

Frank's Zoo (Python)

Instructions

Frank's Zoo

Frank's Zoo is a fairly simple card game for four to six players. It uses a deck of 60 cards, each depicting an animal. For each animal, the card also identifies its predators. The cards are listed in Table 1.

Animal	#cards	Predators	ID
Whale	5	None	12
Elephant	5	Mouse	11
Crocodile	5	Elephant	10
Polar Bear	5	Whale, Elephant	9
Lion	5	Elephant	8
Seal	5	Whale, Polar Bear	7
Fox	5	Elephant, Crocodile, Polar Bear, Lion	6
Perch	5	Whale, Crocodile, Polar Bear, Seal	5
Hedgehog	5	Fox	4
Fish	5	Whale, Crocodile, Seal, Perch	3
Mouse	5	Crocodile, Polar Bear, Lion, Seal, Fox, Hedgehog	2
Mosquito	4	Hedgehog, Fish, Mouse	1
Chameleon	1	special (Joker)	0

Table 1: Cards in *Frank's Zoo*.

Cards are dealt evenly among the players. The player left of the dealer opens, and players get turns going clockwise around the table. When a player gets a turn, they can do one of the following:

- Pass, i.e., skip a turn.
- Move: play one or more cards to defeat the last move that is on the table.
- When opening: play any legal move (there is no move yet to defeat).

When, after a move of one player, all other players pass, the player who played the last move gets to open anew. If this player has already emptied their hand, the opening move goes to the player to their left. The goal of the game is to empty one's hand as quickly as possible.

A legal move is one that consists of cards of one type of animal. This should be:

- Either: A predator for the animal of the previous move. The number of cards should be exactly the same number as there were in the previous move;
- Or: The same animal as the previous move. The number of cards should be exactly one more than the number of cards that were in the previous move.

The Chameleon can be used to replace one of the cards in a move, as long as the move contains at least one "normal" animal card. Furthermore, in a move that contains one or more Elephants, at most one Mosquito can be added, which will then "turn into an Elephant". Note that when you play two cards, namely a Mosquito and the Chameleon, the Chameleon will be a Mosquito. If you want the Chameleon to be

an Elephant, there must be at least one Elephant in the play. There are some other optional rules in the published game, but these have not been implemented.

Examples:

One Hedgehog gets defeated by:

- One Fox
- Two Hedgehogs
- One Hedgehog and one Chameleon

Three Crocodiles get defeated by:

- Three Elephants
- Two Elephants and one Chameleon
- Two Elephants and one Mosquito
- One Elephant, one Mosquito, and one Chameleon
- Theoretically, four Crocodiles would also defeat three Crocodiles, but due to the fact that there are only five Crocodiles total in the deck, that can never happen.

Two Polar Bears get defeated by:

- Two Elephants
- One Elephant and one Chameleon
- One Elephant and one Mosquito
- Two Whales
- One Whale and one Chameleon
- Three Bears
- Two Bears and one Chameleon

This move is not defeated by an Elephant plus a Whale, since a move cannot combine two different kinds of animals. The exception for the previous statement is that one Mosquito may be added to Elephants, as it then “turns into an Elephant.”

When a player empties their hand, they are out of the game. The first player who empties their hand gets a number of points equal to the number of players. The second player to do so gets one point less, the third player again one point less, etc. When everyone but one player is out, the last player receives zero points. The game is usually repeated several times, and the players accumulate their points over the games, the player with the highest end total being the final winner.

Running a Competition

To run a competition, start the file `FranksZoo.py`. If you do that from the command line, you can give command line parameters:

```
python -u FranksZoo.py 1000 aifile1 aifile2 ...
```

The first command line parameter (1000 in the example above) is the number of games that should be played in the competition. After that you can specify a list of python files that contain AIs that you want to enter into the competition. If you do not specify any command line parameters, the number of games that will be played is 100, and the AIs are loaded from the files specified in the list `DEFAULTAIS` (defined in `FranksZoo.py`).

The number of players per game is specified in `FranksZoo.py` in the parameter `MAXPLAYERS`. If there are more AIs in the competition than `MAXPLAYERS`, every turn a random selection of four AIs will be made. This entails that not every AI may play the same number of times, and configurations of the selected players (e.g., who plays first) will also not be consistent. Of course, the shuffling of the deck will also result in variations. Statistically, if the number of games is large enough, the best AI of the pool should rise to the top anyway.

If you want to run the competition from an editor, you can (temporarily) change the values for `DEFAULTLENGTH` and `DEFAULTAIS` at the top of the `FranksZoo.py` file. `DEFAULTLENGTH` indicates how many games you want to run, and `DEFAULTAIS` is a list of all the AI files in the competition. If you change it, for instance, to `['FranksZooBeginners', 'MyAI']`, it will load all the AIs from `FranksZooBeginners.py` and from `MyAI.py`.

You can also simply run the `FranksZoo.py` file with an empty list for `DEFAULTAIS`. In that case, the engine will scan the current folder for all Python files which contain AIs, and will include those AIs in the competition. This means that you should only have Python files with AIs that you want to include in the current folder; move all other AIs to a different folder. In particular, you probably do not want to include the “self-play” AI.

At the end of the competition, the game prints an ordered list of all the AIs that entered, with a number that indicates in how many games they were, and their average score (a floating-point value between 0 and 4). The higher the average score, the better it is. It will also show the total number of seconds that the AI used, and the number of seconds per game.

Classes

Before I discuss how to write an AI for Frank’s Zoo in Python, I want to bring up the most important Python classes that you will use. These classes are found in `FranksZooGame.py` and `FranksZooState.py`.

A card of the game is represented by the class `Animal`. Instances of this class have the following attributes: `name`, `id`, `predator`, `prey`, `transform`, `substitute`, `ordering`, and `timeused`. `name` is the animal’s name, but the best way to refer to an animal is by `id`, which is a number between 0 and 12 (the IDs are included in Table 1), for which constants are defined in `FranksZooGame.py`. `predator` is a list of the animals which are predators for the animal, and `prey` is a list of animals which are preys of the animal. `transform` is a list of the animals into which the animal can transform (a Mosquito can transform into an Elephant), and `substitute` is a list of animals that can be substitutes for the animal. `ordering` is used to order the animals in a hand of cards. `timeused` is used by the engine to keep track of the time used by the AI.

A set of cards is represented by a `Hand`. Instances of `Hand` have only one attribute, which is a list of cards (`Animals`) in the attribute `cards`.

The class `Play` inherits from `Hand`. This is the class that the engine will use to communicate with AIs. A `Play` is a set of cards in the attribute `cards` (from `Hand`), and a few more attributes, namely the following: the name of the player who made this play in the attribute `player`, and two booleans, `opening` and `out`, which indicate whether it was an opening play, and whether it was a play which made the player empty their hand. When an AI creates a `Play` object, only the set of cards should be filled in. The other attributes are filled in by the main program.

In the file `FranksZooState.py` a `State` is defined. This is used to communicate to the AI the current state of the game. It contains an attribute `history`, which is a list of all the plays (as `Play` objects) made in the game until now. It also contains a list `players`, which are instances of the class `PlayerState`. The players are ordered in the sequence that they take turn, starting with the player who got to start the game. For each player there is an attribute `name` which contains the player name, an attribute `score` which contains the average score the player achieved over all the games played until now, and an attribute `handsize` which contains how many cards the player still has in their hand now.

The last important class that you should be aware of is `Player`, which is defined in `FranksZooPlayer.py`. A `Player` is an AI which plays the game.

How to write an AI

To write an AI, create a new Python file, and create a new class in it. This class should inherit its features from `Player`, which is defined in `FranksZooPlayer.py`. You should also create an `__init__()` method that calls the `__init__()` method from `Player`, and uses the `setName()` method to give itself a name. So, suppose that your new AI is called `MyAI`, then you will write in your file (probably called `MyAI.py`) the following code:

```
from FranksZooPlayer import Player

class MyAI(Player):
    def __init__( self ):
        Player.__init__( self )
        self.setName( 'MyAI' )
```

What you now have to do is create at least one method, namely `play().play()` is used to take a turn. It should return a valid play in the form of a `Play` object.

`play()` is defined as follows:

```
def play( self, lastplay, possible, state ):
```

`lastplay` is an object of the class `Play`, which represents the `Play` to which the AI must answer. If the AI must make an opening play, `lastplay` is `None`.

`possible` is a list of `Play` objects which are all the possible valid plays that the `play()` method can return. The first is always representing “pass,” which is an empty list of cards, except when this is an opening play, in which case “pass” is not

allowed – the only exception is that there would be only one card in the hand of the player, namely the Chameleon, as a Chameleon cannot be played on its own. In that case, and that case alone, a player may “pass” on an opening.

`state` represents the current state of the game as a `State` object.

As for the return value of `play()`: `play()` must return a valid `Play` object, i.e., one of the `Plays` in the list `possible`. This can be done by returning one of the objects in `possible`, or creating a `Play` object in the method itself. If you create your own `Play` object, you only should fill in the cards. The other attributes in the `Play` object will be filled in by the main engine. When you return a `Play` object, you do not need to remove the cards from your hand – the main engine will do that. You only need to say “this is what I want to play,” and the rest will be handled automatically.

If you return a `Play` object which is not valid, the engine makes a random play for you. This will also print an error message.

`play()` is called any time a player must take a turn. Even when there is only one possible play that can be made (i.e., `possible` contains only one object), it is still called.

Note that the method `play()` is not allowed to make changes to the arguments passed to the method. These are just meant as information. They are used by other methods as well. Naturally, I could avoid the potential problem of an AI making changes to these arguments by making copies of them before passing them to the `play()` method, but such copying tends to be rather time-intensive, and I don’t want to slow down the process. Besides, if the programmer really wants they can interfere with any object in the program, but that would be cheating, and cheating is not allowed anyway.

There are two more methods that can be used. The `startgame()` method is called at the start of the game, and the `endgame()` method at the end of the game. `startgame()` can be used to make some initializations, while `endgame()` can be used to track statistics. `endgame()` gets the current game state as argument.

Cheating

If you want to cheat in this game, of course that is possible. This is Python. Everything is visible, everything is accessible. Cheating is, in fact, so easy that I have not taken the trouble of trying to prevent it in code. If you use this code for an assignment, you should instruct students that cheating will disqualify them.

What is your AI allowed to do? You may look at the arguments given to the `play()` method. You may not make changes to them. Other than making changes to attributes that you create for your own AI, you are only allowed to effectuate changes by returning plays from the `play()` method. Other changes are not allowed. You are not allowed to look at information that is secret to you during the game, i.e., the cards that the other players have in hand.

Testing

To test the capabilities of your AI, you will probably first try it out against the AIs in `FranksZooBeginners.py`. Know, however, that these AIs are rather weak. You should be able to create an AI that plays a better game.

If you want to play the game yourself against AIs, include the AI that I placed in `FranksZooSelfPlay.py`. Make sure you set the number of games to 1.