Assignment 4: implement a Tetris autoplayer

In this assignment, you will write an AI that plays Tetris. You will push your code to a Git repository, where a server will run it and keep track of your best score.

To get a passing grade (40%) all you need to do is beat the random autoplayer you'll be given. Everyone should be able to do this. To get 100%, you also need to beat everyone else in the year. You grade will be determined by the median score your code achieves from multiple runs with different random block sequences.

To start, perform the following steps:

1 Register for a GitLab account

If you have not done so already, register for an account on UCLs new GitLab instance (`https://scm4.cs.ucl.ac.uk`).[1] You need to use your `@ucl.ac.uk` email address for this. Please *use your real name* when registering; otherwise, we will not know which submissions are yours!

2 Register on the web interface

The web interface that we will be using can be found on `https://engf0002.cs.ucl.ac.uk`. Go to that page, and click the button that reads "Sign in with GitLab". You will be redirected to a page that asks you to grant API access to the grading server; click "Authorize".

3 Fork the boilerplate code

If all went well, you have been redirected back to the web interface. Under the heading "Available contests", there should be an entry labelled "Tetris". Click the button that says "Enrol". This will create a copy (*fork*) the Tetris boilerplate repository into your GitLab account.

4 Clone your fork

To pull in the code, you need to install Git (`https://git-scm.com/`); refer to that website for instructions on how to install Git on your system. After installation, you can pull in your code by running:

```
git clone https://scm4.cs.ucl.ac.uk/YOUR_GITLAB_USERNAME/tetris.git
```

You will be prompted for your GitLab username and password. If this worked, you should have a folder named `tetris` in your working directory. Enter this folder by running `cd tetris`.

5 Run the code

Please use a recent version of Python 3. You can download a bundle with all you need from `https://www.python.org/`. The auto-grader is currently running python 3.7.9, but you should be able to run more recent versions.

Once you are in the code directory, you should be able to start the interface by running

```
python visual.py
```

To get a feeling for the game, you can also run any of the interfaces in manual mode by adding the flag `-m`:

```
python visual.py -m
```

If that interface does not work, there are two alternative interfaces available:

- A command-line interface; run `python cmdline.py` to use it. If you are using Windows, you may need to install the `windows-curses` package first; try running `pip --user install windows-curses`.
- A Pygame-based interface; run `python visual-pygame.py` to use it. For this, you need to have a working copy of Pygame; run `pip --user install pygame` to install it.

The pygame version is recommended as it works fastest when autoplaying.
If you get an interface running, you should see the default player in action. This player is not very clever: it chooses random actions in hopes of eliminating a line. It is your job to write a better AI.

[1] Not to be confused with the old and deprecated GitLab instance (`https://gitlab.cs.ucl.ac.uk`).

6 Our Tetris Variant

In the regular version of Tetris blocks of four squares known as Tetronimos drop, and you must guide them to a good landing place. When one or more lines is completed all the blocks in those lines are eliminated, the blocks above drop, and you score extra points. The goal is to score as many points as possible before the blocks reach the top of the screen. Normally the game gets faster as it progresses, so gets harder.

Our version has a number of differences from normal Tetris:

- If you play it manually, it doesn't get faster. This is because, within limits, we don't want the amount of time you take to calculate a move to matter.
- You only have 400 tetronimos to land. This changes the gameplay significantly. Just staying alive to the end is not going to result in a really high score.
- The scoring is weighted heavily in favour of eliminating more rows at a time. The scores for eliminating 1-4 rows are 25, 100, 400, 1600, so there's a big benefit from not eliminating one row at a time.
- You can discard a tetronimo you don't like. You only get to do this 10 times though. This makes play easier for a human, but it's not necessarily easy for an AI to decide to do this.
- You can substitute a bomb in place of the next tetronimo! When a bomb lands it destroys all the immediately surrounding blocks, and all the blocks above it or the neighbouring squares will fall to the ground. Blocks destroyed by bombs don't score anything. You only get five bombs, so use them wisely. Bombs also make play easier for a human, but add a new element for an AI.

Keybindings

In manual play, the key bindings are:

↑ - rotate clockwise

→ - move right

← - move left

↓ - move down

Space - drop the block

z - rotate anticlockwise

x - rotate clockwise

b - switch next piece for bomb

d - discard current piece

Esc - exit game

If you don't like the key bindings, feel free to change them. They're in `key_to_move` in visual-pygame.py, `UserPlayer.key()` in visual.py, or `UserPlayer.choose_action()` in cmdline.py.

7 Write your player

Your code should be placed in the file `player.py`. A player is a class with a function called `choose_action`. This function will be called with a copy of the Tetris board in the variable `board` every time an action by the player is required. The `choose_action` function you write should decide what actions to take based on the state of the board. To this end, you can use the following information:

- The variable `board.falling` contains information about the currently falling block; most importantly, its shape (`board.falling.shape`) and the coordinates of its cells (`board.falling.cells`). The shape is an instance of the enum `shape`, whose definition can be found in `board.py`.
- The variable `board.next` contains information about the next block that will fall. The `cells` do not mean anything yet (the block is not on the board), but you can check out the `shape`. It is possible that `board.next` may be None if you're testing moves in the sandbox (see below) and the first block has landed.

- The property `board.cells` contains information about the currently occupied cells on the board. More precisely, this property is a **set** of tuples `(x, y)` that signify locations. Note: the origin of the board is in the top-left corner; the x-axis extends to the right, and the y-axis extends downwards.

You can iterate over a set the same way you might iterate over a list; for instance:

```
for (x, y) in board.cells:
    # Do something here.
```

In the above, the order of cells is not guaranteed. One thing you may want to do, as you analyse the board, is iterate over the cells already occupied from top to bottom, left to right. This can be done as follows:

```
for y in range(board.height):
    for x in range(board.width):
        if (x, y) in board.cells:
            # Do something here.
```

The board also has five methods that can be used to try out the effect of actions:

- The method `move()` accepts an instance of the enum `Direction`, whose definition can be found in `board.py`. For instance, to try moving the block to the left, use `board.move(Direction.Left)`. The instance of `board` will then be updated with the effect of that move. To drop the block, use `Direction.Drop`.
- The method `rotate()` accepts an instance of the enum `Rotation`; again, refer to `board.py` for the definition. For instance, to rotate a block anticlockwise, use `board.rotate(Rotation.Anticlockwise)`.
- The method `skip()` simulates the effect of skipping a move; it does not take any arguments.
- The method `discard()` simulates the effect of discarding the current block; it does not take any arguments. If you have no discards left the call will behave the same as `skip()`.
- The method `bomb()` simulates the effect of switching the next block for a bomb; it does not take any arguments. If you have no bombs left, or if board.next is None, the call will behave the same as `skip()`.

Calling any of the above (including `skip`) will also apply the implicit move down that occurs once every turn. Furthermore, if the action (in the case of a move) or the subsequent implicit move down caused the block to land, these functions will return `True`; otherwise, they will return `False`.

To experiment with multiple options, it is useful to get a copy of the board and try the effects of your actions there; actions on the copy will not affect the original, so you can use that state later:

```
sandbox = board.clone()
sandbox.rotate(Rotation.Anticlockwise)
sandbox.move(Direction.Drop)
sandbox.move(Action.Bomb)
# Check if the new board in sandbox is to your liking
```

Once your player is ready to commit to a action, you can tell the interface by returning that action. For instance, you can use `return Rotation.Clockwise` to tell the interface that you want to rotate the block clockwise, or `return Action.Discard` to tell the interface to discard the current block. To skip a move, use the special value `None`, i.e., write `return None`.

If your AI has a plan of multiple action you can either return those as a list, i.e.,

```
return [Rotation.Anticlockwise, Direction.Left, Direction.Drop, Action.Bomb]
```

or you can use the `yield` statement to return each individual action

```
yield Rotation.Anticlockwise
yield Direction.Left
yield None
yield Direction.Drop
yield Action.Discard
```

The latter also works in loops, and can be a nice way to synthesize a plan based on your earlier explorations without first having to build a list and return that. If you're not comfortable with Python's `yield` command, it's fine to just return actions.

If your player returns multiple actions, those actions will only be executed until the currently falling block lands. After that, the system will disregard the remaining moves, generate a new block, and your function `choose_action` will be called again, with the new board containing the new falling piece (previously `board.next`) as well as the new next piece. On the other hand, if the moves supplied by your player run out before the block lands, the function `choose_action` will be called again as well.

8 Push your code

Once you are happy with your player, you can commit your changes to the repository. First, stage your changes by running `git add`, e.g., to stage your changes to `player.py`, run `git add player.py`. Next, you commit your changes by running `git commit`; this will bring up an editor asking for a commit message, where you describe your changes. Once you save and close the editor, the commit will be made. You can also use the `-m` flag to type your message on the command line, e.g. `git commit -m "Fixed all the bugs."`.

Finally, you can upload (*push*) your code to GitLab by running `git push`. You will again be prompted for your GitLab username and password when you do this. When your code is pushed, the server will run it on some (secret) inputs. You should be able to track the score of your submission in real time through the web interface, i.e., on `https://engf0002.cs.ucl.ac.uk/submissions/tetris`.

Near to the deadline, the grading server will get busy. The grading server will grade your submissions in the order you push them, and will use a *fair queuing algorithm* to ensure that you don't get more than your fair share of submissions when it is busy. Fair queuing means that when it wants to choose a submission to run, it will look at all users with queued submissions; the next one it will run is from the user whose last submission it ran was the longest ago. If you keep modifying your code and keep pushing, it may take a long time to get to your most recent submission. **If you no longer care about the result (because, for example, you just fixed a bug), consider cancelling old submissions (via the submissions page) before they are picked up to run.**

Caveat: changing files and state You should not have to change anything but `player.py`, but modifying other files or adding files is not prohibited. In the best case, any changes to the game logic only have an effect locally — the test environment maintains its own copy of the game logic; in the worst case, your changes may crash the code as it is evaluated. Similarly, the test environment maintains its own copy of the board based on the actions that your player takes; the player will not be able to access this state. This means that, for instance, changing `board.score` to a value of your liking will not change the score maintained by the server.

Optional: generate a keypair Typing your username and password for every push can get cumbersome after a while. You can get around this by generating a private and public key, and adding the public key to your GitLab account. First, generate your keypair by running

```
ssh-keygen -t rsa -b 2048
```

This may ask for a passphrase for the key; you can leave it empty if you like, or encrypt the key with your passphrase for extra security. To add the public key to GitLab, first copy the output of the following command:

```
cat ~/.ssh/id_rsa.pub
```

Next, go to your SSH keys on on GitLab (click the icon in the upper-right corner, click "Settings", click "SSH Keys" in the menu on the left) and paste the public key in the field labelled "Key". Once you give the key a title, you can click "Add key". Finally, we need to adjust the way git talks to GitLab, by running

```
git remote set-url origin git@scm4.cs.ucl.ac.uk:YOUR_GITLAB_USERNAME/tetris.git
```

You should now be able to push and pull without giving your credentials (but possibly typing the passphrase for your key if you have not left it empty when generating it).

A word on fairness

To ensure that all submissions get evaluated on a level playing field, the server will evaluate your code in a limited environment. In particular, this means that you are given a limited amount of memory and CPU usage to make your calculations. We don't expect you will hit these limits with a reasonable algorithm. Moreover, your code will not be able to write to the directory where it is running, and it will also not be able to communicate over the network.

Your code will also be given a limited number of blocks and a limited amount of time to run in, so your aim is not to play forever, but to achieve a good score in the time you have. If your code runs out of time or blocks, it will be stopped, but the score will still count. However, if your code runs out of memory or crashes for some other reason, its score will be kept but the submission will not be counted.

We have made every effort to make sure that these limitations are obeyed; if you somehow find a way to circumvent them, we would like to hear it. The real challenge, however, should be to write a great autoplayer!