**Documentation of Red-Blue Nim Game**

**(Coded by M.Essa using Python)**

Introduction

The Red-Blue Nim Game is a two-player strategy game where players take turns removing marbles from two piles, one red and one blue. The objective is to be the last player to take a marble.

There are two main versions of the game:

**Standard Nim:** The player who takes the last marble wins the game.

**Misère Nim:** The player who takes the last marble loses the game.

**Objectives of Implementing the Game in Python:**

**Practice Python programming:** Implementing the game can be a great way to practice various Python concepts, such as:

Data structures (lists, tuples)

Functions and control flow

Algorithms (Minimax, Alpha-Beta Pruning)

Object-oriented programming

**Learn game theory and AI:** Developing a game-playing AI can help you understand game theory and AI concepts like decision-making, search algorithms, and heuristic functions.

**Create a fun and challenging game:** The Red-Blue Nim Game can be a fun and challenging game to play, and implementing it in Python can provide a satisfying project.

**Experiment with different game variations:** You can experiment with different game variations, such as changing the number of piles or the rules for winning.

Game Rules

**2.1 Standard Version**

**Objective:** The players lose if either pile is empty on their turn.

**Gameplay:** Players take turns, presumably removing marbles from one or both piles. If it's a player's turn and one of the piles is empty, they lose.

**2.2 Misère Version**

**Objective:** The players win if either pile is empty on their turn.

**Gameplay:** Unlike the standard version, the goal here is to force the other player into having no options by making one of the piles empty. If you make a pile empty on your turn, you win.

**2.3 Scoring**

**Points System:** Each red marble left in the piles after the game ends is worth 2 points.

Each blue marble left is worth 3 points.

Command Line

**Usage:**

To start the game using the command-line, you'll need to run the Python script from your terminal or command prompt. The general syntax is:

PowerShell

python game.py [options]

python game**.**py **-**h

usage**:** game**.**py **[-**h**]** **[-**r **]** **[-**b**]** **[-**v**]** **[-**f**]** **[-**d**]**

Red-Blue Nim Game Coded by M**.**Essa

optional arguments**:**

**-**h**,** **--**help show this help message and **exit**

**-**r **,** **--**num-red Number of red marbles **(**default**:** 10**)**

**-**b **,** **--**num-blue Number of blue marbles **(**default**:** 8**)**

**-**v **,** **--**version Game version to play standard or misere **(**default**:** standard**)**

**-**f **,** **--**first Who goes first computer or player **(**default**:** computer**)**

**-**d **,** **--**depth Depth of minimax search **(**default**:** 10**)**

**Example Invocation:**

To start a standard Nim game with 15 red marbles, 12 blue marbles, the computer as the first player, and a search depth of 8, you would use:

PowerShell

python game**.**py **-**r 15 **-**b 12 **-**v standard **-**f computer **-**d 8

**Parameters:**

**<num-red>:** Specifies the initial number of red marbles. If not provided, the default is 10.

**<num-blue>:** Specifies the initial number of blue marbles. If not provided, the default is 8.

**<version>:** Determines the game mode:

**standard:** The standard Nim game where the player who takes the last marble wins.

**misere:** The Misère Nim game where the player who takes the last marble loses.If not provided, the default is standard.

**<first-player>:** Specifies who goes first:

**computer:** The computer starts the game.

**human:** The human player starts the game.

If not provided, the default is computer.

**<depth>:** Sets the search depth for the AI's minimax algorithm. A higher depth generally improves the AI's decision-making but can also increase computation time. This parameter is optional. If not provided, the default is 10.

**Additional Notes:**

You can combine multiple parameters in a single command-line invocation. For example, to start a Misère Nim game with equal numbers of red and blue marbles, you could use:

PowerShell

python game.py -r 10 -b 10 -v misere

If you prefer to use the default values for all parameters, you can simply run the script without any arguments:

PowerShell

Python game**.**py

This will start a standard Nim game with 10 red marbles, 8 blue marbles, the computer as the first player, and a search depth of 10.

Game Flow

**4.1 Turn Order:**

The game alternates between the human player and the computer player until a terminal state is reached (i.e., when all red or blue marbles removed from one of the pile).

The First\_player object determines who makes the initial move.

After each player's turn, the change\_first() method is called to switch the turn order for the next round.

First\_Player class is implemented inside game\_class.py

Python

class First\_player:

def \_\_init\_\_(self, computer = False, human = True,human\_error = False):

self.*computer* = computer

self.*human* = human

self.*human\_error* = human\_error

self.*human\_error\_message* = None

def first\_computer(self):

return self.*computer*

def first\_human(self):

return self.*human*

def change\_first(self):

self.*computer* = not self.*computer*

self.*human* = not self.*human*

return None

def get\_human\_error(self):

self.*human\_error* = not self.*human\_error*

return None

def error(self,Error):

self.*human\_error\_message* = Error

return None

def error\_message(self):

return self.*human\_error\_message*

**4.2 Human Move**

The code prompts the human player for their move by printing a message indicating that it's their turn. The player is then asked to choose between Red and Blue, and after that the player is asked to input the number of marbles to take out (1 or 2).

**<<<<<<**Game Version is Standard**>>>>>>**

**<<If** You take the last Red or Blue Marble then You Win**>>**

Computer took out 0 Red Marbles and 2 Blue Marbles

10 Red Marbles and 6 Blue Marbles remaining

Red Marbles **\*\*\*\*\*\*\*\*\*\***

Blue Marbles **\*\*\*\*\*\***

Your Turn

Which Marbles you want to take out Red or Blue**:-** Red

Input the value of Red Marbles**,** you want to take out**:-** 2

The input is validated using Recursion:

* + The input()function is used to take input from the human.
  + If the input is not valid then *get\_human\_error***(self)** method is called.
  + *get\_human\_error***(self)** is flag for human error and it will not allow for computer to play until the error is resolved.
  + Another method is called when the input is not valid which is *error***(**Error**)** , this method is used to print the error on console.
  + Both the methods are in side the First\_player class which is implemented inside the game\_class.py.

**4.3 Computer Move**

* For computer the program uses the Version Class which is implemented inside the game\_class.py
* The Version Class ensures that which version is being played (standard or misere) and according to that it decide the next move for computer.
* The Version Class has two methods *update\_states***(**self**,**new\_state**)** and *get\_next\_move***(**self**)**
* *update\_states***(**self**,**new\_state**)** used to update the state (numer of red and blue marbles) each time when its computer turn.
* *get\_next\_move***(**self**)** is used to determines next move for the computer.

Python

class Version:

    def \_\_init\_\_(self, state=None,depth = 20,standard=True):

        self.state = state

        self.depth = depth

        self.standard = standard

    def update\_states(self,new\_state):

        self.state = new\_state

    def get\_next\_move(self):

        if self.standard:

            return mnimx.next\_move\_standard(self.state,self.depth)

        else:

            return mnimx.next\_move\_misere(self.state,self.depth)

* *get\_next\_move***(**self**)** has two function mnimx.next\_move\_standard(self.state,self.depth) and mnimx.next\_move\_misere(self.state,self.depth)
* The both functions are defined inside the minimax.py file
* Both the function makes every possible state and checks score of every state by using the MinMax algorithm.

Python

def next\_move\_standard(state,depth):

    next = possible\_new\_states(state)

    for i in next:

        score = minimax(i, True,depth)

        if score == -1:

            return -score, i

    return score, i

def next\_move\_misere(state,depth):

    if state == (3,1) or state == (1,3):

        return 1, (1,1)

    next = possible\_new\_states(state)

    for i in next:

        score = minimax(i, True,depth)

        if score == 1:

            return score, i

    return -score, i

* The Function possible\_new\_states(state) is used to generate every possible state.

Python

def possible\_new\_states(state):

    for pile, counters in enumerate(state):

        for decrement in (1, 2):

            if counters >= decrement:

                yield state[:pile] + (counters - decrement,) + state[pile + 1:]

* The score is decided by the minimax(state, max\_turn,depth,alpha = -1 , beta = 1) which use the Recursion and it goes to every endgame possible and checks wheather computer won (1) or lost(-1) in standard game which can be reversed in Misere.
* The function has also the depth, alpha and beta
* The depth is used for decrease the search size which make the compution much faster. When the value of depth becomes zero the the Recursion stop.
* The Alpha Beta Pruning is also used to eliminate branches of the game tree which optimize the search. The Alpha Beta Pruning is important because using the this algorithm program can find the best score before the looking every possible endgame, which the compution much faster.

Python

def minimax(state, max\_turn,depth,alpha = -1 , beta = 1):

    if depth == 0:

        return 1 if max\_turn else -1

    if (score := evaluate(state, max\_turn)) is not None:

        return score

    depth -= 1

    scores = []

    for new\_state in possible\_new\_states(state):

        scores.append(

            score := minimax(new\_state, not max\_turn,depth, alpha, beta)

        )

        if max\_turn:

            alpha = max(alpha, score)

        else:

            beta = min(beta, score)

        if beta <= alpha:

            break

    return (max if max\_turn else min)(scores)

def next\_move\_standard(state,depth):

    next = possible\_new\_states(state)

    for i in next:

        score = minimax(i, True,depth)

        if score == -1:

            return -score, i

    return score, i

* The MinMax and Alpha Beta Pruning is explained in the section in a much better way without coding.

MinMax Algorithm

**5.1 Overview**

The MinMax algorithm is a decision-making algorithm used in game theory and artificial intelligence. It is a recursive algorithm that explores the game tree to determine the optimal move for a player. The algorithm assumes that the opponent will always make the best possible move.

In the context of the Nim game, the MinMax algorithm is used to determine the computer's next move. The algorithm works by considering all possible moves from the current state and evaluating their potential outcomes. The computer chooses the move that leads to the highest estimated score for the current player.

**5.2 Alpha Beta Pruning**

Alpha Beta Pruning is a technique that can significantly improve the efficiency of the MinMax algorithm. It works by eliminating branches of the game tree that cannot possibly lead to the best outcome.

The algorithm maintains two values: alpha and beta. Alpha represents the best score that the maximizing player (the current player) can achieve, while beta represents the best score that the minimizing player (the opponent) can achieve.

If at any point during the search, beta becomes less than or equal to alpha, the current branch can be pruned because it cannot lead to a better outcome for the maximizing player.

**5.3 Move Ordering (Standard)**

The ordering was done inside the game\_version.py for the game\_1 and game\_2

Ordering was done according to standard. First the *get\_next\_move***(**self**)** used to get the next state the we subtract the new state to pervious state, by doing the this we can get the number red or blue marbles to take out form pile.

Python

new\_state = (state[0],state[1])

    comp.update\_states(new\_state)

    selected\_move = comp.get\_next\_move()

    selected\_move = selected\_move[1]

    state\_red = state[0] - selected\_move[0]

    state\_blue = state[1] - selected\_move[1]

    state[0] = state[0] - state\_red

    state[1] = state[1] - state\_blue

**5.4 Move Ordering (Misère)**

The ordering was done inside the game\_version.py for the game\_2 and game\_3

Ordering was done according to standard. First the *get\_next\_move***(**self**)** used to get the next state the we subtract the new state to pervious state, by doing the this we can get the number red or blue marbles to take out form pile.

Python

new\_state = (state[0],state[1])

    comp.update\_states(new\_state)

    selected\_move = comp.get\_next\_move()

    selected\_move = selected\_move[1]

    state\_red = state[0] - selected\_move[0]

    state\_blue = state[1] - selected\_move[1]

    state[0] = state[0] - state\_red

    state[1] = state[1] - state\_blue

Depth Limited Search (Extra Credit)

**6.1 Purpose**

Depth-Limited Search is a way to make a computer think faster when playing games. Instead of exploring every possible move, it sets a limit on how far ahead it looks. This helps the computer make decisions quickly, even in complex games.In the Nim game, using depth-limited search can help the computer choose its moves faster without sacrificing too much accuracy. By limiting how far it looks ahead, the computer can make decisions more efficiently.

Python

if depth == 0:

   return 1 if max\_turn else -1

**6.2 Evaluation Function**

To estimate the "goodness" of a non-terminal game state for the current player.

To provide a numerical value (1 or -1) that can be used by the Minimax algorithm to guide decision-making.

**Check for Terminal State:**

* If the state contains any empty piles (i.e., piles with 0 marbles), the game is considered over.
* In this case, the function returns -1 if it's the maximizing player's turn (indicating a loss) or 1 if it's the minimizing player's turn (indicating a win).

**Non-Terminal State:**

* If the game is not over (i.e., no piles are empty), the function returns 0.

python

def evaluate(state, max\_turn):

    if not all(state):

        return -1 if max\_turn else 1

End of Game

**7.1 Game Over Conditions**

**One of the pile become empty:** If a pile has no remaining marbles, the game is over.

**A player takes the last marble in a pile:** The player who takes the last marble in a pile either wins or loses the game, depending on the version being played:

* **Standard Version:** The player who takes the last marble wins.
* **Misère Version:** The player who takes the last marble loses.

Python

if state[0]\*state[1] == 0:

        if state[0] != 0:

            mesg.human\_won\_message("Red",state[0])

            return None

        else:

            mesg.human\_won\_message("Blue",state[1])

            return None

* human\_won\_message() method used to print the when human wins the game
* computer\_won\_message()method used to print the when human wins the game

Python

def human\_won\_message(self,color,score):

        clear()

        if color.lower() == 'red':

            print("""

                  ██╗  ██╗██╗   ██╗██████╗ ██████╗  █████╗ ██╗   ██╗██╗    ██╗   ██╗ ██████╗ ██╗   ██╗    ██╗    ██╗ ██████╗ ███╗   ██╗

                  ██║  ██║██║   ██║██╔══██╗██╔══██╗██╔══██╗╚██╗ ██╔╝██║    ╚██╗ ██╔╝██╔═══██╗██║   ██║    ██║    ██║██╔═══██╗████╗  ██║

                  ███████║██║   ██║██████╔╝██████╔╝███████║ ╚████╔╝ ██║     ╚████╔╝ ██║   ██║██║   ██║    ██║ █╗ ██║██║   ██║██╔██╗ ██║

                  ██╔══██║██║   ██║██╔══██╗██╔══██╗██╔══██║  ╚██╔╝  ╚═╝      ╚██╔╝  ██║   ██║██║   ██║    ██║███╗██║██║   ██║██║╚██╗██║

                  ██║  ██║╚██████╔╝██║  ██║██║  ██║██║  ██║   ██║   ██╗       ██║   ╚██████╔╝╚██████╔╝    ╚███╔███╔╝╚██████╔╝██║ ╚████║

                  ╚═╝  ╚═╝ ╚═════╝ ╚═╝  ╚═╝╚═╝  ╚═╝╚═╝  ╚═╝   ╚═╝   ╚═╝       ╚═╝    ╚═════╝  ╚═════╝      ╚══╝╚══╝  ╚═════╝ ╚═╝  ╚═══╝

                                                                                                                     """)

            print(f'<<<Final Score is {2\*score}>>>>>>')

        elif color.lower() == 'blue':

            print("""

                  ██╗  ██╗██╗   ██╗██████╗ ██████╗  █████╗ ██╗   ██╗██╗    ██╗   ██╗ ██████╗ ██╗   ██╗    ██╗    ██╗ ██████╗ ███╗   ██╗

                  ██║  ██║██║   ██║██╔══██╗██╔══██╗██╔══██╗╚██╗ ██╔╝██║    ╚██╗ ██╔╝██╔═══██╗██║   ██║    ██║    ██║██╔═══██╗████╗  ██║

                  ███████║██║   ██║██████╔╝██████╔╝███████║ ╚████╔╝ ██║     ╚████╔╝ ██║   ██║██║   ██║    ██║ █╗ ██║██║   ██║██╔██╗ ██║

                  ██╔══██║██║   ██║██╔══██╗██╔══██╗██╔══██║  ╚██╔╝  ╚═╝      ╚██╔╝  ██║   ██║██║   ██║    ██║███╗██║██║   ██║██║╚██╗██║

                  ██║  ██║╚██████╔╝██║  ██║██║  ██║██║  ██║   ██║   ██╗       ██║   ╚██████╔╝╚██████╔╝    ╚███╔███╔╝╚██████╔╝██║ ╚████║

                  ╚═╝  ╚═╝ ╚═════╝ ╚═╝  ╚═╝╚═╝  ╚═╝╚═╝  ╚═╝   ╚═╝   ╚═╝       ╚═╝    ╚═════╝  ╚═════╝      ╚══╝╚══╝  ╚═════╝ ╚═╝  ╚═══╝

                                                                                                                     """)

            print(f'<<<Final Score is {3\*score}>>>>>>')

        return None

**7.2 Scoring Calculation**

**Each red marble left:** 2 points

**Each blue marble left:** 3 points

* If human\_won\_message() or computer\_won\_message()called then it takes the value of the remaining marbles and it will print it on console
* These method check wheather the remaining marbles are red or blue.
* And the color of marbles , the method multiply the remaining marbles to the points which has been assigned for each color.

**For Remaining Red Marbles**

Python

def human\_won\_message(self,color,score):

if color.lower() == 'red':

print(f'<<<<<Final Score is {2\*score}>>>>>>')

**For Remaining Blue Marbles**

Python

def human\_won\_message(self,color,score):

if color.lower() == 'blue':

print(f'<<<<<<Final Score is {3\*score}>>>>>>')

Implementation Details

**Command-line parsing**

The Command-line parsing is implemented inside the game.py

It can take different parameters from the user .

Python

parser = argparse.ArgumentParser(description='Red-Blue Nim Game Coded by M.Essa')

parser.add\_argument('-r','--num-red', type=int, default=10,metavar=' ',help='Number of red marbles (default: 10)')

parser.add\_argument('-b','--num-blue', type=int, default=8, metavar='',help='Number of blue marbles (default: 8)')

parser.add\_argument('-v','--version', choices=['standard','misere'], default='standard',metavar='', help='Game version to play standard or misere (default: standard)')

parser.add\_argument('-f','--first', choices=['player','computer'], default='computer',metavar='', help='Who goes first computer or player (default: computer)')

parser.add\_argument('-d','--depth', type=int, default=10, metavar='',help='Depth of minimax search (default: 10)')

args = parser.parse\_args()

**Game mechanics**

The Game mechanic is implemented inside the game\_class.py which govern and guide the player's actions like printing error , printing the state of the game, switching the player, updating the state, switching the different Games modes

* **Printing error and State**

python

class message:

    def \_\_init\_\_(self,red,blue,player):

        self.red = red

        self.blue = blue

        self.player = player

    def get\_message\_state\_1(self):

        clear()

        print(f'<<<<<<Game Version is Standard>>>>>>')

        print(f'<<If You take the last  Red or Blue Marble then You Win>>\n')

        print(f'{self.red} Red Marbles and {self.blue} Blue Marbles remaining \n')

        print("Red Marbles " + self.red\*"\*")

        print("Blue Marbles " + self.blue\*"\*")

        print(f'\nYour Turn')

        return None

    def get\_message\_state\_2(self,state\_red,state\_blue):

        clear()

        if self.player.human\_error:

            print(f'<<<<<<Game Version is Standard>>>>>>')

            print(f'<<If You take the last  Red or Blue Marble then You Win>>\n')

            print(f'{self.red} Red Marbles and {self.blue} Blue Marbles remaining \n')

            print("Red Marbles " + self.red\*"\*")

            print("Blue Marbles " + self.blue\*"\*")

            print(f'Type Error ---> ({self.player.error\_message()})\n')

            print(f'\nYour Turn')

            self.player.get\_human\_error()

        else:

            print(f'<<<<<<Game Version is Standard>>>>>>')

            print(f'<<If You take the last  Red or Blue Marble then You Win>>\n')

            print(f'Computer took out {state\_red} Red Marbles and {state\_blue} Blue Marbles')

            print(f'{self.red} Red Marbles and {self.blue} Blue Marbles remaining \n')

            print("Red Marbles " + self.red\*"\*")

            print("Blue Marbles " + self.blue\*"\*")

            print(f'\nYour Turn')

        return None

    def get\_message\_state\_3(self):

        clear()

        print(f'<<<<<<Game Version is Misere>>>>>>')

        print(f'<<If You take the last  Red or Blue Marble then You Lose>>\n')

        print(f'{self.red} Red Marbles and {self.blue} Blue Marbles remaining \n')

        print("Red Marbles " + self.red\*"\*")

        print("Blue Marbles " + self.blue\*"\*")

        print(f'\nYour Turn')

        return None

    def get\_message\_state\_4(self,state\_red,state\_blue):

        clear()

        if self.player.human\_error:

            print(f'<<<<<<Game Version is Misere>>>>>>')

            print(f'<<If You take the last  Red or Blue Marble then You Lose>>\n')

            print(f'{self.red} Red Marbles and {self.blue} Blue Marbles remaining \n')

            print("Red Marbles " + self.red\*"\*")

            print("Blue Marbles " + self.blue\*"\*")

            print(f'Type Error ---> ({self.player.error\_message()})\n')

            print(f'\nYour Turn')

            self.player.get\_human\_error()

        else:

            print(f'<<<<<<Game Version is Misere>>>>>>')

            print(f'<<If You take the last  Red or Blue Marble then You Lose>>\n')

            print(f'Computer took out {state\_red} Red Marbles and {state\_blue} Blue Marbles')

            print(f'{self.red} Red Marbles and {self.blue} Blue Marbles remaining \n')

            print("Red Marbles " + self.red\*"\*")

            print("Blue Marbles " + self.blue\*"\*")

            print(f'\nYour Turn')

        return None

* **switching the player**

python

class First\_player:

    def \_\_init\_\_(self, computer = False, human = True,human\_error = False):

        self.computer = computer

        self.human = human

        self.human\_error = human\_error

        self.human\_error\_message = None

    def first\_computer(self):

        return self.computer

    def first\_human(self):

        return self.human

    def change\_first(self):

        self.computer = not self.computer

        self.human = not self.human

        return None

    def get\_human\_error(self):

        self.human\_error = not self.human\_error

        return None

    def error(self,Error):

        self.human\_error\_message = Error

        return None

    def error\_message(self):

        return self.human\_error\_message

* **Updating the state**

class Version:

    def \_\_init\_\_(self, state=None,depth = 20,standard=True):

        self.state = state

        self.depth = depth

        self.standard = standard

    def update\_states(self,new\_state):

        self.state = new\_state

    def get\_next\_move(self):

        if self.standard:

            return mnimx.next\_move\_standard(self.state,self.depth)

        else:

            return mnimx.next\_move\_misere(self.state,self.depth)

* **Switching the different Games modes**

class Game:

    def \_\_init\_\_(self,state,computer,player):

        self.state = state

        self.computer = computer

        self.player = player

        self.state\_red = 0

        self.state\_blue = 0

    def start\_game(self):

        if self.computer.standard:

            if self.player.human:

                gv.game\_1(self.state,self.computer,self.player,self.state\_red,self.state\_blue)

            else:

                gv.game\_2(self.state,self.computer,self.player,self.state\_red,self.state\_blue)

        else:

            if self.player.human:

                gv.game\_3(self.state,self.computer,self.player,self.state\_red,self.state\_blue)

            else:

                gv.game\_4(self.state,self.computer,self.player,self.state\_red,self.state\_blue)

**Human and computer moves**

The Human and computer moves are implemented inside the game\_version.py

* **Human move**

if player.first\_human():

            mesg.get\_message\_state\_1()

            player.change\_first()

    else:

            mesg.get\_message\_state\_2(state\_red,state\_blue)

    red\_blue = input(f'Which Marbles you want to take out Red or Blue:- ')

    if red\_blue.lower() == 'red':

        take = input(f'Input the value of Red Marbles, you want to take out:- ')

        if take.isdigit() == False:

            player.get\_human\_error()

            player.error('Please input a number')

            return game\_1(state,comp,player,state\_red,state\_blue)

        take = int(take)

        if abs(take) > state[0]:

            player.get\_human\_error()

            player.error(f'Sorry in Red only {state[0]} Marbles are left')

            return game\_1(state,comp,player,state\_red,state\_blue)

        if take > 2 or take < 1:

            player.get\_human\_error()

            player.error('Sorry you can only take out 1 or 2 balls')

            return game\_1(state,comp,player,state\_red,state\_blue)

        state[0] = state[0] - take

    elif red\_blue.lower() == 'blue':

        take = input(f'Input the value of Blue Marbles, you want to take out:- ')

        if take.isdigit() == False:

            player.get\_human\_error()

            player.error('Please input a number')

            return game\_1(state,comp,player,state\_red,state\_blue)

        take = int(take)

        if abs(take) > state[1]:

            player.get\_human\_error()

            player.error(f'Sorry in Blue only {state[1]} Marbles are left')

            return game\_1(state,comp,player,state\_red,state\_blue)

        if take > 2 or take < 1:

            player.get\_human\_error()

            player.error('Sorry you can only take out 1 or 2 balls')

            return game\_1(state,comp,player,state\_red,state\_blue)

        state[1] = state[1] - take

    else:

        player.get\_human\_error()

        player.error('Sorry you can only choose Red or Blue')

        return game\_1(state,comp,player,state\_red,state\_blue)

    if player.human\_error:

        player.get\_human\_error()

* **Computer move**

    new\_state = (state[0],state[1])

    comp.update\_states(new\_state)

    selected\_move = comp.get\_next\_move()

    selected\_move = selected\_move[1]

    state\_red = state[0] - selected\_move[0]

    state\_blue = state[1] - selected\_move[1]

    state[0] = state[0] - state\_red

    state[1] = state[1] - state\_blue

* **AI decision-making with MinMax and Alpha Beta Pruning**

The AI decision-making with MinMax and Alpha Beta Pruning is implemented inside the minimax.py

Python

def possible\_new\_states(state):

    for pile, counters in enumerate(state):

        for decrement in (1, 2):

            if counters >= decrement:

                yield state[:pile] + (counters - decrement,) + state[pile + 1:]

def evaluate(state, max\_turn):

    if not all(state):

        return -1 if max\_turn else 1

def minimax(state, max\_turn,depth,alpha = -1 , beta = 1):

    if depth == 0:

        return 1 if max\_turn else -1

    if (score := evaluate(state, max\_turn)) is not None:

        return score

    depth -= 1

    scores = []

    for new\_state in possible\_new\_states(state):

        scores.append(

            score := minimax(new\_state, not max\_turn,depth, alpha, beta)

        )

        if max\_turn:

            alpha = max(alpha, score)

        else:

            beta = min(beta, score)

        if beta <= alpha:

            break

    return (max if max\_turn else min)(scores)

def next\_move\_standard(state,depth):

    next = possible\_new\_states(state)

    for i in next:

        score = minimax(i, True,depth)

        if score == -1:

            return -score, i

    return score, i

def next\_move\_misere(state,depth):

    if state == (3,1) or state == (1,3):

        return 1, (1,1)

    next = possible\_new\_states(state)

    for i in next:

        score = minimax(i, True,depth)

        if score == 1:

            return score, i

    return -score, i

def main():

    #print(minimax((20,12),True,10))

    print(next\_move\_misere((20,21),10))

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Demonstration**

**1:- Starting the game using Command Line**

PS D:\Essa\_intership\week\_1> python game.py -h

usage: game.py [-h] [-r ] [-b] [-v] [-f] [-d]

Red-Blue Nim Game Coded by M.Essa

optional arguments:

-h, --help show this help message and exit

-r , --num-red Number of red marbles (default: 10)

-b , --num-blue Number of blue marbles (default: 8)

-v , --version Game version to play standard or misere (default: standard)

-f , --first Who goes first computer or player (default: computer)

-d , --depth Depth of minimax search (default: 10)

PS D:\Essa\_intership\week\_1> python game.py -r 5 -b 5 -v standard -f player -d 12

**2:- Program ask for the human input**

<<<<<<Game Version is Standard>>>>>>

<<If You take the last Red or Blue Marble then You Win>>

5 Red Marbles and 5 Blue Marbles remaining

Red Marbles \*\*\*\*\*

Blue Marbles \*\*\*\*\*

Your Turn

Which Marbles you want to take out Red or Blue:- blue

Input the value of Blue Marbles, you want to take out:- 2

**3:- Computer takes it’s move**

<<<<<<Game Version is Standard>>>>>>

<<If You take the last Red or Blue Marble then You Win>>

Computer took out 2 Red Marbles and 0 Blue Marbles

3 Red Marbles and 3 Blue Marbles remaining

Red Marbles \*\*\*

Blue Marbles \*\*\*

Your Turn

Which Marbles you want to take out Red or Blue:-

**4:- Program again ask for the human input**

<<<<<<Game Version is Standard>>>>>>

<<If You take the last Red or Blue Marble then You Win>>

Computer took out 2 Red Marbles and 0 Blue Marbles

3 Red Marbles and 3 Blue Marbles remaining

Red Marbles \*\*\*

Blue Marbles \*\*\*

Your Turn

Which Marbles you want to take out Red or Blue:- red

Input the value of Red Marbles, you want to take out:- 2

**5:- Conclusion ( Computer won in this case )**

██████╗ ██████╗ ███╗ ███╗██████╗ ██╗ ██╗████████╗███████╗██████╗ ██╗ ██╗ ██████╗ ███╗ ██╗

██╔════╝██╔═══██╗████╗ ████║██╔══██╗██║ ██║╚══██╔══╝██╔════╝██╔══██╗ ██║ ██║██╔═══██╗████╗ ██║

██║ ██║ ██║██╔████╔██║██████╔╝██║ ██║ ██║ █████╗ ██████╔╝ ██║ █╗ ██║██║ ██║██╔██╗ ██║

██║ ██║ ██║██║╚██╔╝██║██╔═══╝ ██║ ██║ ██║ ██╔══╝ ██╔══██╗ ██║███╗██║██║ ██║██║╚██╗██║

╚██████╗╚██████╔╝██║ ╚═╝ ██║██║ ╚██████╔╝ ██║ ███████╗██║ ██║ ╚███╔███╔╝╚██████╔╝██║ ╚████║

╚═════╝ ╚═════╝ ╚═╝ ╚═╝╚═╝ ╚═════╝ ╚═╝ ╚══════╝╚═╝ ╚═╝ ╚══╝╚══╝ ╚═════╝ ╚═╝ ╚═══╝

**<<<<<<**Final Score **is** 9**>>>>>>**

PS D**:**\Essa\_intership\week\_1**>**

**The End Of Documentation**