Single-Agent Combat

In this challenge you are required to supply a decision-making agent to win in a 1-on-1 combat in an urban area between two ground vehicles.
You are given a digital-surface-map (or DSM) of an urban village.
There are two agents you (the blue agent) and the enemy (the red agent).

Rules of the game

The rules of engagement are as follows:

1. Each agent begins a game with a random position.
2. At each step, the agent may choose one of 9 possible actions - (8 directions - top-right, right, bottom-right, bottom, top, bottom-left, left, top-left and stay in place).
3. At each step, the world queries each of the players for their actions, and moves them accordingly.
4. It then decides that the blue (red) agent won if two conditions are satisfied:
    i. There is a line of sight (LOS) between blue and red
    ii. There exists a blue maneuver, such that if red stays put, there is no LOS
    iii. There is no red maneuver, such that if blue stays put, there is no LOS.

    In other words, blue wins, if it can force the red agent into a position, where he can "shoot" the red agent, and hide behind a building, whereas the red agent is out in the open. The game then defines a "terminal state" as a state where one agent wins, or both agents are at the same position.
5. Finally, the game provides a score to each agent. The scoring is as follows: the victorious party receives `250 - 5 * #num. of moves` points, and the losing party receives `-250 + 5 * # num. of moves` points, this is in order to maintain this as a zero-sum game.

Capture the Flag !

The main simulation (main.py) runs in 200 episodes, where each episode is a separate game.

Each game lasts for at most 250 steps, after which a tie is declared.

In order to capture the flag the agent must win at least 150 episodes against the designated enemy player.

The level of the designated enemy player is chosen upon initialization of the decision maker
with a parameter that is `EASY_AGENT`, `MEDIUM_AGENT`, `HARD_AGENT`

The enemy player is formed by constructing a class of RafaelDecisionMaker with the appropriate parameter.

For example:

```
red_decision_maker = RafaelDecisionMaker(HARD_AGENT)
```

1. **Easy** - EASY_AGENT
2. **Medium** - MEDIUM_AGENT
3. **Hard** - HARD_AGENT
4. **King of the Hill** - Until the end of the CTF we are going to perform a competition to find the best agent in a 'king of the hill' fashion, meaning that the current best agent will be obfuscated and published and the other competitors must beat this agent to be the best agent, for more details PM one of the staff members, the best team at the end of the competition will receive the flag for the challenge.

The Desicion Maker

The code is comprised of 3 main components:

1. The environment - which simulates the actual battle, moves the pieces, declares the winner, counts the steps, etc.
2. the entity - which represents the agents physical properties (basically, its position)
3. The decision-maker - which is the "brains" of the entity - and decides on its actions at each step.

The decision maker is a module which given a state of a game, decides on the best course of action for its agent.
The state is just a class representing my position, and the enemy's position:

```
class State(object):
    def __init__(self, my_pos: Position, enemy_pos: Position):

        self.my_pos = my_pos
        self.enemy_pos = enemy_pos
```

The AgentAction is just a class containing all possible moves by the agent:

```
class AgentAction(IntEnum):

    TopRight = 1
    Right = 2
    BottomRight = 3
    Bottom = 4
    Stay = 5
    Top = 6
    BottomLeft = 7
    Left = 8
    TopLeft = 9
```

The competing teams must supply the environment with a decision maker.
It is an abstract class with 3 methods that must be implemented:

`update_context()` which receives a new state from the world, the integer reward associated with that state, and a boolean variable
indicating whether this state is a terminal state.

`get_action()` which receives the current state and returns the best course of action for the agent.
`set_initial_state()` used to initialize the decision maker with the state at the first round.

```
class AbsDecisionMaker(metaclass=abc.ABCMeta):

    def update_context(self, new_state: State, reward, is_terminal):
        pass
    def get_action(self, state: State)-> AgentAction:
        pass
    def set_initial_state(self, state: State):
        pass
```

The Environment

The environment is comprised of a digital surface map (or DSM) which is a `15 x 15` black matrix, with gray squares indicating the position of buildings.
Buildings are obstacles that one cannot pass through or see through.
Note that one may move diagonally to a gray square without the square interfering in the move.