Write a class named BlackBoxGame for playing an abstract board game called Black Box. You can see the rules here (https://en.wikipedia.org/wiki/Black\_Box\_%28game%29). It takes place on a 10x10 grid. Rows 0 and 9, and columns 0 and 9 (border squares), are used by the guessing player for shooting rays into the black box. The atoms are restricted to being within rows 1-8 and columns 1-8.

In our version, the guessing player will start with 25 points. As stated on the Wikipedia page, "Each entry and exit location counts as a point" that is deducted from the current score. If any entry/exit location of the current ray is shared with any entry/exit of a previous ray, then it should not be deducted from the score again. Each incorrect guess of an atom position will cost 5 points, but repeat guesses should not be deducted from the score again.

There is an online implementation here (http://www.pythononline.co.uk/blackbox/) that you can try out to get a feel for the game. It uses different scoring and only allows exactly four atoms (whereas our version allows any number of atoms >= 1).

Tip: Probably the easiest way of representing the board is to use a list of lists.

You're required to print the board for testing purposes.

Your BlackBoxGame class must include the following methods:

* An init method that takes as its parameter a list of (row, column) tuples for the locations of the atoms in the black box, and initializes any data members. You may assume that the given coordinates are valid and don't contain any duplicates. You may also assume that the list contains at least one tuple.
* A method named shoot\_ray that takes as its parameters the row and column (in that order) of the border square where the ray originates. If the chosen row and column designate a corner square or a non-border square, it should return False. Otherwise, shoot\_ray should return a tuple of the row and column (in that order) of the exit border square. If there is no exit border square (because there was a hit), then shoot\_ray should return None. The guessing player's score should be adjusted accordingly. (Note: if the return value of a function can have different types, it's a very good idea to specify that in the docstring.)
* A method named guess\_atom that takes as parameters a row and column (in that order). If there is an atom at that location, guess\_atom should return True, otherwise it should return False. The guessing player's score should be adjusted accordingly.
* A method named get\_score that takes no parameters and returns the current score.
* A method named atoms\_left that takes no parameters and returns the number of atoms that haven't been guessed yet.

Feel free to add whatever other classes, methods, or data members you want. All data members must be **private**. All methods must have **no more than 20-25 lines of code** - don't try to get around this by making really long or complicated lines of code. (The rule in real life is just to not stuff too much into a single function, but that's probably too nebulous a rule for you at this point, and if your function is over 25 lines, you probably are trying to stuff too much into it.)

Whether you think of the list indices as being [row][column] or [column][row] doesn't matter as long as you're consistent.

Here's a very simple example of how the class could be used:

game = BlackBoxGame([(3,2),(1,7),(4,6),(8,8)])

move\_result = game.shoot\_ray(3,9)

game.shoot\_ray(0,2)

guess\_result = game.guess\_atom(5,5)

score = game.get\_score()

atoms = game.atoms\_left()