**PROGRAM-Consoled Based Ludo GAME**

**Run.py**

#!/usr/bin/env python3

from ludo.cli import CLIGame

CLIGame().start()

**cli.py-**

from .game import Player, Gamefrom .painter import present\_6\_die\_name

from .recorder import RunRecord, MakeRecord

from os import linesep

**class CLIGame():**

def \_\_init\_\_(self):

self.prompt\_end = "> "

self.game = Game()

# used for nicer print

self.prompted\_for\_pawn = False

# saving game data

self.record\_maker = MakeRecord()

# getting game data

self.record\_runner = None

def validate\_input(self, prompt, desire\_type, allawed\_input=None,

error\_mess="Invalid Option!", str\_len=None):

'''

loop while receive correct value

param allowed\_input can be list of allowed values

param str\_len is two sized tuple if min and max

'''

prompt += linesep + self.prompt\_end

while True:

choice = input(prompt)

if not choice:

print(linesep + error\_mess)

continue

try:

choice = desire\_type(choice)

except ValueError:

print(linesep + error\_mess)

continue

if allawed\_input:

if choice in allawed\_input:

break

else:

print("Invalid Option!")

continue

elifstr\_len:

min\_len, max\_len = str\_len

if min\_len<len(choice) <max\_len:

break

else:

print(linesep + error\_mess)

else:

break

print()

return choice

def get\_user\_initial\_choice(self):

text = linesep.join(["choose option",

"0 - start new game",

"1 - continue game",

"2 - run (review) recorded game"])

choice = self.validate\_input(text, int, (0, 1, 2))

return choice

def prompt\_for\_file(self, mode="rb"):

'''return file descriptor'''

text = "Enter filename (name of the record)"

while True:

filename = self.validate\_input(text, str)

try:

file\_descr = open(filename, mode=mode)

return file\_descr

except IOError as e:

print(e)

print("Try again")

def does\_user\_want\_save\_game(self):

'''return True if user want to save

game or False

'''

text = linesep.join(["Save game?",

"0 - No",

"1 - Yes"])

choice = self.validate\_input(text, int, (0, 1))

return choice == 1

def prompt\_for\_player(self):

''' get player attributes from input,

initial player instance and

add player to the game

'''

available\_colours = self.game.get\_available\_colours()

text = linesep.join(["choose type of player",

"0 - computer",

"1 - human"])

choice = self.validate\_input(text, int, (0, 1))

if choice == 1:

name = self.validate\_input("Enter name for player",

str, str\_len=(1, 30))

available\_options = range(len(available\_colours))

if len(available\_options) > 1:

# show available colours

options = ["{} - {}".format(index, colour)

for index, colour in

zip(available\_options,

available\_colours)]

text = "choose colour" + linesep

text += linesep.join(options)

choice = self.validate\_input(text, int, available\_options)

colour = available\_colours.pop(choice)

else:

# only one colour left

colour = available\_colours.pop()

player = Player(colour, name, self.prompt\_choose\_pawn)

elif choice == 0:

# automatically assign colours

colour = available\_colours.pop()

player = Player(colour)

self.game.add\_palyer(player)

def prompt\_for\_players(self):

'''put all players in the game'''

counts = ("first", "second", "third", "fourth last")

text\_add = "Add {} player"

for i in range(2):

print(text\_add.format(counts[i]))

self.prompt\_for\_player()

print("Player added")

text = linesep.join(["Choose option:",

"0 - add player",

"1 - start game with {} players"])

for i in range(2, 4):

choice = self.validate\_input(text.format(str(i)), int, (0, 1))

if choice == 1:

break

elif choice == 0:

print(text\_add.format(counts[i]))

self.prompt\_for\_player()

print("Player added")

def prompt\_choose\_pawn(self):

'''used when player (human) has more than

one possible pawn to move.

This method is pass as a callable during

player instantiation

'''

text = present\_6\_die\_name(self.game.rolled\_value,

str(self.game.curr\_player))

text += linesep + "has more than one possible pawns to move."

text += " Choose pawn" + linesep

pawn\_options = ["{} - {}".format(index + 1, pawn.id)

for index, pawn

in enumerate(self.game.allowed\_pawns)]

text += linesep.join(pawn\_options)

index = self.validate\_input(

text, int, range(1, len(self.game.allowed\_pawns) + 1))

self.prompted\_for\_pawn = True

return index - 1

def prompt\_to\_continue(self):

text = "press Enter to continue" + linesep

input(text)

def print\_players\_info(self):

word = "start" if self.game.rolled\_value is None else "continue"

print("Game {} with {} players:".format(

word,

len(self.game.players)))

for player in self.game.players:

print(player)

print()

def print\_info\_after\_turn(self):

'''it used game attributes to print info'''

pawns\_id = [pawn.id for pawn in self.game.allowed\_pawns]

# nicer print of dice

message = present\_6\_die\_name(self.game.rolled\_value,

str(self.game.curr\_player))

message += linesep

if self.game.allowed\_pawns:

message\_moved = "{} is moved. ".format(

self.game.picked\_pawn.id)

if self.prompted\_for\_pawn:

self.prompted\_for\_pawn = False

print(message\_moved)

return

message += "{} possible pawns to move.".format(

" ".join(pawns\_id))

message += " " + message\_moved

if self.game.jog\_pawns:

message += "Jog pawn "

message += " ".join([pawn.id for pawn in self.game.jog\_pawns])

else:

message += "No possible pawns to move."

print(message)

def print\_standing(self):

standing\_list = ["{} - {}".format(index + 1, player)

for index, player in enumerate(self.game.standing)]

message = "Standing:" + linesep + linesep.join(standing\_list)

print(message)

def print\_board(self):

print(self.game.get\_board\_pic())

def run\_recorded\_game(self):

'''get history of game (rolled\_value

and index's allowed pawn) from

record\_runnerin order to replay game'''

self.load\_recorded\_players()

self.print\_players\_info()

self.prompt\_to\_continue()

for rolled\_value, index in self.record\_runner:

self.game.play\_turn(index, rolled\_value)

self.print\_info\_after\_turn()

self.print\_board()

self.prompt\_to\_continue()

self.print\_board()

def continue\_recorded\_game(self):

'''move forward the game by calling

play\_turn method to the moment

where game was interrupted.

'''

self.load\_recorded\_players()

self.record\_players()

for rolled\_value, index in self.record\_runner:

self.game.play\_turn(index, rolled\_value)

self.record\_maker.add\_game\_turn(

self.game.rolled\_value, self.game.index)

self.print\_players\_info()

self.print\_info\_after\_turn()

self.print\_board()

def record\_players(self):

'''save players on recorder'''

for player in self.game.players:

self.record\_maker.add\_player(player)

def load\_recorded\_players(self):

'''get recorded (save) players from

recorder and put them in game

'''

if self.record\_runner is None:

file\_descr = self.prompt\_for\_file()

self.record\_runner = RunRecord(file\_descr)

file\_descr.close()

for player in self.record\_runner.get\_players(

self.prompt\_choose\_pawn):

self.game.add\_palyer(player)

def load\_players\_for\_new\_game(self):

self.prompt\_for\_players()

self.print\_players\_info()

self.record\_players()

def play\_game(self):

'''mainly calling play\_turn

Game's method while game finished

'''

try:

while not self.game.finished:

self.game.play\_turn()

self.print\_info\_after\_turn()

self.print\_board()

self.record\_maker.add\_game\_turn(

self.game.rolled\_value, self.game.index)

self.prompt\_to\_continue()

print("Game finished")

self.print\_standing()

self.offer\_save\_game()

except (KeyboardInterrupt, EOFError):

print(linesep +

"Exiting game. " +

"Save game and continue same game later?")

self.offer\_save\_game()

raise

def offer\_save\_game(self):

'''offer user save game'''

if self.does\_user\_want\_save\_game():

file\_descr = self.prompt\_for\_file(mode="wb")

self.record\_maker.save(file\_descr)

file\_descr.close()

print("Game is saved")

def start(self):

'''main method, starting cli'''

print()

try:

choice = self.get\_user\_initial\_choice()

if choice == 0: # start new game

self.load\_players\_for\_new\_game()

self.play\_game()

elif choice == 1: # continue game

self.continue\_recorded\_game()

if self.game.finished:

print("Could not continue.",

"Game is already finished",

linesep + "Exit")

else:

self.prompt\_to\_continue()

self.play\_game()

elif choice == 2: # review played game

self.run\_recorded\_game()

except (KeyboardInterrupt, EOFError):

print(linesep + "Exit Game")

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

CLIGame().start()

**Game.py-**

from collections import namedtuple, dequeimport random

from .painter import PaintBoard

# Thanks to Angel Angelov

# This is piece or a token in ludo game

# Simple class has only index, colour and id attributes

Pawn = namedtuple("Pawn", "index colour id")

class Player():

'''Knows (holds) his pawns,

also know his colour

and choose which pawn to move

if more than one are possible

'''

def \_\_init\_\_(self, colour, name=None, choose\_pawn\_delegate=None):

'''choose\_pawn\_delegate is callable.

if choose\_pawn\_delegate is not None it is called

with argument list of available pawns to move

and expect chosen index from this list

if it is None (means computer) random index is chosen

'''

self.colour = colour

self.choose\_pawn\_delegate = choose\_pawn\_delegate

self.name = name

if self.name is None and self.choose\_pawn\_delegate is None:

self.name = "computer"

self.finished = False

# initialize four pawns with

# id (first leter from colour and index (from 1 to 4))

self.pawns = [Pawn(i, colour, colour[0].upper() + str(i))

for i in range(1, 5)]

def \_\_str\_\_(self):

return "{}({})".format(self.name, self.colour)

def choose\_pawn(self, pawns):

'''Delegate choice to choose\_pawn\_delegatefunc attribute

if it is not None

'''

if len(pawns) == 1:

index = 0

eliflen(pawns) > 1:

if self.choose\_pawn\_delegate is None:

index = random.randint(0, len(pawns) - 1)

else:

index = self.choose\_pawn\_delegate()

return index

class Board():

'''

Knows where are pawns.

Pawns are assigned with position numbers.

Can move (change position number) pawn.

Knows other things like

what distance pawn must past to reach end.

It just board. It does not know rules of the game.

'''

# common (shared) squares for all pawns

BOARD\_SIZE = 56

# save (private) positions (squares) for each colour

# This is squares just before pawn finished

BOARD\_COLOUR\_SIZE = 7

COLOUR\_ORDER = ['yellow', 'blue', 'red', 'green']

# distance between two neighbourcolours

# (The distance from start square of one colour

# to start square of next colour)

COLOUR\_DISTANCE = 14

def \_\_init\_\_(self):

#fn1353c

# get dict of start position for every colour

Board.COLOUR\_START = {

colour: 1 + index \* Board.COLOUR\_DISTANCE for

index, colour in enumerate(Board.COLOUR\_ORDER)}

# get dict of end position for every colour

Board.COLOUR\_END = {

colour: index \* Board.COLOUR\_DISTANCE

for index, colour in enumerate(Board.COLOUR\_ORDER)}

Board.COLOUR\_END['yellow'] = Board.BOARD\_SIZE

# dict where key is pawn and

# value is two size tuple holds position

# Position is combination of

# common (share) square and coloured (private) square.

self.pawns\_possiotion = {}

# painter is used to visually represent

# the board and position of the pawns

self.painter = PaintBoard()

# pool means before start1353

self.board\_pool\_position = (0, 0)

def set\_pawn(self, pawn, position):

'''save position'''

self.pawns\_possiotion[pawn] = position

def put\_pawn\_on\_board\_pool(self, pawn):

self.set\_pawn(pawn, self.board\_pool\_position)

def is\_pawn\_on\_board\_pool(self, pawn):

'''return True of False'''

return self.pawns\_possiotion[pawn] == self.board\_pool\_position

def put\_pawn\_on\_starting\_square(self, pawn):

start = Board.COLOUR\_START[pawn.colour.lower()]

position = (start, 0)

self.set\_pawn(pawn, position)

def can\_pawn\_move(self, pawn, rolled\_value):

'''check if pawn can outside board colour size'''

common\_poss, private\_poss = self.pawns\_possiotion[pawn]

if private\_poss + rolled\_value>self.BOARD\_COLOUR\_SIZE:

return False

return True

def move\_pawn(self, pawn, rolled\_value):

'''change pawn position, check

if pawn reach his color square

'''

common\_poss, private\_poss = self.pawns\_possiotion[pawn]

end = self.COLOUR\_END[pawn.colour.lower()]

if private\_poss> 0:

# pawn is already reached own final squares

private\_poss += rolled\_value

elifcommon\_poss<= end and common\_poss + rolled\_value> end:

# pawn is entering in own squares

private\_poss += rolled\_value - (end - common\_poss)

common\_poss = end

else:

# pawn will be still in common square

common\_poss += rolled\_value

if common\_poss>self.BOARD\_SIZE:

common\_poss = common\_poss - self.BOARD\_SIZE

position = common\_poss, private\_poss

self.set\_pawn(pawn, position)

def does\_pawn\_reach\_end(self, pawn):

'''if pawn must leave game'''

common\_poss, private\_poss = self.pawns\_possiotion[pawn]

if private\_poss == self.BOARD\_COLOUR\_SIZE:

return True

return False

def get\_pawns\_on\_same\_postion(self, pawn):

'''return list of pawns on same position'''

position = self.pawns\_possiotion[pawn]

return [curr\_pawn for curr\_pawn, curr\_postion in

self.pawns\_possiotion.items()

if position == curr\_postion]

def paint\_board(self):

'''painter object expect dict of

key - occupied positions and

value - list of pawns on that position

'''

positions = {}

for pawn, position in self.pawns\_possiotion.items():

common, private = position

if not private == Board.BOARD\_COLOUR\_SIZE:

positions.setdefault(position, []).append(pawn)

return self.painter.paint(positions)

class Die():

MIN = 1

MAX = 6

@staticmethod

def throw():

return random.randint(Die.MIN, Die.MAX)

class Game():

'''Knows the rules of the game.

Knows for example what to do when

one pawn reach another

or pawn reach end or

player roll six and so on

'''

def \_\_init\_\_(self):

self.players = deque()

self.standing = []

self.board = Board()

# is game finished

self.finished = False

# last rolled value from die (dice)

self.rolled\_value = None

# player who last rolled die

self.curr\_player = None

# curr\_player's possible pawn to move

self.allowed\_pawns = []

# curr\_player's chosen pawn to move

self.picked\_pawn = None

# chosen index from allowed pawn

self.index = None

# jog pawn if any

self.jog\_pawns = []

def add\_palyer(self, player):

self.players.append(player)

for pawn in player.pawns:

self.board.put\_pawn\_on\_board\_pool(pawn)

def get\_available\_colours(self):

'''if has available colour on boards'''

used = [player.colour for player in self.players]

available = set(self.board.COLOUR\_ORDER) - set(used)

return sorted(available)

def \_get\_next\_turn(self):

'''Get next player's turn.

It is underscore because if called

outside the class will break order

'''

if not self.rolled\_value == Die.MAX:

self.players.rotate(-1)

return self.players[0]

def get\_pawn\_from\_board\_pool(self, player):

'''when pawn must start'''

for pawn in player.pawns:

if self.board.is\_pawn\_on\_board\_pool(pawn):

return pawn

def get\_allowed\_pawns\_to\_move(self, player, rolled\_value):

''' return all pawns of a player which rolled value

from die allowed to move the pawn

'''

allowed\_pawns = []

if rolled\_value == Die.MAX:

pawn = self.get\_pawn\_from\_board\_pool(player)

if pawn:

allowed\_pawns.append(pawn)

for pawn in player.pawns:

if not self.board.is\_pawn\_on\_board\_pool(pawn) and\

self.board.can\_pawn\_move(pawn, rolled\_value):

allowed\_pawns.append(pawn)

return sorted(allowed\_pawns, key=lambda pawn: pawn.index)

def get\_board\_pic(self):

return self.board.paint\_board()

def \_jog\_foreign\_pawn(self, pawn):

pawns = self.board.get\_pawns\_on\_same\_postion(pawn)

for p in pawns:

if p.colour != pawn.colour:

self.board.put\_pawn\_on\_board\_pool(p)

self.jog\_pawns.append(p)

def \_make\_move(self, player, pawn):

'''tell the board to move pawn.

After move ask board if pawn reach end or

jog others pawn. Check if pawn and player finished.

'''

if self.rolled\_value == Die.MAX and\

self.board.is\_pawn\_on\_board\_pool(pawn):

self.board.put\_pawn\_on\_starting\_square(pawn)

self.\_jog\_foreign\_pawn(pawn)

return

self.board.move\_pawn(pawn, self.rolled\_value)

if self.board.does\_pawn\_reach\_end(pawn):

player.pawns.remove(pawn)

if not player.pawns:

self.standing.append(player)

self.players.remove(player)

if len(self.players) == 1:

self.standing.extend(self.players)

self.finished = True

else:

self.\_jog\_foreign\_pawn(pawn)

def play\_turn(self, ind=None, rolled\_val=None):

'''this is main method which must be used to play game.

Method ask for next player's turn, roll die, ask player

to choose pawn, move pawn.

ind and rolled\_val are suitable to be used when

game must be replicated (recorded)

ind is chosen index from allowed pawns

'''

self.jog\_pawns = []

self.curr\_player = self.\_get\_next\_turn()

if rolled\_val is None:

self.rolled\_value = Die.throw()

else:

self.rolled\_value = rolled\_val

self.allowed\_pawns = self.get\_allowed\_pawns\_to\_move(

self.curr\_player, self.rolled\_value)

if self.allowed\_pawns:

if ind is None:

self.index = self.curr\_player.choose\_pawn(

self.allowed\_pawns)

else:

self.index = ind

self.picked\_pawn = self.allowed\_pawns[self.index]

self.\_make\_move(self.curr\_player, self.picked\_pawn)

else:

self.index = -1

self.picked\_pawn = None

**OUTPUT-**

C:\Users\User\Desktop\python\ludoconsole\_based>python run.py

choose option

0 - start new game

1 - continue game

2 - run (review) recorded game

> 0

Add first player

choose type of player

0 - computer

1 - human

> 0

Player added

Add second player

choose type of player

0 - computer

1 - human

> 1

Enter name for player

> shruti

choose colour

0 - blue

1 - green

2 - red

> 0

Player added

Choose option:

0 - add player

1 - start game with 2 players

> 1

Game start with 2 players:

computer(yellow)

shruti(blue)

---------

| # # |

| # | shruti(blue)

| # # |

---------

No possible pawns to move.

################################################################################

###########

# # | | # |

#

# YELLOW #-----#-----#-----# V BLUE

#

# # # # #

#

# ----------- #-----#-----#-----# -----------

#

# | Y1 | Y2 | # # # # | B1 | B2 |

#

# ----------- #-----#-----#-----# -----------

#

# | Y3 | Y4 | # # # # | B3 | B4 |

#

# ----------- #-----#-----#-----# -----------

#

# # # # #

#

# #-----#-----#-----#

#

# --> # # # #

#

#####################################-----#-----#-----##########################

###########

# | | | | | | # # | | | | |

| #

#-----#####################################-----################################

#####-----#

# | | | | | | | X | | | | | |

| #

#-----#####################################-----################################

#####-----#

# | | | | | | # # | | | | |

| #

#####################################-----#-----#-----##########################

###########

# # # # #

<-- #

# GREEN #-----#-----#-----# RED

#

# # # # #

#

# ----------- #-----#-----#-----# -----------

#

# | | | # # # # | | |

#

# ----------- #-----#-----#-----# -----------

#

# | | | # # # # | | |

#

# ----------- #-----#-----#-----# -----------

#

# # # # #

#

# ^ #-----#-----#-----#

#

# | # | | #

#

################################################################################

###########

press Enter to continue

---------

| |

| # | computer(yellow)

| |

---------

No possible pawns to move.

################################################################################

###########

# # | | # |

#

# YELLOW #-----#-----#-----# V BLUE

#

# # # # #

#

# ----------- #-----#-----#-----# -----------

#

# | Y1 | Y2 | # # # # | B1 | B2 |

#

# ----------- #-----#-----#-----# -----------

#

# | Y3 | Y4 | # # # # | B3 | B4 |

#

# ----------- #-----#-----#-----# -----------

#

# # # # #

#

# #-----#-----#-----#

#

# --> # # # #

#

#####################################-----#-----#-----##########################

###########

# | | | | | | # # | | | | |

| #

#-----#####################################-----################################

#####-----#

# | | | | | | | X | | | | | |

| #

#-----#####################################-----################################

#####-----#

# | | | | | | # # | | | | |

| #

#####################################-----#-----#-----##########################

###########

# # # # #

<-- #

# GREEN #-----#-----#-----# RED

#

# # # # #

#

# ----------- #-----#-----#-----# -----------

#

# | | | # # # # | | |

#

# ----------- #-----#-----#-----# -----------

#

# | | | # # # # | | |

#

# ----------- #-----#-----#-----# -----------

#

# # # # #

#

# ^ #-----#-----#-----#

#

# | # | | #

#

################################################################################

###########

press Enter to continue

has more than one possible pawns to move. Choose pawn

1 - B1

2 - B2

> 1

B1 is moved.

################################################################################

###########

# # | | # |

#

# YELLOW #-----#-----#-----# V BLUE

#

# # # # #

#

# ----------- #-----#-----#-----# -----------

#

# | Y1 | Y2 | # # # # | | B2 |

#

# ----------- #-----#-----#-----# -----------

#

# | Y3 | Y4 | # # # # | B3 | B4 |

#

# ----------- #-----#-----#-----# -----------

#

# # # # #

#

# #-----#-----#-----#

#

# --> # # # #

#

#####################################-----#-----#-----##########################

# | | | | | | # #B1 | | | | |

| #

#-----#####################################-----################################

# | | | | | | | X | | | | | |

| #

#-----#####################################-----################################

# | | | | | | # # | | | | |

| #

#####################################-----#-----#-----##########################

###########

# # # # #

<-- #

# GREEN #-----#-----#-----# RED

#

# # # # #

#

# ----------- #-----#-----#-----# -----------

#

# | | | # # # # | | |

#

# ----------- #-----#-----#-----# -----------

#

# | | | # # # # | | |

#

# ----------- #-----#-----#-----# -----------

#

# # # # #

#

# ^ #-----#-----#-----#

#

# | # | | #

#

###############################################################################