NAME:- RHYTHM SHAH UID:- 20BIT053

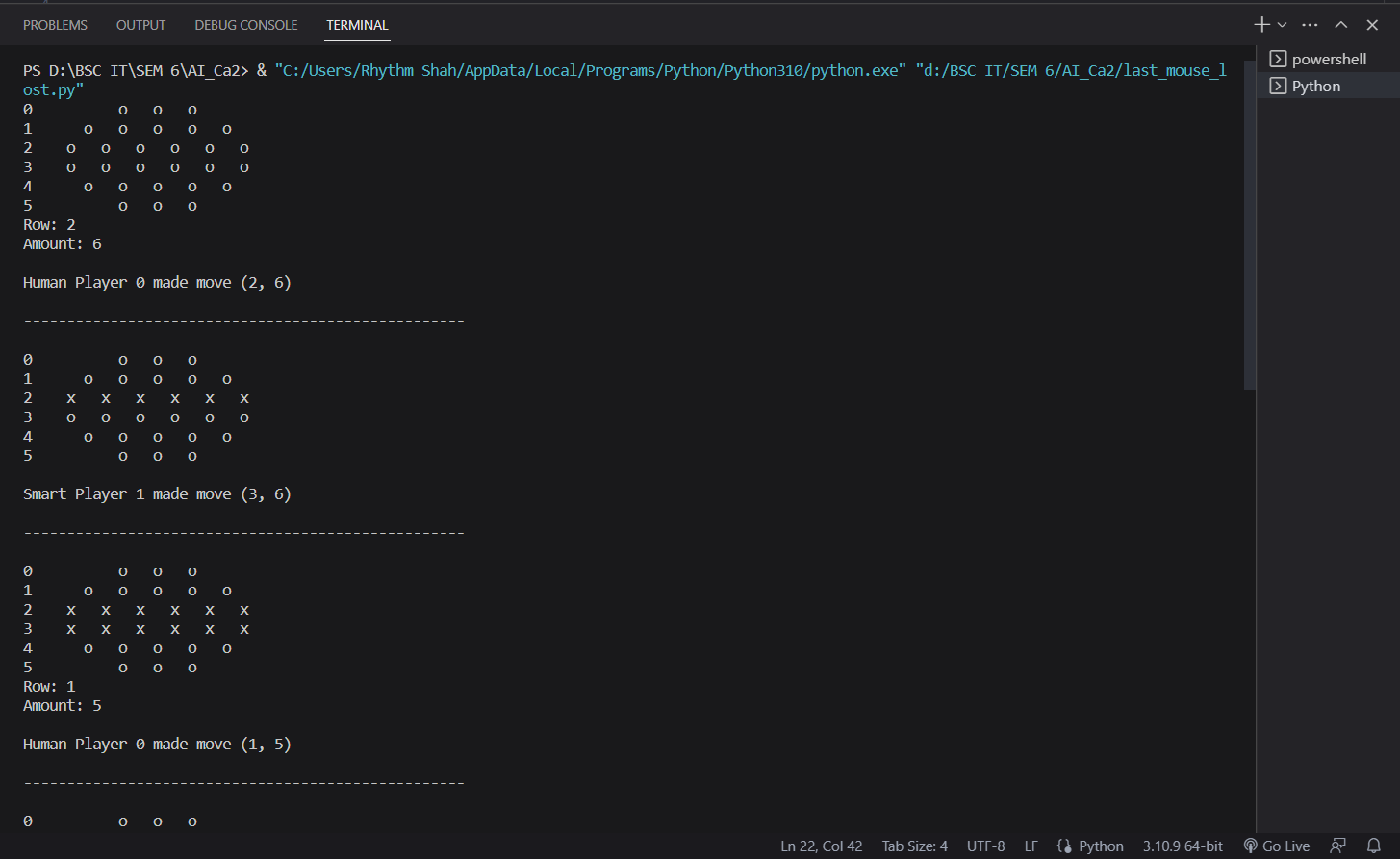
**O – Board Game**

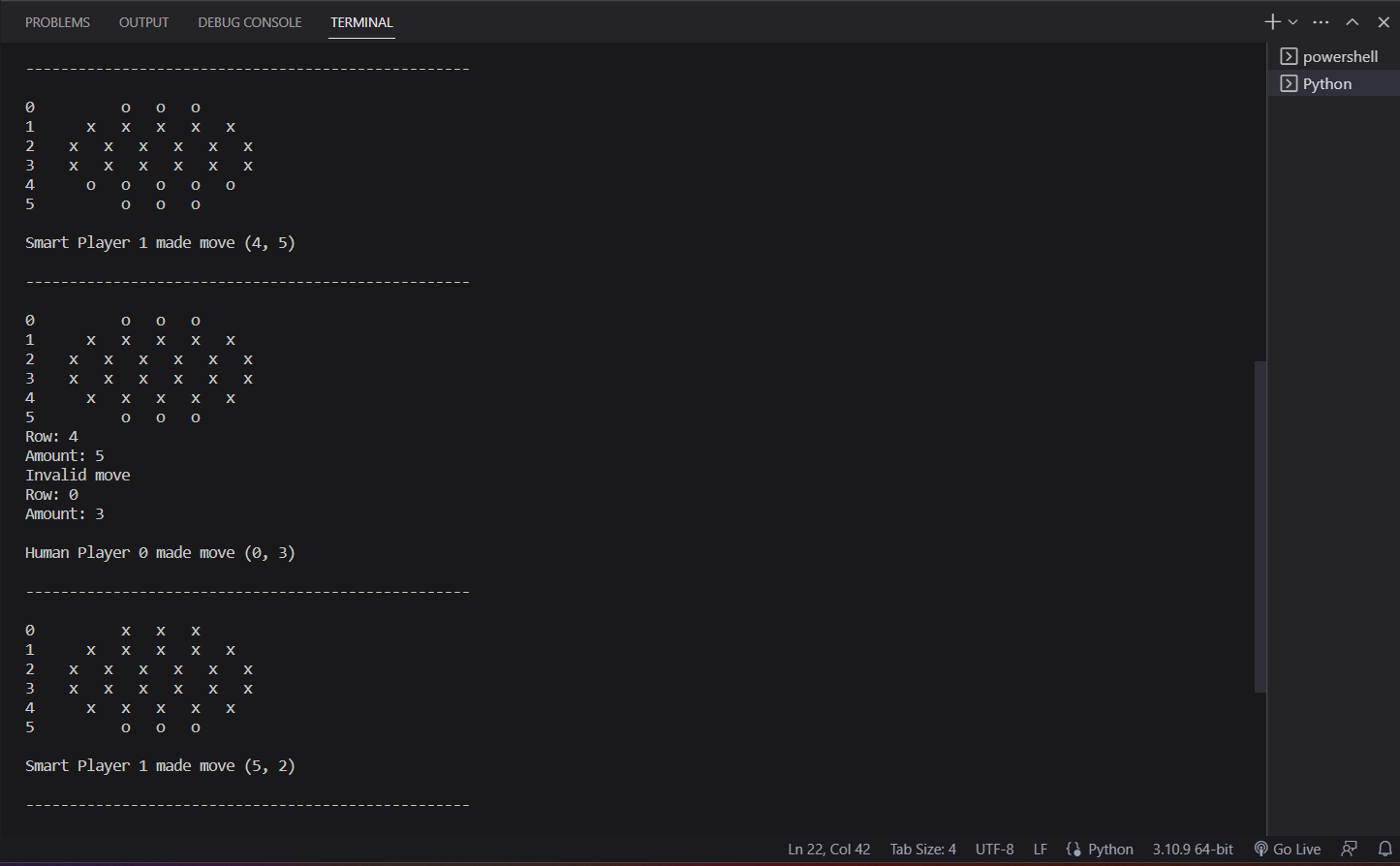
This game only works on one rule - Last to play loses!

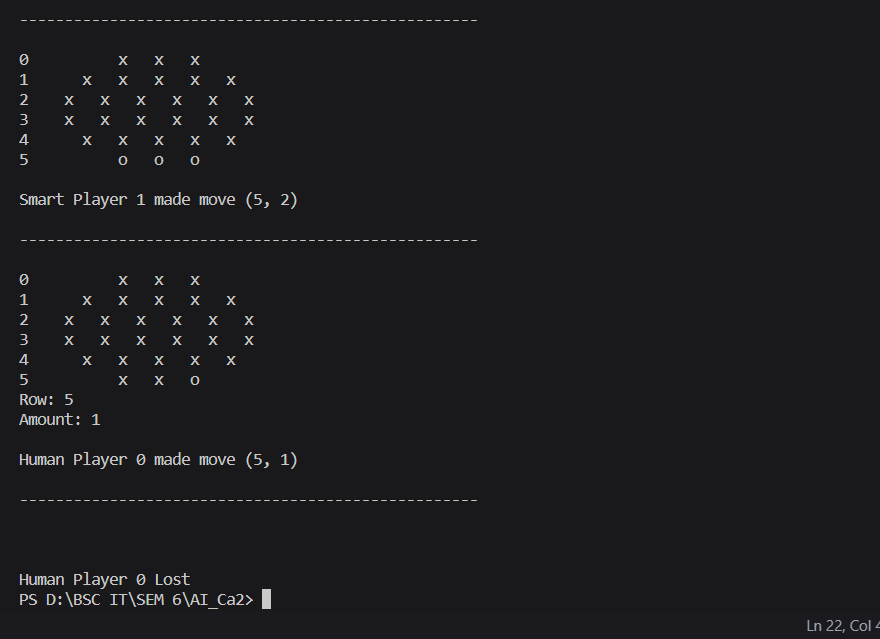
In this project I have used AI Minimax Algorithm. Whenever the human player plays something the only thing AI does is copy the move and mirror it unless it is the last row to solve. If it’s the last row to play then the algorithm will copy Human player’s move-1 piece. Hence, it leaves Human no choice but to play the last move.

The Code for this algorithm is attached below

Screenshots:







1. Board

|  |
| --- |
| 1. *from* math *import* floor *as* fl 2. class Board: 3. def \_\_init\_\_(*self*): 4. self.b = [['o', 'o', 'o'], ['o', 'o', 'o', 'o', 'o'], ['o', 'o', 'o', 'o', 'o', 'o'], 5. ['o', 'o', 'o', 'o', 'o', 'o'], ['o', 'o', 'o', 'o', 'o'], ['o', 'o', 'o']] 6. def update\_b(*self*, *r*, *a*): 7. i = 0 8. *while* i < len(self.b[r]) and a > 0: 9. *if* self.b[r][i] == 'o': 10. self.b[r][i] = 'x' 11. a -= 1 12. i += 1 13. def spot\_avail(*self*, *r*): 14. s = 0 15. *for* i *in* range(len(self.b[r])-1, -1, -1): 16. *if* self.b[r][i] == 'x': 17. *break* 18. s += 1 19. *return* s 20. def row\_empty(*self*, *r*): 21. *for* i *in* range(len(self.b[r])-1, -1, -1): 22. *if* self.b[r][i] == 'o': 23. *return* False 24. *return* True 25. def draw(*self*): 26. *for* i *in* range(len(self.b)): 27. print(str(i) + ' '\*(int(fl((i-2.5)\*\*2)+4)), *end*='') 28. *for* c *in* self.b[i]: 29. print(c, *end*='   ') 30. print() 31. def g\_o(*self*): 32. *for* i *in* self.b: 33. *for* c *in* range(len(i)-1, -1, -1): 34. *if* i[c] == 'o': 35. *return* False 36. *else*: 37. *break* 38. *return* True 39. def \_\_len\_\_(*self*): 40. *return* len(self.b) 41. def num\_row(*self*): 42. temp\_l = [] 43. *for* i *in* range(len(self.b)): 44. s = 0 45. *for* c *in* range(len(self.b[i])-1, -1, -1): 46. *if* self.b[i][c] == 'o': 47. s += 1 48. *else*: 49. *break* 50. temp\_l.append(s) 51. *return* temp\_l 52. def diff(*self*, *r1*, *r2*): 53. r10 = 0 54. r20 = 0 55. *for* i *in* range(0, len(self.b[r1])): 56. *if* self.b[r1][i] == 'o': 57. r10 += 1 58. *if* self.b[r2][i] == 'o': 59. r20 += 1 60. *if* r10 > r20: 61. *return* (r1, r10 - r20) 62. *else*: 63. *return* (r2, r20 - r10) |

2. Player.py

|  |
| --- |
| *from* random *import* randint *as* rand  *import* collections *as* col  class Player:      def \_\_init\_\_(*self*, *board*):          self.board = board      def move(*self*):  *raise* NotImplementedError  class RandomPlayer(Player):      def move(*self*):          r = rand(0, 5)  *while* self.board.row\_empty(r):              r = rand(0, 5)  *return* (r, rand(1, self.board.spot\_avail(r)) + 1)      def \_\_str\_\_(*self*):  *return* 'Random Player'  class HumanPlayer(Player):      def move(*self*):  *try*:              pr = int(input('Row: '))              pa = int(input('Amount: '))  *except* ValueError:              pr = -1              pa = -1  *while* pr > 5 or self.board.row\_empty(pr) or pa <= 0 or pr < 0:              print('Invalid move')  *try*:                  pr = int(input('Row: '))                  pa = int(input('Amount: '))  *except* ValueError:                  pr = -1                  pa = -1  *return* (pr, pa)      def \_\_str\_\_(*self*):  *return* 'Human Player'  class SmartPlayer(Player):      def \_\_init\_\_(*self*, *board*, *pn*):          Player.\_\_init\_\_(self, board)          self.pn = pn      def \_rmove(*self*):          r = rand(0, 5)  *while* self.board.row\_empty(r):              r = rand(0, 5)  *return* (r, rand(1, self.board.spot\_avail(r)))      def move(*self*):          cur\_b = col.Counter(self.board.num\_row())  *if* cur\_b in [{0: 2, 1: 4}, {0: 2, 1: 3, 2: 1}, {0: 2, 1: 3, 3: 1}, {0: 2, 1: 3, 4: 1}, {0: 2, 1: 3, 5: 1}, {0: 2, 1: 3, 6: 1}, {1: 6}, {1: 5, 2: 1}, {1: 5, 3: 1}, {1: 5, 4: 1}, {1: 5, 5: 1}, {1: 5, 6: 1}, {0: 4, 1: 2}, {0: 4, 1: 1, 2: 1}, {0: 4, 1: 1, 3: 1}, {0: 4, 1: 1, 4: 1}, {0: 4, 1: 1, 5: 1}, {0: 4, 1: 1, 6: 1}]:              m = 0              r = 0  *for* i *in* range(len(self.board)):  *if* self.board.spot\_avail(i) > m:                      m = self.board.spot\_avail(i)                      r = i  *return* (r, m)  *elif* cur\_b in [{0: 1, 1: 4, 2: 1}, {0: 1, 1: 4, 3: 1}, {0: 1, 1: 4, 4: 1}, {0: 1, 1: 4, 5: 1}, {0: 1, 1: 4, 6: 1}, {0: 3, 1: 2, 2: 1}, {0: 3, 1: 2, 3: 1}, {0: 3, 1: 2, 4: 1}, {0: 3, 1: 2, 5: 1}, {0: 3, 1: 2, 6: 1}, {0: 5, 2: 1}, {0: 5, 3: 1}, {0: 5, 4: 1}, {0: 5, 5: 1}, {0: 5, 6: 1}]:              m = 0              r = 0  *for* i *in* range(len(self.board)):  *if* self.board.spot\_avail(i) > m:                      m = self.board.spot\_avail(i)                      r = i  *return* (r, m - 1)  *else*:  *if* self.board.b[0] != self.board.b[5]:  *return* self.board.diff(0, 5)  *if* self.board.b[1] != self.board.b[4]:  *return* self.board.diff(1, 4)  *if* self.board.b[2] != self.board.b[3]:  *return* self.board.diff(2, 3)  *else*:  *return* self.\_rmove()      def \_\_str\_\_(*self*):  *return* 'Smart Player' |

3. AI-SmartPlayer.py

|  |
| --- |
| *from* sys *import* maxsize  class Node:      def \_\_init\_\_(*self*, *depth*, *pnum*, *board*, *move*=(0, 0), *val*=0):          self.depth = depth          self.pnum = pnum          self.board = board          self.val = val          self.move = move          self.children = []          self.create\_children()      def create\_children(*self*):  *if* self.depth < 0 or self.val != 0:  *return*  *# This is much messier than before but cuts down tree size because of the symmetry of the board*  *if* self.board.b[0] == self.board.b[5]:  *if* self.board.b[1] == self.board.b[4]:  *if* self.board.b[2] == self.board.b[3]:  *for* r *in* range(3):  *if* not self.board.row\_empty(r):  *for* a *in* range(self.board.spot\_avail(r)):                                  tempb = self.board.dupe()                                  tempb.update\_b(r, a+1)                                  self.children.append(Node(self.depth - 1, -1\*self.pnum, tempb, (r, a+1), self.real\_val(self.board)))  *else*:  *for* r *in* range(4):  *if* not self.board.row\_empty(r):  *for* a *in* range(self.board.spot\_avail(r)):                                  tempb = self.board.dupe()                                  tempb.update\_b(r, a+1)                                  self.children.append(Node(self.depth - 1, -1\*self.pnum, tempb, (r, a+1), self.real\_val(self.board)))  *else*:  *if* self.board.b[2] == self.board.b[3]:  *for* r *in* [0, 1, 2, 4]:  *if* not self.board.row\_empty(r):  *for* a *in* range(self.board.spot\_avail(r)):                                  tempb = self.board.dupe()                                  tempb.update\_b(r, a+1)                                  self.children.append(Node(self.depth - 1, -1\*self.pnum, tempb, (r, a+1), self.real\_val(self.board)))  *else*:  *for* r *in* [0, 1, 2, 3, 4]:  *if* not self.board.row\_empty(r):  *for* a *in* range(self.board.spot\_avail(r)):                                  tempb = self.board.dupe()                                  tempb.update\_b(r, a+1)                                  self.children.append(Node(self.depth - 1, -1\*self.pnum, tempb, (r, a+1), self.real\_val(self.board)))  *else*:  *if* self.board.b[1] == self.board.b[4]:  *if* self.board.b[2] == self.board.b[3]:  *for* r *in* [0, 1, 2, 5]:  *if* not self.board.row\_empty(r):  *for* a *in* range(self.board.spot\_avail(r)):                                  tempb = self.board.dupe()                                  tempb.update\_b(r, a+1)                                  self.children.append(Node(self.depth - 1, -1\*self.pnum, tempb, (r, a+1), self.real\_val(self.board)))  *else*:  *for* r *in* [0, 1, 2, 3, 5]:  *if* not self.board.row\_empty(r):  *for* a *in* range(self.board.spot\_avail(r)):                                  tempb = self.board.dupe()                                  tempb.update\_b(r, a+1)                                  self.children.append(Node(self.depth - 1, -1\*self.pnum, tempb, (r, a+1), self.real\_val(self.board)))  *else*:  *if* self.board.b[2] == self.board.b[3]:  *for* r *in* [0, 1, 2, 4, 5]:  *if* not self.board.row\_empty(r):  *for* a *in* range(self.board.spot\_avail(r)):                                  tempb = self.board.dupe()                                  tempb.update\_b(r, a+1)                                  self.children.append(Node(self.depth - 1, -1\*self.pnum, tempb, (r, a+1), self.real\_val(self.board)))  *else*:  *for* r *in* range(6):  *if* not self.board.row\_empty(r):  *for* a *in* range(self.board.spot\_avail(r)):                                  tempb = self.board.dupe()                                  tempb.update\_b(r, a+1)                                  self.children.append(Node(self.depth - 1, -1\*self.pnum, tempb, (r, a+1), self.real\_val(self.board)))  *# if self.depth >= 0 and self.val == 0:*  *#   for r in range(6):*  *#       if not self.board.row\_empty(r):*  *#           for a in range(self.board.spot\_avail(r)):*  *#               tempb = self.board.dupe()*  *#               tempb.update\_b(r, a+1)*  *#               self.children.append(Node(self.depth - 1, -1\*self.pnum, tempb, (r, a+1), self.RealVal(self.board)))*      def real\_val(*self*, *b*):  *if* b.g\_o():  *return* maxsize \* -1 \* self.pnum  *elif* b.one\_left() or b.win\_board():  *return* maxsize \* self.pnum  *return* 0  def min\_max(*node*, *depth*, *pnum*):  *if* (depth == 0) or (abs(node.val) == maxsize):  *return* node.val      best\_val = maxsize \* -pnum  *for* child *in* node.children:          val = min\_max(child, depth - 1, -1 \* pnum)  *if* abs(maxsize \* pnum - val) < abs(maxsize \* pnum - best\_val):              best\_val = val  *# if bestV == maxsize\*pnum:*  *#   break*  *return* best\_val  def good\_depth(*b*):      l = b.num\_row()      s = 0  *for* i *in* l:          s += i  *if* s > 20:  *return* 0      r = 0  *for* i *in* l:  *if* i > 0:              r += 1  *if* r > 4 and s > 15:  *return* 2  *if* r > 3 and s > 10:  *return* 4  *if* r > 3:  *return* 6  *return* 8 |

4. Game.py

|  |
| --- |
| *from* board *import* Board  *from* player *import* HumanPlayer, RandomPlayer, SmartPlayer  class Game:      def \_\_init\_\_(*self*):          self.b = Board()      def make\_move(*self*, *r*, *a*):          self.b.update\_b(r, a)      def draw\_board(*self*):          self.b.draw()      def game\_over(*self*):  *return* self.b.g\_o()      def run\_game(*self*, *num\_h*, *num\_r*, *num\_s*):          t = 0          l = []  *for* i *in* range(num\_h):              l.append(HumanPlayer(self.b))              t += 2  *for* i *in* range(num\_r):              l.append(RandomPlayer(self.b))              t += 2  *for* i *in* range(num\_s):              l.append(SmartPlayer(self.b, -1 + t))              t += 2  *while* not self.game\_over():  *for* playr *in* l:                  self.draw\_board()                  plm = playr.move()                  self.make\_move(plm[0], plm[1])                  print('\n' + str(playr) + ' ' + str(l.index(playr)) + ' made move (' + str(plm[0]) + ', ' + str(plm[1]) + ')\n\n---------------------------------------------------\n')  *if* self.game\_over():                      print('\n\n' + str(playr) + ' ' + str(l.index(playr)) + ' Lost')  *return* l.index(playr)  *if* \_\_name\_\_ == '\_\_main\_\_':      g = Game()      g.run\_game(1, 0, 1) |