TDT4113 (PLAB2) Project 2: Rock Scissors Paper

This document describes Task 2 in PLAB 2, “Rock, Scissors, Paper”.

- The task must be solved individually
- The task should be solved with object-oriented code written in Python.
- The deadline for the assignment is 2 weeks.

Your implementation will be uploaded to Blackboard no later than Wednesday, February 03, 2021 at 8:00 am and is demonstrated at 8-10 and 12-16 on the same day.

1 About the game “Rock, Scissors, Paper”

In the version of the game there are 2 players, Player 1 and Player 2. Each player can choose one of three actions: “rock”, “scissors” or “paper”. In the first phase of the game, players make their choices without revealing them to each other. In the second phase, the two players simultaneously show what they have chosen, and a winner is determined. If Player 1 and Player 2 have chosen the same, it is a draw if the winner is not selected based on the following rules (see also Fig. 1):

- Rock beats scissors.
- Scissors beats paper.
- Paper beats rock.

2 Implementation

The implementation you create should be written in object-oriented Python. You need to implement

- some semi-smart players for “Rock, Scissors, Paper”
- and a test environment to try them out.

It may also be useful to develop some help classes to make implementation as easy as possible possible. We discuss these one by one below. In order to get a complete overview of the task it is advisable to read the entire document before you begin the implementation.

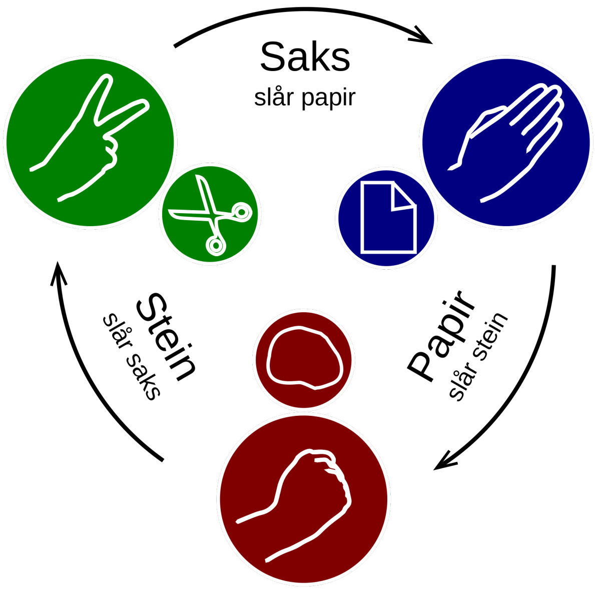

Figure 1: How to determine winner in Rock, Scissors, Paper.

2.1 The players

It is recommended to create a base class for players. We call it `Player` here. `Player` should (at least) have the following methods:

- `select_action`: This method selects which action to perform (play rock, scissors or paper) and return this.
- `receive_result`: After a single game is over, the player will know what was chosen by both players and who won. It can choose to learn from this information (see especially the players `Historian` and `MostCommon` described below).
- `enter_name`: This method specifies the name of the class so that we can report this in interface.

We will create a variety of players with different playing strategies. Their minimum requirements are given below:

- `Random`: This kind of players randomly select whether to make rock, scissors, or paper. To implement this, for example, it may be okay to use `random.randint(0, 2)`. To obtain this method, you must first import `random`.
- `Sequential`: This kind of players sequentially go through the various actions, playing rock, scissors, paper, rock, scissors, paper, rock, scissors, . . . in a fixed sequence no matter how the opponent behaves.
- `MostCommon`: This kind of players look at the opponent's choices over time and counts how many times opponents have played rock, scissors and paper. Then it assumes that the opponent will again play what he or she has played most often so far, and a `MostCommon` player therefore chooses optimally based on this. In the first game, where `MostCommon` has not seen any of the opponents' choices and thus does not know what to choose and so it chooses randomly.

Example: Suppose opponents played this sequence: rock, scissors, rock, rock, paper, scissors, paper, rock, paper, rock, rock, scissors, paper, scissors. We see that rock is most common in the history of the opponent, so most usually assume that rock will come again. The most common feature of the paper is therefore paper (as paper beats rock).

- `Historian`: This kind of players look for patterns in the way the opponent plays. `Historian` is defined by a parameter "`remember`". The description starts with looking at the situation `remember = 1`.

  Example: Let's again assume that opponents have played the sequence rock, scissors, rock, rock, paper, scissors, paper, rock, paper, rock, rock, scissors, paper, scissors. `Historian` looks at the last choice (scissors) and goes back in history to find what the opponent usually plays after a scissors. The opponent has played scissors three times before; two of these were followed by paper, while one was followed by rock. `Historian` thus thinks that the most common thing after scissors is paper, and therefore assumes that the next action of the opponent will be paper. `Historian` thus chooses scissors (because scissors win over paper).

  When `remember` is greater than 1, `Historian` looks for the sub-sequence to consist of the last `remember` actions instead of just looking for the latest when deciding what to play. If `remember=2`, it means that `Historian` first checks which 2 actions were played recently. In this case it was (paper, scissors), and `Historian` will look for this sub-sequence in history. The combination was played only once earlier in that order, and then the opponent chose to continue with paper. `Historian (remember=2)` thus assumes that there is now paper from opponent after (paper, scissors), and therefore selects scissors as it own action.

If we look at the same sequence but with `remember=3`, we must look for the sub-sequence (scissors, paper, scissors) in the story. This sequence has not been played before, and in such cases the Historian should select randomly.

2.2 Help-classes

It can be useful to define an `Action` class, which represents the action selected ("Playing Rock", "Playing Scissors", or "Playing Paper"). The point of this class is to define if one `action` wins over another. This is done by defining `__eq__` and `__gt__`; see the introduction video on object-oriented Python, especially on **operator overloading**.

The `SingleGame` class can be used to conduct and represent a single game. `SingleGame` is used to query players' choices, to find who has won, to report back to players, and to report to the console. To do so, the class needs these methods:

- `__init__(self, player1, player2)`: Initiate an instance of the class, where player1 and player2 are the two players.
- `perform_game`: Ask each player about their choice. Determine the result based on the rule that one point is awarded to the winner and zero to the loser (in the draw, both players get half point). Report the choices and results back to the players.
- `show_result`: textual reporting of the single game: What was chosen and who won?

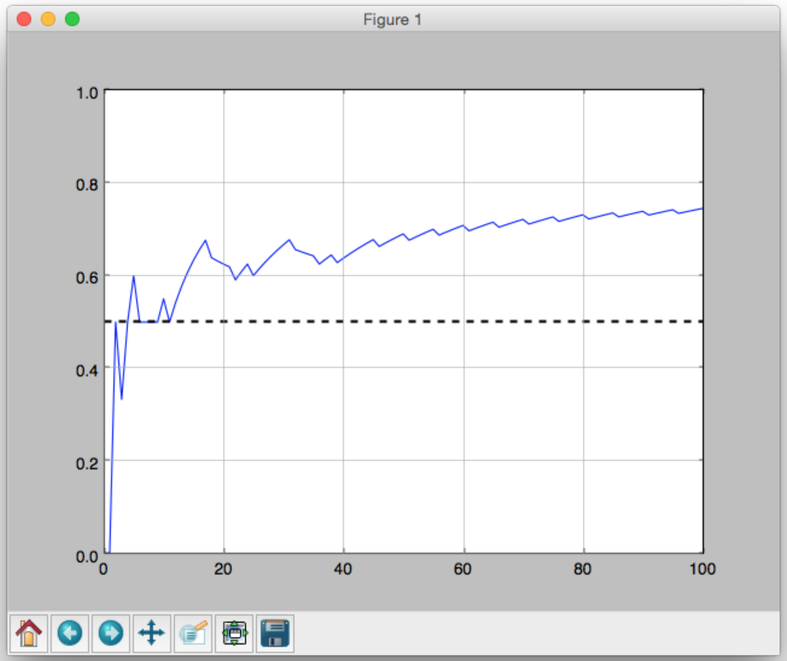

Figure 2: The evolution of the average number of points per game for `Historian(2)` against `MostCommon`. The tournament goes over 100 games and shows that `Historian(2)` gets a good idea of what it should do pretty quickly.

2.3 Text-based tournament to test two automatic players

To test our players, you must implement a test environment, here defined in the `Tournament` class. The class must have functionality at least with the following methods:

- `__init__(self, player1, player2, number_of_games)`: Set up the instance with the two players and remember to play the `number_of_games` times against each other.
- `arrange_singlegame`: To arrange a single game, the system must ask both players what action they choose, check who won, report the choices and results back to the players, and provide a textual description of the result, for example in the form of a single line:

  ```
  Historian (2):  Stein.  Most Common:  Scissors -> Historian (2) wins
  ```

  Note that much of this functionality is in the `SingleGame` class, so `Tournament.arrange_singlegame` can get most of its functionality from there.
- `arrange_tournament`: Complete the `number_of_games` single games between the two players. Report the win percentage for each of them when the tournament is complete. It is also interesting to show how the win percentage develops over time (see Fig. 2 which shows the score for `Historian(2)` against `MostCommon` over 100 games). This is most easily done by importing `matplotlib.pyplot` and using its plot method.

3 Criteria to pass the task

To pass this task you must:

- Solve it alone by the deadline - which is in 2 weeks.
- Code the 4 `Player` classes `Sequential`, `Random`, `MostCommon` and `Historian`.
- Implement text-based interface and use it to conduct tournaments.
- Implement graphics as in Figure 2 to see players' ability to learn over time.
- Your code should achieve Pylint score 8.0 or higher.

Extra challenge - not mandatory: Implement your own idea for a good Player. To be "good", your player should achieve at least 60% score against `Histrorian(2)` over 1000 rounds and at least 95% score against `Sequential`.

— o —