Final Project

# Project Description

1. Project Title
   1. Tic Tac Toe Game Engine
2. Project Description and Scope
   1. This project will be a game of tic tac toe with an intelligent opponent that will respond to the players moves with its own moves.
3. What Type of Agent? (Problem Solving, Logic, Learning, Other)
   1. The agent will be a logic agent
4. Description of the Task Environment (Performance Measure, Environment, Percepts, Actuators)
   1. The task environment is a simulated tic tac toe game board with a human player and an AI agent
   2. The performance measure is odds of winning given a particular move
   3. The environment will be the game board
   4. The percept will be an evaluation of the remaining available positions on each turn
   5. The actuator will be the selection of a position on the game board
5. Description of any datasets used in the project.
   1. I will create my own dataset which will be every possible sequence of gameplay
6. How far are you in your project? If you have implemented any parts of your code, include them in the submitted document.
   1. I have started but I do not have any working code yet. I am still working out the best design for the gameboard, player, and game classes.

# Project Submission

## Video

<https://youtu.be/2MnSmyIERPA>

## 

## Code

import abc

import functools

import pathlib

import random

WINS = [

[0, 1, 2],

[3, 4, 5],

[6, 7, 8],

[0, 3, 6],

[1, 4, 7],

[2, 5, 8],

[0, 4, 8],

[2, 4, 6]

]

class Board:

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

self.positions = [0, 1, 2, 3, 4, 5, 6, 7, 8]

self.sequence = []

def display(self):

p = self.positions

div = "+---+---+---+\n"

r1 = f"| {p[0]} | {p[1]} | {p[2]} |\n"

r2 = f"| {p[3]} | {p[4]} | {p[5]} |\n"

r3 = f"| {p[6]} | {p[7]} | {p[8]} |\n"

print(div + r1 + div + r2 + div + r3 + div)

@property

def available\_positions(self):

return [p for p in self.positions if isinstance(p, int)]

def assign\_position(self, position, symbol):

self.positions[position] = symbol

self.sequence.append(position)

def is\_full(self):

return len(self.available\_positions) == 0

def has\_win(self):

for win in WINS:

if self.positions[win[0]] == self.positions[win[1]] == self.positions[win[2]]:

return True

return False

class Player(abc.ABC):

def \_\_init\_\_(self, symbol: str):

self.symbol = symbol

self.positions = []

@abc.abstractmethod

def get\_position(self, board: Board) -> int:

"""Implement the logic for selecting a position"""

class Human(Player):

def get\_position(self, board):

board.display()

while True:

position = int(input("Choose an available position: "))

if position not in board.available\_positions:

print("That position is taken")

continue

else:

break

return position

class Random(Player):

def get\_position(self, board):

position = random.choice(board.available\_positions)

print(f"Random selected position {position}.")

return position

class AIEasy(Player):

"""Works toward a win if any remain, but is not optimized.

It leverages a knowledge base of known winning game sequences.

"""

@functools.cache

def load\_winning\_sequences(self):

"""Knowledge base is compiled from the perspective of the first player.

Therefore, the AI needs to choose appropriate game sequences depending on

whether it goes first or second.

This method is cached because it is called at every turn.

"""

if self.symbol == "x":

sequence\_file\_name = "win.txt"

else:

sequence\_file\_name = "lose.txt"

fp = pathlib.Path(\_\_file\_\_).parent / "data" / sequence\_file\_name

with open(fp, "r") as f:

return [line.strip() for line in f.readlines()]

def get\_position(self, board: Board) -> int:

current\_sequence = "".join([str(i) for i in board.sequence])

winning\_sequences = self.load\_winning\_sequences()

available\_winning\_sequences = [s for s in winning\_sequences if s.startswith(current\_sequence)]

if available\_winning\_sequences:

random\_winning\_sequences = random.choice(available\_winning\_sequences)

next\_move\_index = len(current\_sequence)

return int(random\_winning\_sequences[next\_move\_index])

else:

return random.choice(board.available\_positions)

class AIMedium(Player):

"""This bot prioritize winning moves.

It leverages a knowledge base of winning game combinations.

"""

def get\_intersection(self, l1, l2):

return [i for i in l1 if i in l2]

def get\_difference(self, l1, l2):

return [i for i in l1 if i not in l2]

def get\_opponent\_symbol(self):

if self.symbol == "x":

return "o"

return "x"

def get\_position(self, board: Board) -> int:

# Begin randomly

if not self.positions:

position = random.choice(board.available\_positions)

self.positions.append(position)

return position

# Filter out any wins opponent is working on

possible\_wins = []

for win in WINS:

if not any([board.positions[p] == self.get\_opponent\_symbol() for p in win]):

possible\_wins.append(win)

for win in possible\_wins:

# if already have 2 positions, choose winning position

if len(self.get\_intersection(win, self.positions)) == 2:

position = self.get\_difference(win, self.positions)[0]

self.positions.append(position)

return position

for win in possible\_wins:

# Continue working on the first win in progress

if len(self.get\_intersection(win, self.positions)) == 1:

position = random.choice(self.get\_difference(win, self.positions))

self.positions.append(position)

return position

position = random.choice(board.available\_positions)

self.positions.append(position)

return position

class AIHard(Player):

"""This bot prioritize winning moves and blocks opponent wins.

It leverages a knowledge base of winning game combinations.

"""

def get\_intersection(self, l1, l2):

return [i for i in l1 if i in l2]

def get\_difference(self, l1, l2):

return [i for i in l1 if i not in l2]

def get\_opponent\_symbol(self):

if self.symbol == "x":

return "o"

return "x"

def get\_opponent\_positions(self, positions):

opponent\_symbol = self.get\_opponent\_symbol()

opponent\_positions = []

for position, symbol in enumerate(positions):

if symbol == opponent\_symbol:

opponent\_positions.append(position)

return opponent\_positions

def get\_position(self, board: Board) -> int:

# Begin randomly

if not self.positions:

position = random.choice(board.available\_positions)

self.positions.append(position)

return position

# Filter out any wins opponent is working on

possible\_wins = []

opponent\_wins = []

opponent\_positions = self.get\_opponent\_positions(board.positions)

for win in WINS:

if self.get\_intersection(win, opponent\_positions):

opponent\_wins.append(win)

else:

possible\_wins.append(win)

# Block opponents win

for win in opponent\_wins:

if len(self.get\_intersection(win, opponent\_positions)) == 2:

position = self.get\_difference(win, opponent\_positions)[0]

if position not in board.available\_positions:

continue

self.positions.append(position)

return position

for win in possible\_wins:

# if already have 2 positions, choose winning position

if len(self.get\_intersection(win, self.positions)) == 2:

position = self.get\_difference(win, self.positions)[0]

self.positions.append(position)

return position

for win in possible\_wins:

# Continue working on the first win in progress

if len(self.get\_intersection(win, self.positions)) == 1:

position = random.choice(self.get\_difference(win, self.positions))

self.positions.append(position)

return position

position = random.choice(board.available\_positions)

self.positions.append(position)

return position

class Game:

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

self.board = Board()

self.player\_x = Human("x")

self.player\_o = AIHard("o")

self.current\_player = self.player\_x

def switch\_players(self):

if self.current\_player == self.player\_x:

self.current\_player = self.player\_o

else:

self.current\_player = self.player\_x

def announce\_winner(self):

self.board.display()

print(f"Player {self.current\_player.symbol} wins!")

def announce\_draw(self):

self.board.display()

print("The game ended in a draw.")

def play(self):

while True:

position = self.current\_player.get\_position(self.board)

self.board.assign\_position(position, self.current\_player.symbol)

if self.board.has\_win():

self.announce\_winner()

break

if self.board.is\_full():

self.announce\_draw()

break

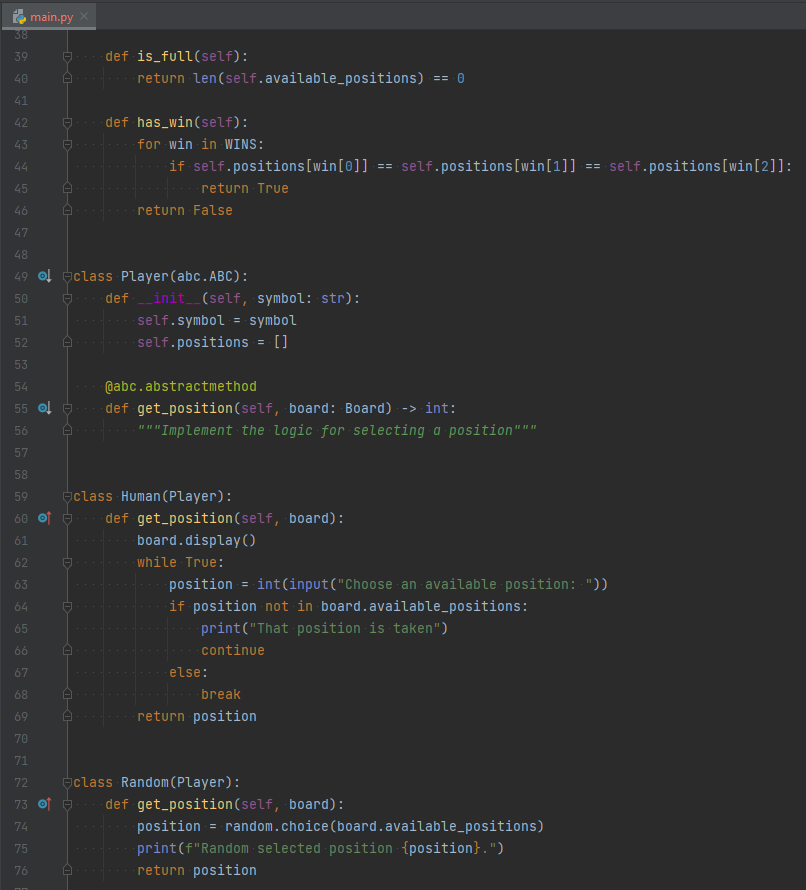
self.switch\_players()

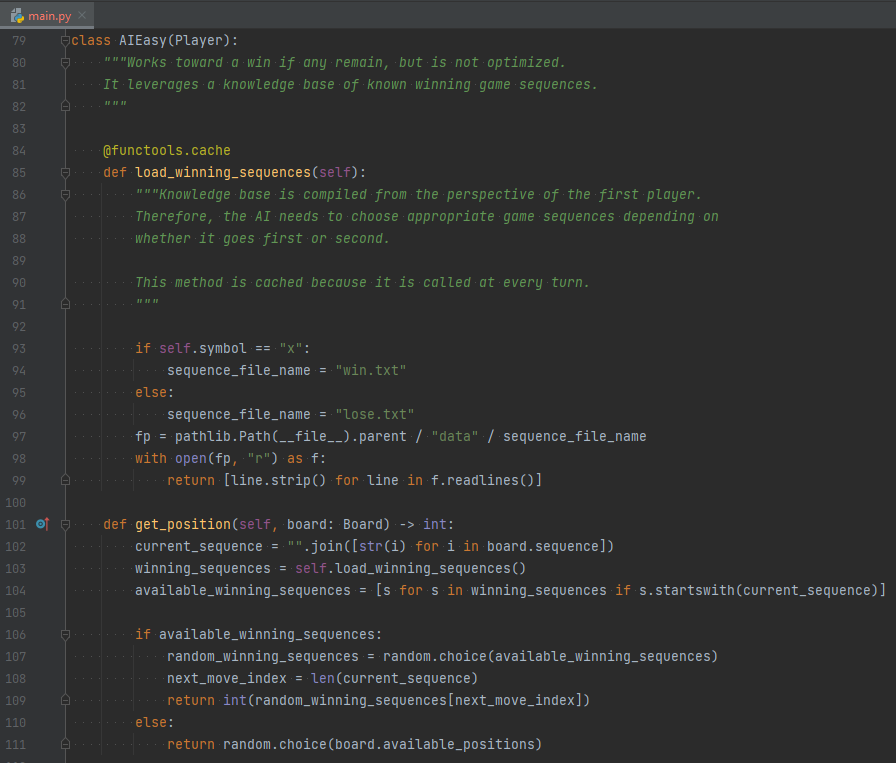
if \_\_name\_\_ == '\_\_main\_\_':

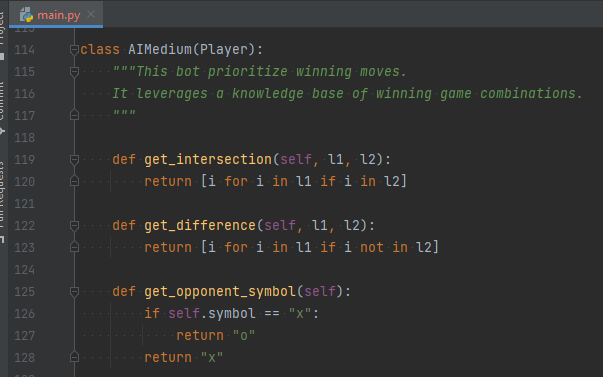
game = Game()

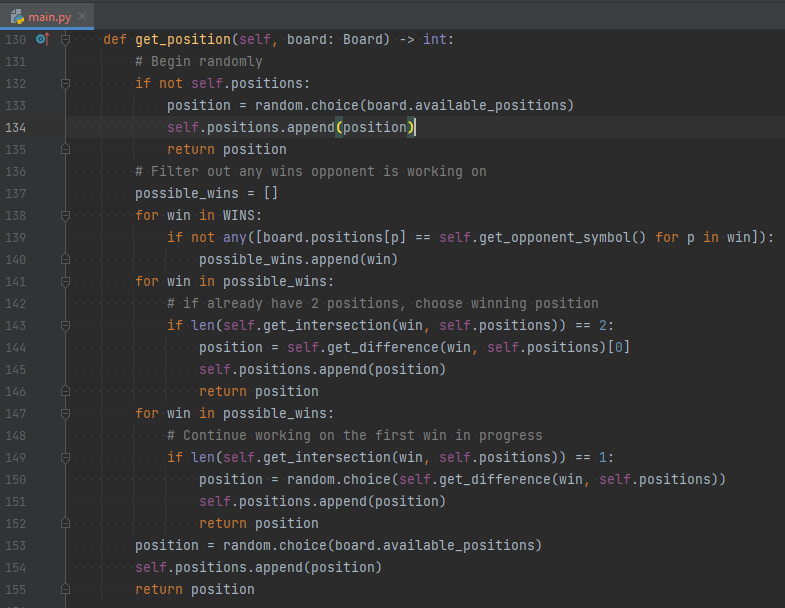
game.play()

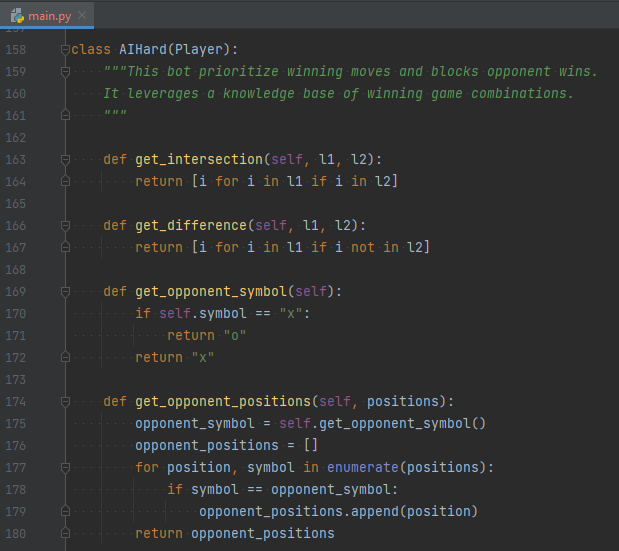
## Screenshots

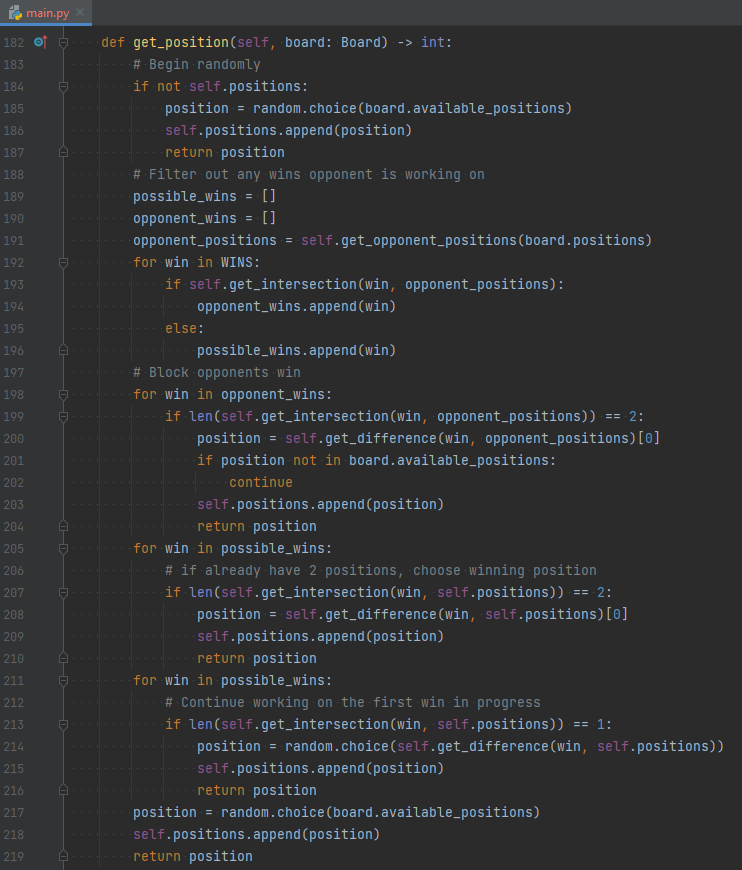














## Results

