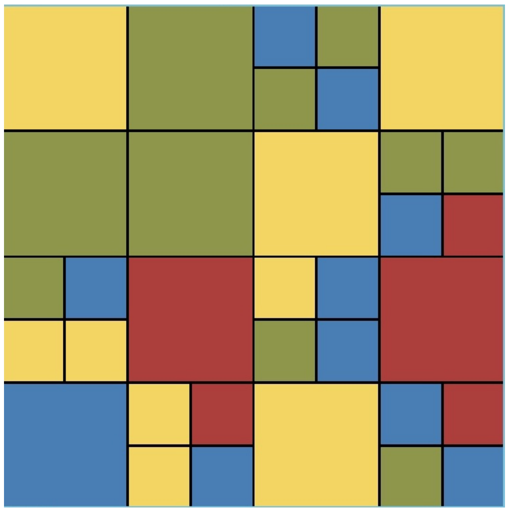
File(s): `goal.py`

Now let's get scoring working. The unit we use when scoring against a goal is a unit cell. The size of a unit cell depends on the maximum depth in the `Block`. For example, with maximum depth of 4, we might get this board:



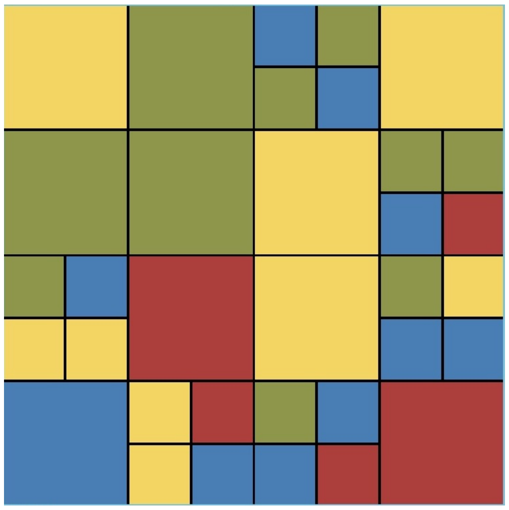
If you count down through the levels, you'll see that the smallest blocks are at level 4. Those blocks are unit cells. It would be possible to generate that same board even if maximum depth were 5. In that case, the unit cells would be one size smaller, even though no Block has been divided to that level.

Notice that the perimeter may include unit cells of the target colour as well as larger blocks of that colour. For a larger block, **only the unit-cell-sized portions on the perimeter count**. For example, suppose maximum depth were 3, the target colour were red, and the board were in this state:



Only the red blocks on the edge would contribute, and the score would be 4: one for each of the two unit cells on the right edge, and two for the unit cells inside the larger red block that are actually on the edge. (Notice that the larger red block isn't divided into four unit cells, but we still score as if it were.)

Remember that **corner cells count twice towards the score**. So if the player rotated the lower right block to put the big red block on the corner (below) the score would rise to 6:



Now that we understand these details of scoring for a perimeter goal, we can implement it.

1. It is very difficult to compute a score for a perimeter goal or a blob goal by walking through the tree structure. (Think about that!) The goals are much more easily assessed by walking through a two-dimensional representation of the game board. Your next task is to provide that possibility: in module `goal.py`, define function `_flatten`.
2. Now implement the `score` method in class `PerimeterGoal` to truly calculate the score. Begin by flattening the board to make your job easier!

Check your work: Now when you play the game, if a player has a perimeter goal, you should see the score changing. Check to confirm that it is changing correctly.

Task 7: Implement scoring for blob goals

File(s): `goal.py`

Scoring with a blob goal involves flattening the tree, iterating through the cells in the flattened tree, and finding out, for each cell, what size of blob it is part of (if it is part of a blob of the target colour). The score is the biggest of these.

But how do we find out the size of the blob that a cell is part of? (Sounds like a helper method, eh?) We'll start from the given cell and

- if it's not the target colour, then it is not in a blob of the target colour, so this cell should report 0.
- if it is of the target colour, then it is in a blob of the target colour. It might be a very small blob consisting of itself only, or a bigger one. It must ask its neighbours the size of blob that *they* are in, and then use that to report its own blob size. (Sounds, recursive, eh?)

A potential problem with this is that when we ask a neighbour for their blob size, they will count us in that blob size, and this cell will end up being double counted (or worse). To avoid such issues, we will keep track of which cells have already been "visited" by the algorithm. To do this, make another nested list structure that is exactly parallel to the flattened tree. In each cell, store:

- -1 if the cell has not been visited yet
- 0 if it has been visited, and it is not of the target colour
- 1 if it has been visited and is of the target colour

Your task is to implement this algorithm.

1. Open `goal.py` and read the docstring for helper method `BlobGoal._undiscovered_blob_size`.
2. Draw a 4-by-4 grid with a small blob on it, and a parallel 4-by-4 grid full of -1 values. Pick a cell that is in your blob, and suppose we call `BlobGoal._undiscovered_blob_size`. **Trace what the method should do. Remember not to unwind the recursion!** Just assume that when you ask a neighbour to report its answer, it will do it correctly (and will update the `visited` structure correctly).
3. Implement `BlobGoal._undiscovered_blob_size`.
4. Now replace your placeholder implementation of `BlobGoal.score` with a real one. Use `_undiscovered_blob_size` as a helper method.

Although we only have two types of goal, you can see that to add a whole new kind of goal, such as stringing a colour along a diagonal, one would only have to define a new child class of `Goal`, implement the `score` method for that goal, and then update the code that configures the game to include the new goal as a possibility.

Check your work: Now when you play the game, a player's score should update after each move, regardless of what type of goal the player has.

Task 8: Add random players

File(s): `player.py`

Before you can implement a `Player` sub-class, you need to understand the game loop. In `game.py`, the `Game.run_game` method contains what seems like an infinite loop. This is the game loop, and it will iterate many times per second. On each iteration, the game will ask the operating system for any events that have occured (i.e., it will call `pygame.event.get()`). The game itself reacts to only the events it is interested in – we call this *event-driven programming*. The loop will exit if the event indicates that the user would like to quit the game (e.g., if they have closed the window). Otherwise, the event is forwarded to a `Player` object. After all events are processed, the game is updated by asking the player to make a move. Because the game loop needs to keep iterating quickly, our `Player` object must respond *quickly*.

The `HumanPlayer` object is already implemented and reacts to specific keyboard events. When a keyboard key is pressed, the `HumanPlayer` object will translate the key into a desired action (if possible). A very short time (fractions of a second) later, the game will ask the player to make a move. If there is no desired action, then the `HumanPlayer` will not make a move (i.e., by returning `None`). Otherwise, the action that was translated from a keyboard event is converted into a move based on the block the user is selecting.

For both `RandomPlayer` and `SmartPlayer`, we have provided a private attribute `_proceed`. This attribute is initially `False`. Only when the user clicks their mouse will the attribute be set to `True` (e.g., see `RandomPlayer.process_event`) This communicates that the `RandomPlayer` or `SmartPlayer` should make a move. Once a move is determined, you must set the `_proceed` attribute back to `False` so that the next time it is this player's turn, the player will wait for a mouse click again. The skeleton code for this is already provided for you in both `RandomPlayer.generate_move` and `SmartPlayer.generate_move`.

We will now implement the class `RandomPlayer`. **Do not change the following methods:**

- `RandomPlayer.get_selected_block`
- `RandomPlayer.process_event`

Your first task is to implement the initializer (`RandomPlayer.__init__`). You must inherit the attributes of the `Player` class and document them. Once that is done, you will need to implement how the `RandomPlayer` will make a random but **valid** move on the board. We will do this in the `RandomPlayer.generate_move` method. **You must note mutate the given `board` parameter** when generating a random move! In order to assess whether a move is valid, the `RandomPlayer` must create a copy of the board for each move it wants to try. To do this, to implement the `Block.create_copy` method. Once you have a copy, you can mutate that copy by applying the move on it and checking to see if it was successful. If you implemented `Block.create_copy` correctly, then mutating the copy should not impact the original board.

Hint: No block from your copy should not have the same `id` as a block from the original board. But you can always find the block you copied from the original board based on the `position` and `level` of the copy. Do you already have a function that can do this?

Task 8: Add smart players

File(s): `player.py`, `block.py`

We will now implement the class `SmartPlayer`. **Do not change the following methods:**

- `SmartPlayer.get_selected_block`
- `SmartPlayer.process_event`

Your first task is to implement the initializer (`SmartPlayer.__init__`). You must inherit the attributes of the `Player` class and document them. A `SmartPlayer` has a “difficulty” level, which indicates how difficult it is to play against it.

Next, implement how the `SmartPlayer` will make moves on the board by completing the `SmartPlayer.generate_move` method. A `SmartPlayer` randomly generates n valid moves, where n is the `SmartPlayer`’s difficulty value. It then picks the one that yields the best score, **without taking into account any penalty that might apply to an action**.

When scoring each of the valid moves, **you must not mutate the given `board` parameter**. The `SmartPlayer` must create a copy of the board for each move it wants to assess. One you have a copy, you can mutate that copy by applying the move on it.

If no best move was found and the current score is best, the `SmartPlayer` should pass.

Check your work: Now you can run games with all types of players.

Polish!

Take some time to polish up. This step will improve your mark, but it also feels so good. Here are some things you can do:

- Pay attention to any violations of the “PEP8” Python style guidelines that PyCharm points out. Fix them!
- In each module you are submitting, run the provided `python_ta.check_all()` code to check for errors. Fix them!
- Check your docstrings to make sure they are precise and complete and that they follow the conventions of the Function Design Recipe and the Class Design Recipe.
- Read through and polish your internal comments.

- Remove any code you added just for debugging, such as print statements.
- Remove any `pass` statement where you have added the necessary code.
- Remove the word “TODO” and/or “FIXME” wherever you have completed the task.
- Take pride in your gorgeous code!

Submission instructions

The following files are not to be submitted: `actions.py`, `game.py`, `renderer.py`, `settings.py`. Your code should run as if they were not modified from the original starter code.

1. Login to MarkUs and create a group for the assignment (or specify that you’re working alone).
2. **DOES YOUR CODE RUN**?!
3. Submit the files: `block.py`, `blocky.py`, `goal.py`, `player.py`.
4. On a fresh Teaching Lab machine, download all of the files you submitted, and test your code thoroughly. *Your code will be tested on the Teaching Lab machines, so it must run in that environment.*
5. Congratulations, you are finished with your third (and last) assignment in CSC148! Go have some chocolate or do a cartwheel. :)