Iteration 2 - *Date 2/12/22*

Aims for this iteration

The main aim for this iteration is to create a working chess engine. I may also attempt to start building the frontend interface but this goal is secondary. I then should be able to create a basic chess game that the user can play against the bot.

My chess engine should:

* be functional for basic chess but will not include special moves such as on passant, promotion and castling.
* at least have some user interface which is at minimum console based and allows the user to play a game of chess.
* have a minimax function that is efficient enough to run at depth 2 in a reasonable time.
* It should be rigorously tested to ensure that it is working and that there aren’t any nasty surprises later down the line.

Functionality that the prototype will have

So to meet my main goal and successfully build a chess engine my end product must:

* Be able to correctly determine the legal moves available to a player, accounting for the different movements of pieces and check
* The engine should be able to identify check
* The engine should be able to give a static evaluation of each board state
* The engine should allow for a move to be executed: creating a new child board state that can be examined
* The engine should be able to identify then the game is over, who has won and the outcome type (stalemate, checkmate)

My minimax function should:

* Be able to beat a randomly moving opponent
* Have some consideration of efficiency and efficacy

My unit tests should:

* Be fully automated allowing them to be easily re-run to diagnose problems
* Be a mix of data driven and logic driven tests where appropriate
* My unit tests for minimax should be able to finish and so some efficiency is needed as it will need to undertake many games at depth 2 or 3

Annotated code screenshots with description

Quick note: I have annotated my code by adding comments explaining it in depth. Further comments are made with reference to specific function or classes in word.

I have not added comments explain my VUE GUI as it is now redundant and is no longer being developed (still useful as parts can be reused). I will also explain both the code and the tests in tandem as they were created in tandem (I test as I went along not at the end).

I first began by attempting to make my interface. This was not used in the final product but will be used in future iterations. I used VUE js to create the interface shown:

(screenshot of how I ran the code)

.Text

Description automatically generated

Text

Description automatically generated

Going to the local host, the webpage is running locally on my computer

A screenshot of a game

Description automatically generated with medium confidence

A picture containing chart

Description automatically generated

On clicking a square, a message is printed to the console to show that the correct square click is registered

.

Graphical user interface, application

Description automatically generated

The code for this user interface is below

<template>

    <h1>Chess Game V2</h1>

    <div :style="cssVars" class="chess\_board">

        <table>

            <tr v-for="[row\_num, row] in board.entries()" :key="row">

                <td v-for="[col\_num, square] in row.entries()" :key="square" @click="square\_click(row\_num, col\_num)">

                    <!-- {{row\_num}}, {{col\_num}} -->

                    <!-- {{square !== null? square: ""}} -->

                    <!-- playing around with white pieces -->

                    <!-- https://stackoverflow.com/questions/4772906/css-is-it-possible-to-add-a-black-outline-around-each-character-in-text -->

                    <span style="color:white; text-shadow: 1px 0 0 #000, 0 -1px 0 #000, 0 1px 0 #000, -1px 0 0 #000;" v-if="square == '♙'">♟</span>

                    <!-- <span style="color:black" v-if="square == '♙'">♙</span> -->

                    <!-- <span style="color:white" v-if="square == '♙'">♙</span> -->

                    <!-- <span style="color:black; text-shadow: 1px 0 0 #000, 0 -1px 0 #000, 0 1px 0 #000, -1px 0 0 #000;" v-else-if="square == '♟'">♟</span> -->

                    <span style="color:black" v-else-if="square == '♟'">♟</span>

                    <!-- <span v-else></span> -->

                </td>

            </tr>

        </table>

    </div>

    <h2>Turn {{turn\_num}}: {{next\_to\_go == 'user'? "your turn": "computer's turn"}}</h2>

    <div class="option\_button">

        <button>Concede</button>

        <button>Reset</button>

    </div>

    <div class="pieces\_left\_table">

        <table>

            <tr>

                <th>White Pieces Left: {{pieces\_left.black}}/6</th>

                <th>Black Pieces Left: {{pieces\_left.white}}/6</th>

            </tr>

            <tr>

                <td>{{ '♙'.repeat(pieces\_left.white) }}</td>

                <td>{{ '♟'.repeat(pieces\_left.black) }}</td>

            </tr>

        </table>

    </div>

    <h2>Previous Moves:</h2>

    <div class="previous\_moves\_table">

        <table>

            <tr>

                <th>White previous moves:</th>

                <th>Black previous moves:</th>

            </tr>

            <tr>

                <td>

                    <div v-for="move in previous\_moves" :key="move.num">

                        <span v-if="move.player=='white'">

                            Move {{move.num}}: ♙ {{ move.from }} to {{ move.to }}

                        </span>

                    </div>

                </td>

                <td>

                    <div v-for="move in previous\_moves" :key="move.num">

                        <span v-if="move.player=='black'">

                            Move {{move.num}}: ♟ {{ move.from }} to {{ move.to }}

                        </span>

                    </div>

                </td>

            </tr>

        </table>

    </div>

</template>

<script>

    import {handle\_square\_click} from '@/assets/scripts/external.js'

    // import \* from '@/assets/scripts/external.js'

    const white\_sq\_color = '#f5e6bf';

    const black\_sq\_color = '#66443a';

    // white pawn: '♙'; black pawn: '♟'

    const pawn\_white = '♙';

    const pawn\_black = '♟';

    export default {

        name: "ChessGame",

        ready\_for\_next\_move: false,

        // pawn characters

        // https://en.wikipedia.org/wiki/Chess\_symbols\_in\_Unicode

        data(){return{

            board: [

                Array(8).fill(null),

                Array(8).fill(pawn\_black),

                Array(8).fill(null),

                Array(8).fill(null),

                Array(8).fill(null),

                Array(8).fill(null),

                Array(8).fill(pawn\_white),

                Array(8).fill(null)

            ],

            next\_to\_go: 'user',

            turn\_num: 1,

            pieces\_left: {

                black: 2,

                white: 3

            },

            previous\_moves: [

                { num: 1, player: "white", from: "B1", to: "B3" },

                { num: 2, player: "black", from: "A5", to: "A4" },

                { num: 3, player: "white", from: "B3", to: "A4" },

                { num: 4, player: "black", from: "E5", to: "E3" },

            ],

        }},

        methods: {

            square\_click(row, col) {

                // console.log(`square click: (${row},${col})`)

                handle\_square\_click(row, col)

            }

        },

        // https://www.telerik.com/blogs/passing-variables-to-css-on-a-vue-component

        computed: {

            cssVars(){return{

                '--black\_sq\_color': black\_sq\_color,

                '--white\_sq\_color': white\_sq\_color,

            }}

        }

    }

</script>

<style scoped>

    /\* :root {

        https://www.vectorstock.com/royalty-free-vector/chess-field-in-beige-and-brown-colors-vector-24923385

        https://imagecolorpicker.com/en

        --black\_sq\_color: #66443a;

        --white\_sq\_color: #f5e6bf;

    } \*/

    h1 {

        text-align: center;

        text-decoration: underline;

    }

    h2 {

        text-align: center;

        font-size: 32px;

    }

    table {

        margin: auto;

        table-layout: fixed;

        text-align: center;

    }

    .option\_button {

        margin: auto;

        text-align: center;

        /\* margin-left: 0.125vw;

        margin-right: 0.125vw; \*/

        margin-left: 20px;

        margin-right: 20px;

    }

    .chess\_board td {

        height: 10.5vh;

        width: 10.5vh;

        /\* dimension so it is all in view for ipad air portrait  (820 \* 1180) \*/

        /\* height: 12vw;

        width: 12vw; \*/

        text-align: center;

        /\* border: 3px black solid; \*/

        font-size: 50px;

        /\* font-weight: 100; \*/

    }

    .chess\_board table {

        border: 1px black solid;

    }

    /\* these odd and even rules dictate the color of chess board squares \*/

    .chess\_board tr:nth-child(2n-1) > td:nth-child(2n-1) {

    background-color: var(--white\_sq\_color);

        /\* color: black; \*/

    }

    .chess\_board tr:nth-child(2n-1) > td:nth-child(2n) {

        background-color: var(--black\_sq\_color);

        /\* color: white; \*/

    }

    .chess\_board tr:nth-child(2n) > td:nth-child(2n-1) {

        background-color: var(--black\_sq\_color);

        /\* color: white; \*/

    }

    .chess\_board tr:nth-child(2n) > td:nth-child(2n) {

        background-color: var(--white\_sq\_color);

        /\* color: black; \*/

    }

    .previous\_moves\_table th, td {

        width: 35vw;

        /\* border: 3px solid black; \*/

        font-size: 28px;

    }

    .pieces\_left\_table th, td {

        width: 35vw;

        /\* border: 3px solid black; \*/

        font-size: 28px;

    }

</style>

It is not fully finished as the idea of developing a VUE js GUI was shelved.

­

On the one hand there are many benefits to a VUE GUI. Most importantly, the html content is dynamically updated from JavaScript variables. For example the board shown automatically updates with the content of the 2d array board. The main downside however is that the single page nature of a VUE webpage clashed with what I wanted the flask server to do. This is where one single HTTP request is sent and instead multiple pages are handled by JavaScript loading different components. I used this source and decided that I wanted to implement my user interface like this:

<https://www.digitalocean.com/community/tutorials/how-to-add-authentication-to-your-app-with-flask-login>

I then moved on to implementing the backend chess engine in python. I knew that I intended to implement my chess engine using decision trees and specifically the minimax function. This function is suitable as chess is a complete information game as there is not information hidden such as a hand of cards. It is also a zero sum game as one player’s good move worsens another player’s chance of winning. It is also turn based.

To implement minimax I would need 2 functions. A utility function to determine a number for how favourable the current game state is to the maximiser (utility) and a child game state generating function that could provide all possible game states that could be achieved in one move from the current game state.

To implement the utility function I would need a robust game over function and a static evaluation function to provide an approximate guess as to the utility of a board state.

To implement the child game state generating function I will need to have a function to action a move and generate a child game state as well as a function to generate all legal moves of the current game state.

These function would need to be rigorously tested. If they contained logic errors, then the minimax function would fail without and easy way to determine why. However there functions themselves are hard to implement and so they would need to be broken down into simpler function. I clearly would also need to employ some high quality testing in order to produce sufficiently reliable code.

I decided that at its most abstracted, I would be solving this problem with a series objects built on vectors and rules. So I began by creating a vector class. It was sufficiently simple that it didn’t need any major debugging. This would still provide me a good opportunity to get to grips with my unit-test process.

I decided to use a library called data driven tests (ddt) as it would let me write a single function to execute a single type of test, then run this test for every set of test data specified in a given file.

I thought that this would be better that generating the test data and expected outcome withing the test function as this method is more transparent and reduces the risk of logic errors within the tests themselves.

Here was my vector class (**vector.py**)

# import dataclass to reduce boilerplate code

from dataclasses import dataclass

# frozen = true means that the objects will be immutable

@dataclass(frozen=True)

class Vector():

    # 2d vector has properties i and j

    i: int

    j: int

    # code to allow for  +  -  and  \*  operators to be used with vectors

    def \_\_add\_\_(self, other):

        assert isinstance(other, Vector), "both objects must be instances of the Vector class"

        return Vector(

            i=self.i + other.i,

            j=self.j + other.j

        )

    def \_\_sub\_\_(self, other):

        assert isinstance(other, Vector), "both objects must be instances of the Vector class"

        return Vector(

            i=self.i - other.i,

            j=self.j - other.j

        )

    def \_\_mul\_\_(self, multiplier: int):

        return Vector(

            i=self.i \* multiplier,

            j=self.j \* multiplier

        )

    # check if a vector is in board

    def in\_board(self):

        """Assumes that the current represented vector is a position vector

        checks if it points to a square that isn't in the chess board"""

        return self.i in range(8) and self.j in range(8)

    # alternative way to create instance, construct from chess square

    @classmethod

    def construct\_from\_square(cls, to\_sqr):

        """Example from and to squares are A3 -> v(0, 2) and to B4 -> v(1, 3)"""

        to\_sqr = to\_sqr.upper()

        letter, number = to\_sqr

        # map letters and numbers to 0 to 7 and create new vector object

        return cls(

            i=ord(letter.upper()) - ord("A"),

            j=int(number)-1

        )

    # this function is the reverse and converts a position vector to a square

    def to\_square(self) -> str:

        letter = chr(self.i + ord("A"))

        number = self.j+1

        return f"{letter}{number}"

    # this function checks if 2 vectors are equal

    def \_\_eq\_\_(self, other) -> bool:

        try:

            # assert same subclass like rook

            assert isinstance(other, type(self))

            assert self.i == other.i

            assert self.j == other.j

        except AssertionError:

            return False

        else:

            return True

    # used to put my objects in set

    def \_\_hash\_\_(self):

        return hash((self.i, self.j))

and here were my tests

starting with the python file containing the logic to run the unit tests (**test\_vector.py)**

import ddt

import unittest

# from vector.vector import Vector

from vector import Vector

# function for path to vector related test data

def test\_path(file\_name):

    return f"./test\_data/vector/{file\_name}.yaml"

# augment my test case class with ddt decorator

@ddt.ddt

class Test\_Case(unittest.TestCase):

    # performs many checks of construct form square

    @ddt.file\_data(test\_path("from\_square"))

    def test\_square\_to\_vector(self, square, expected\_vector):

        self.assertEqual(

            Vector.construct\_from\_square(square),

            Vector(\*expected\_vector),

            msg=f"\nVector.construct\_from\_square('{square}')  !=  Vector(i={expected\_vector[0]}, j={expected\_vector[1]})"

        )

    # performs many checks of adding vectors

    @ddt.file\_data(test\_path("vector\_add"))

    def test\_add\_vectors(self, vector\_1, vector\_2, expected\_vector):

        self.assertEqual(

            Vector(\*vector\_1) + Vector(\*vector\_2),

            Vector(\*expected\_vector),

            msg=f"\nVector(\*{vector\_1}) + Vector(\*{vector\_2})  !=  Vector(\*{expected\_vector})"

        )

    # performs many checks of multiplying vectors

    @ddt.file\_data(test\_path("vector\_multiply"))

    def test\_multiply\_vectors(self, vector, multiplier, expected):

        self.assertEqual(

            Vector(\*vector) \* multiplier,

            Vector(\*expected)

        )

    # many tests of vector in board

    @ddt.file\_data(test\_path("vector\_in\_board"))

    def test\_in\_board(self, vector, expected):

        self.assertEqual(

            Vector(\*vector).in\_board(),

            expected,

            msg=f"\nVector(\*{vector}).in\_board()  !=  {expected}"

        )

    # many tests of vector out of board

    @ddt.file\_data(test\_path('vector\_to\_square'))

    def test\_to\_square(self, vector, expected):

        self.assertEqual(

            Vector(\*vector).to\_square(),

            expected

        )

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

    unittest.main()

Here are the test data files:

**from\_square.yaml**

test1:

  square: 'G3'

  expected\_vector: [6, 2]

test2:

  square: 'A8'

  expected\_vector: [0, 7]

test3:

  square: 'H1'

  expected\_vector: [7, 0]

test4:

  square: 'C4'

  expected\_vector: [2, 3]

test5:

  square: 'F6'

  expected\_vector: [5, 5]

# # invalid

# test6:

#   square: 'A5'

#   expected\_vector: [5, 5]

**Vector\_add.yaml**

test1:

  vector\_1: [1, 4]

  vector\_2: [4, 6]

  expected\_vector: [5, 10]

test2:

  vector\_1: [1, 7]

  vector\_2: [4, 6]

  expected\_vector: [5, 13]

test3:

  vector\_1: [1, 3]

  vector\_2: [0, 6]

  expected\_vector: [1, 9]

test4:

  vector\_1: [-1, 32]

  vector\_2: [3, 6]

  expected\_vector: [2, 38]

test5:

  vector\_1: [6, 0]

  vector\_2: [6, 7]

  expected\_vector: [12, 7]

# # invalid delete me

# test6:

#   vector\_1: [6, 0]

#   vector\_2: [6, 7]

#   expected\_vector: [0, 7]

**Vector\_in\_board.yaml**

test1:

  vector: [0, 0]

  expected: True

test2:

  vector: [7, 0]

  expected: True

test3:

  vector: [0, 7]

  expected: True

test4:

  vector: [7, 7]

  expected: True

test5:

  vector: [4, 6]

  expected: True

test6:

  vector: [3, 2]

  expected: True

test7:

  vector: [-1, -1]

  expected: False

test8:

  vector: [-5, 4]

  expected: False

test9:

  vector: [8, 7]

  expected: False

test10:

  vector: [-4, 4]

  expected: False

test11:

  vector: [10, 4]

  expected: False

# adding comment inside causes logic error and false pass on test

# # invalid delete me

# test:

#   vector: [4, 6]

#   expected: True

**Vector\_multiply.yaml**

test1:

  vector: [1, 1]

  multiplier: 1

  expected: [1, 1]

test2:

  vector: [1, 1]

  multiplier: -1

  expected: [-1, -1]

test3:

  vector: [1, 1]

  multiplier: 5

  expected: [5, 5]

test4:

  vector: [5, 0]

  multiplier: 0

  expected: [0, 0]

**vector\_to\_square.yaml**

test1:

  vector: [0, 0]

  expected: A1

test2:

  vector: [5, 7]

  expected: F8

test3:

  vector: [6, 2]

  expected: G3

test4:

  vector: [0, 6]

  expected: A7

test5:

  vector: [7, 7]

  expected: H8

as can be seen with the console:

Graphical user interface, text

Description automatically generated

The decorators I added to my tests are unusual as they return many mutations of my original generic function that complete a specific test from my yaml file. This allows me to perform targeted testing

Text

Description automatically generated

As is shown above, I can run all test, just vector tests or a specific test form a yaml file. I know the tests are working as I can change some of the tests data to create a test that I expect to fail.

For example:

# adding comment inside causes logic error and false pass on test

# invalid delete me

test:

  vector: [4, 6]

  expected: False

I have uncommented out this invalid test in the **vector\_in\_board.yaml** file

When I run the tests:

Text

Description automatically generated

There is a summary of which of the tests fail and a traceback, including a message to help determine why one of the tests failed.

As part of testing my tests in this way I realised that my tests weren’t working. I determined that this was because I had names them all test in my yaml file. To fix this I named them test1, test2, ect.

I think it is important to document tests and code at the same time as they were developed in tandem. I didn’t move to code the next stage until all of my vector untit tests were working. This was valuable as it allowed me to use the vector module with the assumption that it was working perfectly in unit tests and code for later modules that built upon my vectors module.

The next logic to code was the various rules for how pieces could move on a chess board. To do this I created a pieces module.

I wanted to make a class for each piece to describe its value (needed for the utility function) and the ways in which is can move within a chess board. As I knew that some logic would be repeated withing the piece classes I decided to have a parent class piece and a series of child classes that inherit form it. Since I also knew that I would want to do general operations on a set of pieces later on (such as iterate through them for movement vectors or to sum up their value) I decided to use an abstract base class. This ensures that all of the piece classes have some key attributes and methods (they share the same interface). This means that I won’t need to differentiate between pieces and have different logic for each one (with the exception of the king).

I found a series of standard values and matrices for determining addition value based on location within the board on the following website

<https://www.chessprogramming.org/Simplified_Evaluation_Function>

They were not in a useful form (e.g. csv download):

Calendar

Description automatically generated

To solve this I used a jupyter notebook to process them

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated(repeat for all pieces)

I then wrote them to a json file for future use.

Text

Description automatically generated

With this strategy for valuing a piece and my testes vector library I created my pieces library

**pieces.py**

# to do: make value and value matrix unchangeable of private

# https://stackoverflow.com/questions/31457855/cant-instantiate-abstract-class-with-abstract-methods

# import libraries and other local modules

import abc

from itertools import chain as iter\_chain, product as iter\_product

from typing import Callable

from vector import Vector

from assorted import ARBITRARILY\_LARGE\_VALUE

# Here is an abstract base class for piece,

# it dictates that all child object have the specified abstract attributes else and error will occur

# this ensures that all piece objects have the same interface

class Piece(abc.ABC):

    # required

    # color is public

    color: str | None

    # value and value matrix is protected

    # value is inherent value

    \_value: int

    # value matrix is additional value based on location

    \_value\_matrix: tuple[tuple[float]]

    # must be given before init

    @abc.abstractproperty

    def \_value\_matrix(): pass

    @abc.abstractproperty

    def \_value(): pass

    @abc.abstractproperty

    def color(): pass

    @abc.abstractmethod

    def symbol(self) -> str:

        """uses color to determine the appropriate symbol"""

    # not needed as abstract method as come classes will nor override

    def \_\_init\_\_(self, color):

        self.color = color

        self.last\_move = None

    # this should use the position vector and value matrix to get the value of the piece

    def get\_value(self, position\_vector: Vector):

        # flip if black as matrices are all for white pieces

        if self.color == "W":

            row, column = 7-position\_vector.j, position\_vector.i

        else:

            row, column = position\_vector.j, position\_vector.i

        # return sum of inherent value + value relative to positon on board

        return self.\_value + self.\_value\_matrix[row][column]

    # this function should yield all the movement vector tha the piece can move by

    # this doesn't account for check and is based on rules specific to each piece as well an checking if a vector is outside the board

    @abc.abstractmethod

    def generate\_movement\_vectors(self, pieces\_matrix, position\_vector):

        pass

    # when str(piece) called give the symbol

    def \_\_str\_\_(self):

        # return f"{self.color}{self.symbol}"

        return self.symbol()

    # standard repr method

    def \_\_repr\_\_(self):

        return f"{type(self).\_\_name\_\_}(color='{self.color}')"

    # logic that would be otherwise repeated in many of the child classes

    # determines the contents of a given square

    def square\_contains(self, square):

        """returns 'enemy' 'ally' or None for empty"""

        # check if empty

        if square is None:

            return "empty"

        # else the square must contain a piece, so examine its color

        if square.color == self.color:

            return "ally"

        else:

            return "enemy"

    # again reduces repeated logic

    # checks the result of a position vector

    # if not illegal (out of board) the square contents is returned

    def examine\_position\_vector(self, position\_vector: Vector, pieces\_matrix):

        """returns 'enemy' 'ally' 'empty' or 'illegal'  """

        # check if the vector is out of the board

        if not position\_vector.in\_board():

            return 'illegal'

        # for the rest of the code I can assume the vector is in board

        # else get the square at that vector

        row, column = 7-position\_vector.j, position\_vector.i

        square = pieces\_matrix[row][column]

        # examine its contents

        return self.square\_contains(square)

    # equality operator

    def \_\_eq\_\_(self, other):

        try:

            # assert same subclass like rook

            assert isinstance(other, type(self))

            assert self.color == other.color

            # i am not checking that pieces had the same last move as I want to compare kings without for the check function

            # assert self.last\_move == other.last\_move

            # value and value\_matrix should never be changes

        except AssertionError:

            return False

        else:

            return True

    # making pieces hashable allows for pieces matrices to be hashed and allows for pieces and pieces matrices to be put in sets,

    # also essential for piping data between python interpreter instances (different threads) for multitasking

    def \_\_hash\_\_(self):

        return hash((self.symbol(), self.color, self.last\_move))

# this class inherits from Piece an so it inherits some logic and some requirements as to how its interface should be

# as many child classes are similar I will explain this one in depth and then only explain notable features of others

class Pawn(Piece):

    # defining abstract properties, needed before init

    \_value = 100

    \_value\_matrix: tuple[tuple[float]] = [

        [0, 0, 0, 0, 0, 0, 0, 0],

        [50, 50, 50, 50, 50, 50, 50, 50],

        [10, 10, 20, 30, 30, 20, 10, 10],

        [5, 5, 10, 25, 25, 10, 5, 5],

        [0, 0, 0, 20, 20, 0, 0, 0],

        [5, -5, -10, 0, 0, -10, -5, 5],

        [5, 10, 10, -20, -20, 10, 10, 5],

        [0, 0, 0, 0, 0, 0, 0, 0]

    ]

    color = None

    # define symbol method (str method)

    def symbol(self): return f"{self.color}P"

    # override init constructor

    def \_\_init\_\_(self, color):

        # perform super's instructor

        super().\_\_init\_\_(color)

        # but in addition...

        # decide the vectors that the piece can move now as it is based on color

        multiplier = 1 if color == "W" else -1

        # method defined here as it only used here

        # decided if the pawn is allowed to move foreward 2 based on square contents and last move

        def can\_move\_foreword\_2(square):

            return square is None and self.last\_move is None

        # tuple contains pairs of vector and contition that must be met

        # (in the form of a function that takes square and returns a boolean)

        self.movement\_vector\_and\_condition: tuple[Vector, Callable] = (

            # v(0, 1) for foreword

            (Vector(0, multiplier), lambda square: self.square\_contains(square) == "empty"),

            # v(0, 2) for foreword as first move

            (Vector(0, 2\*multiplier), can\_move\_foreword\_2),

            # v(-1, 1) and v(1, 1) for take

            (Vector(1, multiplier), lambda square: self.square\_contains(square) == 'enemy'),

            (Vector(-1, multiplier), lambda square: self.square\_contains(square) == 'enemy'),

        )

    # generate movement vectors

    def generate\_movement\_vectors(self, pieces\_matrix, position\_vector):

        # iterate through movement vectors and conditions

        for movement\_vector, condition in self.movement\_vector\_and\_condition:

            # get resultant vector

            resultant\_vector = position\_vector + movement\_vector

            # if vector\_out of range continue

            if not resultant\_vector.in\_board():

                continue

            # get the contents of the square corresponding to the resultant

            row, column = 7-resultant\_vector.j, resultant\_vector.i

            piece = pieces\_matrix[row][column]

            # if the condition is met, yield the vector

            if condition(piece):

                yield movement\_vector

class Knight(Piece):

    \_value = 320

    \_value\_matrix: tuple[tuple[float]] = [

        [-50, -40, -30, -30, -30, -30, -40, -50],

        [-40, -20, 0, 0, 0, 0, -20, -40],

        [-30, 0, 10, 15, 15, 10, 0, -30],

        [-30, 5, 15, 20, 20, 15, 5, -30],

        [-30, 0, 15, 20, 20, 15, 0, -30],

        [-30, 5, 10, 15, 15, 10, 5, -30],

        [-40, -20, 0, 5, 5, 0, -20, -40],

        [-50, -40, -30, -30, -30, -30, -40, -50]

    ]

    color = None

    # n for knight as king takes k

    def symbol(self): return f"{self.color}N"

    def generate\_movement\_vectors(self, pieces\_matrix, position\_vector):

        # this function yields all 8 possible vectirs

        def possible\_movement\_vectors():

            vectors = (Vector(2, 1), Vector(1, 2))

            # for each x multiplier, y multiplier and vector combination

            for i\_multiplier, j\_multiplier, vector in iter\_product((-1, 1), (-1, 1), vectors):

                # yield corresponding vector

                yield Vector(

                    vector.i \* i\_multiplier,

                    vector.j \* j\_multiplier

                )

        # iterate through movement vectors

        for movement\_vector in possible\_movement\_vectors():

            # get resultant

            resultant\_vector = position\_vector + movement\_vector

            # look at contents of square

            contents = self.examine\_position\_vector(position\_vector=resultant\_vector, pieces\_matrix=pieces\_matrix)

            # if square is empty yield vector

            if contents == "empty":

                yield movement\_vector

class Bishop(Piece):

    \_value = 330

    \_value\_matrix: tuple[tuple[float]] = [

        [-20, -10, -10, -10, -10, -10, -10, -20],

        [-10, 0, 0, 0, 0, 0, 0, -10],

        [-10, 0, 5, 10, 10, 5, 0, -10],

        [-10, 5, 5, 10, 10, 5, 5, -10],

        [-10, 0, 10, 10, 10, 10, 0, -10],

        [-10, 10, 10, 10, 10, 10, 10, -10],

        [-10, 5, 0, 0, 0, 0, 5, -10],

        [-20, -10, -10, -10, -10, -10, -10, -20]

    ]

    color = None

    def symbol(self): return f"{self.color}B"

    def generate\_movement\_vectors(self, pieces\_matrix, position\_vector):

        # sourcery skip: use-itertools-product

        # repeat for all 4 vector directions

        for i, j in iter\_product((1, -1), (1, -1)):

            unit\_vector = Vector(i, j)

            # iterate through length multipliers

            for multiplier in range(1, 8):

                # get movement and resultant vectors

                movement\_vector = unit\_vector \* multiplier

                resultant\_vector = position\_vector + movement\_vector

                # examine the contents of the square and use switch case to decide behaviour

                match self.examine\_position\_vector(position\_vector=resultant\_vector, pieces\_matrix=pieces\_matrix):

                    case 'illegal':

                        # if vector extends out of the board stop extending

                        break

                    case 'ally':

                        # break of of for loop (not just match case)

                        # as cannot hop over piece so don't explore longer vectors in same direction

                        break

                    case 'enemy':

                        # this is a valid move

                        yield movement\_vector

                        # break of of for loop (not just match case)

                        # as cannot hop over piece so don't explore longer vectors in same direction

                        break

                    case 'empty':

                        # is valid

                        yield movement\_vector

                        # and keep exploring, don't break

class Rook(Piece):

    \_value = 500

    \_value\_matrix: tuple[tuple[float]] = [

        [0, 0, 0, 0, 0, 0, 0, 0],

        [5, 10, 10, 10, 10, 10, 10, 5],

        [-5, 0, 0, 0, 0, 0, 0, -5],

        [-5, 0, 0, 0, 0, 0, 0, -5],

        [-5, 0, 0, 0, 0, 0, 0, -5],

        [-5, 0, 0, 0, 0, 0, 0, -5],

        [-5, 0, 0, 0, 0, 0, 0, -5],

        [0, 0, 0, 5, 5, 0, 0, 0]

    ]

    color = None

    # r for rook

    def symbol(self): return f"{self.color}R"

    # this code is very similar in structure to that of a bishop just with different direction vectors

    def generate\_movement\_vectors(self, pieces\_matrix, position\_vector):

        # sourcery skip: use-itertools-product

        unit\_vectors = (

            Vector(0, 1),

            Vector(0, -1),

            Vector(1, 0),

            Vector(-1, 0),

        )

        # for unit\_vector, multiplier in iter\_product(unit\_vectors, range(1, 8)):

        for unit\_vector in unit\_vectors:

            for multiplier in range(1, 8):

                movement\_vector = unit\_vector \* multiplier

                resultant\_vector = position\_vector + movement\_vector

                # note cases that contain only break are not redundant, they break the outer for loop

                match self.examine\_position\_vector(position\_vector=resultant\_vector, pieces\_matrix=pieces\_matrix):

                    case 'illegal':

                        # if vector extends out of the board stop extending

                        break

                    case 'ally':

                        # break of of for loop (not just match case)

                        # as cannot hop over piece so don't explore longer vectors in same direction

                        break

                    case 'enemy':

                        # this is a valid move

                        yield movement\_vector

                        # break of of for loop (not just match case)

                        # as cannot hop over piece so don't explore longer vectors in same direction

                        break

                    case 'empty':

                        # is valid

                        yield movement\_vector

                        # and keep exploring, don't break

class Queen(Piece):

    \_value = 900

    \_value\_matrix: tuple[tuple[float]] = [

        [-20, -10, -10, -5, -5, -10, -10, -20],

        [-10, 0, 0, 0, 0, 0, 0, -10],

        [-10, 0, 5, 5, 5, 5, 0, -10],

        [-5, 0, 5, 5, 5, 5, 0, -5],

        [0, 0, 5, 5, 5, 5, 0, -5],

        [-10, 5, 5, 5, 5, 5, 0, -10],

        [-10, 0, 5, 0, 0, 0, 0, -10],

        [-20, -10, -10, -5, -5, -10, -10, -20]

    ]

    color = None

    def symbol(self): return f"{self.color}Q"

    # this code also uses a similar structure to the rook or bishop

    def generate\_movement\_vectors(self, pieces\_matrix, position\_vector):

        # unit\_vectors = (Vector(i, j) for i, j in iter\_product((-1, 0, 1), (-1, 0, 1)) if i != 0 and j != 0)

        unit\_vectors = (Vector(i, j) for i, j in iter\_product((-1, 0, 1), (-1, 0, 1)) if i != 0 or j != 0)

        # for unit\_vector, multiplier in iter\_product(unit\_vectors, range(1, 8)):

        for unit\_vector in unit\_vectors:

            for multiplier in range(1, 8):

                movement\_vector = unit\_vector \* multiplier

                resultant\_vector = position\_vector + movement\_vector

                match self.examine\_position\_vector(position\_vector=resultant\_vector, pieces\_matrix=pieces\_matrix):

                    case 'illegal':

                        # if vector extends out of the board stop extending

                        break

                    case 'ally':

                        # break of of for loop (not just match case)

                        # as cannot hop over piece so don't explore longer vectors in same direction

                        break

                    case 'enemy':

                        # this is a valid move

                        yield movement\_vector

                        # break of of for loop (not just match case)

                        # as cannot hop over piece so don't explore longer vectors in same direction

                        break

                    case 'empty':

                        # is valid

                        yield movement\_vector

                        # and keep exploring, don't break

class King(Piece):

    # not needed as static eval does't add up kings value

    \_value = ARBITRARILY\_LARGE\_VALUE

    # there are 2 matrices to represent the early and late game for the king

    value\_matrix\_early: tuple[tuple[float]] = [

        [-30, -40, -40, -50, -50, -40, -40, -30],

        [-30, -40, -40, -50, -50, -40, -40, -30],

        [-30, -40, -40, -50, -50, -40, -40, -30],

        [-30, -40, -40, -50, -50, -40, -40, -30],

        [-20, -30, -30, -40, -40, -30, -30, -20],

        [-10, -20, -20, -20, -20, -20, -20, -10],

        [20, 20, 0, 0, 0, 0, 20, 20],

        [20, 30, 10, 0, 0, 10, 30, 20]

    ]

    value\_matrix\_late = [

        [-50, -40, -30, -20, -20, -30, -40, -50],

        [-30, -20, -10, 0, 0, -10, -20, -30],

        [-30, -10, 20, 30, 30, 20, -10, -30],

        [-30, -10, 30, 40, 40, 30, -10, -30],

        [-30, -10, 30, 40, 40, 30, -10, -30],

        [-30, -10, 20, 30, 30, 20, -10, -30],

        [-30, -30, 0, 0, 0, 0, -30, -30],

        [-50, -30, -30, -30, -30, -30, -30, -50]

    ]

    color = None

    # initially value matrix is the early one

    \_value\_matrix: tuple[tuple[float]] = value\_matrix\_early

    # def \_\_init\_\_(self, \*args, \*\*kwargs):

    #     self.\_value\_matrix = self.value\_matrix\_early

    #     super().\_\_init\_\_(self, \*args, \*\*kwargs)

    def symbol(self): return f"{self.color}K"

    # based on total pieces, changes the value matrix

    def update\_value\_matrix(self, pieces\_matrix):

        # counts each empty square as 0 and each full one as 1 then sums them up to get total pieces

        # if total pieces less than or equal to 10: then late game

        if sum(int(isinstance(square, Piece)) for square in iter\_chain(pieces\_matrix)) <= 10:

            self.value\_matrix = self.value\_matrix\_late

        # else early game

        else:

            self.value\_matrix = self.value\_matrix\_early

    # this generates the movement vectors for the king

    def generate\_movement\_vectors(self, pieces\_matrix, position\_vector):

        # take the opportunity to update the value matrix

        self.update\_value\_matrix(pieces\_matrix)

        # all 8 movement vectors

        unit\_vectors = (Vector(i, j) for i, j in iter\_product((-1, 0, 1), (-1, 0, 1)) if i != 0 or j != 0)

        # for each movement vector, get the resultant vector

        for movement\_vector in unit\_vectors:

            resultant\_vector = position\_vector + movement\_vector

            # examine contents of square and use switch case to decide behaviour

            match self.examine\_position\_vector(position\_vector=resultant\_vector, pieces\_matrix=pieces\_matrix):

                case 'illegal':

                    continue

                case 'ally':

                    continue

                case 'enemy':

                    # this is a valid move

                    yield movement\_vector

                case 'empty':

                    # is valid

                    yield movement\_vector

# used by other modules to convert symbol to piece

PIECE\_TYPES = {

    'P': Pawn,

    'N': Knight,

    'B': Bishop,

    'R': Rook,

    'K': King,

    'Q': Queen

}

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

# ensures that all classes are valid (not missing any abstract properties) whenever the module is imported

Pawn('W')

Knight('W')

Bishop('W')

Rook('W')

Queen('W')

King('W')

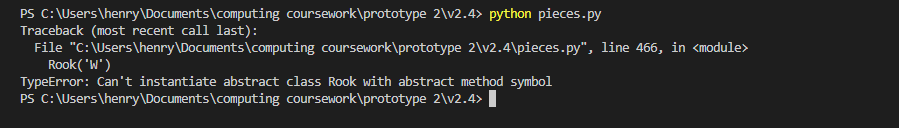
I can remove one of the required abstract methods form the Rook class to show you the error this causes.

Change

    # # r for rook

    # def symbol(self): return f"{self.color}R"

Result



This behaviour is valid as it means that, if I am sure that a variable contains an object of type Piece, not matter which piece, I can be sure that the interface will be the same (e.g. I can assume they all have a symbol method).

For the King class I gave it a value which is arbitrarily high. This comes from a file called **assorted.py**

# this file is just a file of short assorted constants and functions that are general in use

# It only contains small functions as I have tried to group large, similar functions logically in there own file

# this is used in the static evaluation and minimax process. It is used to represent infinity in a way that still allows comparrison

ARBITRARILY\_LARGE\_VALUE = 1\_000\_000

# this function is relatively redundant but allows for print statements in debugging

# in later iteration this may be replaced with logging.

# it is useful as it allows for DEBUG print statements without needing to remove them when finished

DEBUG = True

def dev\_print(\*args, \*\*kwargs):

    if DEBUG:

        print(\*args, \*\*kwargs)

# this is an exception that allows for the game data to be bound to it

# this allows for the relevant chess game that caused the error to be examined afterwards

# it is a normal exception except the constructor has been modified to save the game data as a property

class \_\_ChessExceptionTemplate\_\_(Exception):

    def \_\_init\_\_(self, \*args, \*\*kwargs) -> None:

        # none if key not present

        self.game = kwargs.pop("game", None)

        super().\_\_init\_\_(\*args, \*\*kwargs)

# these are custom exceptions.

# they contain no logic but have distinct types allowing for targeted error handling

class InvalidMove(\_\_ChessExceptionTemplate\_\_):

    pass

class NotUserTurn(\_\_ChessExceptionTemplate\_\_):

    pass

The kings inherent value isn’t relevant as both players will always have a king and so the value will always cancel out. I also used 2 value matrices for the king. I determined that the late game is when there are 10 or less total pieces though this is arbitrary.

The testing for the pieces module was as follows:

**Test\_pieces.py**

import unittest

import ddt

import pieces

# as vector is already tested we can use it here and assume it won't cause any logic errors

from vector import Vector

def test\_dir(file\_name): return f"test\_data/pieces/{file\_name}.yaml"

EMPTY\_PIECES\_MATRIX = ((None,)\*8,)\*8

@ddt.ddt

class Test\_Case(unittest.TestCase):

    # this test is based on testing the the movement vectors of a piece places at some position in an empty board are as expected

    @ddt.file\_data(test\_dir('test\_empty\_board'))

    def test\_empty\_board(self, piece\_type, square, expected\_move\_squares):

        # deserialize

        position\_vector = Vector.construct\_from\_square(square)

        piece: pieces.Piece = pieces.PIECE\_TYPES[piece\_type]('W')

        movement\_vectors = piece.generate\_movement\_vectors(pieces\_matrix=EMPTY\_PIECES\_MATRIX, position\_vector=position\_vector)

        resultant\_squares = list(

            map(

                lambda movement\_vector: (movement\_vector+position\_vector).to\_square(),

                movement\_vectors

            )

        )

        # assert as expected

        # can use sets to prevent order being an issue as vectors are hashable

        self.assertEqual(

            set(resultant\_squares),

            set(expected\_move\_squares),

            msg=f"\n\nactual movement squares {sorted(resultant\_squares)}  !=  expected movement squares {sorted(expected\_move\_squares)}\n{repr(piece)} at {square}"

        )

    # this test assert that that a pieces movement vectors are as expected when the piece is surrounded by other pieces

    @ddt.file\_data(test\_dir('test\_board\_populated'))

    def test\_board\_populated(self, pieces\_matrix, square, expected\_piece\_symbol, expected\_move\_squares):

        # deserialize

        def list\_map(function, iterable): return list(map(function, iterable))

        def descriptor\_to\_piece(descriptor) -> pieces.Piece:

            # converts WN to knight object with a color attribute of white

            if descriptor is None:

                return None

            color, symbol  = descriptor

            piece\_type: pieces.Piece = pieces.PIECE\_TYPES[symbol]

            return piece\_type(color=color)

        def row\_of\_symbols\_to\_pieces(row): return list\_map(descriptor\_to\_piece, row)

        # update pieces\_matrix replacing piece descriptors to piece objects

        pieces\_matrix = list\_map(row\_of\_symbols\_to\_pieces, pieces\_matrix)

        position\_vector = Vector.construct\_from\_square(square)

        row, column = 7-position\_vector.j, position\_vector.i

        # assert piece as expected

        piece: pieces.Piece = pieces\_matrix[row][column]

        self.assertEqual(

            piece.symbol(),

            expected\_piece\_symbol,

            msg=f"\nPiece at square {square} was not the expected piece"

        )

        # assert movement vectors as expected

        movement\_vectors = piece.generate\_movement\_vectors(pieces\_matrix=pieces\_matrix, position\_vector=position\_vector)

        resultant\_squares = list(

            map(

                lambda movement\_vector: (movement\_vector+position\_vector).to\_square(),

                movement\_vectors

            )

        )

        self.assertEqual(

            set(resultant\_squares),

            set(expected\_move\_squares),

                msg=f"\n\nactual movement squares {sorted(resultant\_squares)}  !=  expected movement squares {sorted(expected\_move\_squares)}\n{repr(piece)} at {square}"

        )

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

    unittest.main()

**test\_board\_populated.yaml**

test1:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [BP,   null, BP,   null, null, null, null, null],

    [null, WP,   null, null, null, null, null, null]

  ]

  square: B1

  expected\_piece\_symbol: WP

  expected\_move\_squares: [

    A2, C2, B2, B3

  ]

test2:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, BP,   null, null, null, null],

    [null, null, null, null, BR,   null, null, null],

    [null, BR,   null, BQ,   null, null, WB,   null],

    [null, null, BP,   null, null, null, BP,   null],

    [null, WR,   null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null]

  ]

  square: D4

  expected\_piece\_symbol: BQ

  expected\_move\_squares: [

    D5,

    E4, F4, G4,

    E3, F2, G1,

    D3, D2, D1,

    C4,

    C5, B6, A7

  ]

test3:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, BP,   null, null, null, null],

    [null, null, null, null, BR,   null, null, null],

    [null, BR,   null, BR,   null, null, WB,   null],

    [null, null, BP,   null, null, null, BP,   null],

    [null, WR,   null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null]

  ]

  square: D4

  expected\_piece\_symbol: BR

  expected\_move\_squares: [

    D5,

    E4, F4, G4,

    D3, D2, D1,

    C4,

  ]

test4:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, BP, null, null, null, null],

    [null, null, WP, null, WP, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null]

  ]

  square: D7

  expected\_piece\_symbol: BP

  expected\_move\_squares: [

    D6, D5,

    C6, E6

  ]

test5:

  pieces\_matrix: [

    [null, null, null, BK, null, null, null, null],

    [null, null, null, WP, null, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  square: D8

  expected\_piece\_symbol: BK

  expected\_move\_squares: [

    C7, D7, E7,

    C8, E8

  ]

test6:

  pieces\_matrix: [

    [BR,   BK,   BB,   BK,   BQ,   BB,   BK,   BR  ],

    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],

    [WR,   WN,   WB,   WK,   WQ,   WB,   WN,   WR  ]

  ]

  square: B1

  expected\_piece\_symbol: WN

  expected\_move\_squares: [A3, C3]

test7:

  pieces\_matrix: [

    [BR,   BN,   BB,   BK,   BQ,   BB,   BN,   BR  ],

    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],

    [WR,   WN,   WB,   WK,   WQ,   WB,   WN,   WR  ]

  ]

  square: G1

  expected\_piece\_symbol: WN

  expected\_move\_squares: [F3, H3]

**test\_empty\_board.yaml**

test1:

  piece\_type: 'N'

  square: E4

  expected\_move\_squares: [

    D2, F2, D6, F6, C5, C3, G5, G3

  ]

test2:

  piece\_type: Q

  square: C3

  expected\_move\_squares: [

    A1, B2, D4, E5, F6, G7, H8,

    C1, C2, C4, C5, C6, C7, C8,

    A3, B3, D3, E3, F3, G3, H3,

    A5, B4, D2, E1

  ]

test3:

  piece\_type: K

  square: B2

  expected\_move\_squares: [

    C1, C2, C3,

    B1, B3,

    A1, A2, A3,

  ]

test4:

  piece\_type: P

  square: E3

  expected\_move\_squares: [

    E4, E5

  ]

test5:

  piece\_type: R

  square: F6

  expected\_move\_squares: [

    F1, F2, F3, F4, F5, F7, F8,

    A6, B6, C6, D6, E6, G6, H6

  ]

test6:

  piece\_type: B

  square: D7

  expected\_move\_squares: [

    C8, E6, F5, G4, H3,

    E8, C6, B5, A4,

  ]

test7:

  piece\_type: K

  square: D8

  expected\_move\_squares: [

    C7, D7, E7,

    C8, E8

  ]

These tests are explained by the comments. They focus on testing the movement of the pieces (needed for the legal moves function).

This sub problem of deciding where a piece could move was made easier by abstracting away complications such as how check affects the legal moves. This additional logic will be added on by later modules that will rely on the pieces module. I wrote the tests as I coded the pieces module and so I was able to detect and fix issues as I developed the module until and the tests needed had been written and were working.

Evidence:

Text

Description automatically generated

There was, in hind sight a flaw with my testing of this module which I will get to later (mirroring where I was in my development journey when I discovered it).

Next I created a board state class. This class was to be responsible for keeping track of a single board state: a snapshot in the whole game. It would include a utility function, a move executer function and a generate legal moves function. It would be immutable. This was because I knew that the minimax function would want to make many different combinations of moves on a given board state without irreversibly mutating the object or needing to make a deep copy.

Here is the class I came up with:

**Board\_state.py**

from copy import deepcopy

from dataclasses import dataclass

from itertools import product as iter\_product

import pieces as pieces\_mod

from assorted import ARBITRARILY\_LARGE\_VALUE

from vector import Vector

# this is a pieces matrix for the starting position is chess

# white is at the bottom as it is from the user's perspective and I am currently assuming the user is white.

# I can change this in the frontend later

STARTING\_POSITIONS: tuple[tuple[pieces\_mod.Piece]] = (

    (

        pieces\_mod.Rook(color="B"),

        pieces\_mod.Knight(color="B"),

        pieces\_mod.Bishop(color="B"),

        pieces\_mod.Queen(color="B"),

        pieces\_mod.King(color="B"),

        pieces\_mod.Bishop(color="B"),

        pieces\_mod.Knight(color="B"),

        pieces\_mod.Rook(color="B")

    ),

    (pieces\_mod.Pawn(color="B"),)\*8,

    (None,)\*8,

    (None,)\*8,

    (None,)\*8,

    (None,)\*8,

    (pieces\_mod.Pawn(color="W"),)\*8,

    (

        pieces\_mod.Rook(color="W"),

        pieces\_mod.Knight(color="W"),

        pieces\_mod.Bishop(color="W"),

        pieces\_mod.Queen(color="W"),

        pieces\_mod.King(color="W"),

        pieces\_mod.Bishop(color="W"),

        pieces\_mod.Knight(color="W"),

        pieces\_mod.Rook(color="W")

    )

)

# frozen makes each instance of the board\_state class is immutable

@dataclass(frozen=True)

class Board\_State():

    next\_to\_go: str = "W"

    # the pieces matrix keeps track of the board positions and the pieces

    pieces\_matrix: tuple[tuple[pieces\_mod.Piece]] = STARTING\_POSITIONS

    # this function outputs the board in a user friendly way

    # 8| BR  BN  BB  BQ  BK  BB  BN  BR

    # 7| ·   ·   BP  BP  BP  BP  BP  BP

    # 6| ·   ·   ·   ·   ·   ·   ·   ·

    # 5| BP  BP  ·   ·   ·   ·   ·   ·

    # 4| ·   ·   ·   WP  ·   WP  ·   ·

    # 3| ·   ·   ·   ·   ·   ·   ·   ·

    # 2| WP  WP  WP  ·   WP  ·   WP  WP

    # 1| WR  WN  WB  WQ  WK  WB  WN  WR

    #   (A   B   C   D   E   F   G   H )

    def print\_board(self):

        # convert each piece to a symbol BP and replace none with dots

        # add numbers to left and add letters at the bottom

        numbers = range(8, 0, -1)

        letters = [chr(i) for i in range(ord("A"), ord("H")+1)]

        for row, number in zip(self.pieces\_matrix, numbers):

            # clean up row of pieces

            symbols\_row = map(lambda piece: piece.symbol() if piece else "· ", row)

            pretty\_row = f"{number}| {'  '.join(symbols\_row)}"

            print(pretty\_row)

        print(f"  ({'   '.join(letters)} )")

    def get\_piece\_at\_vector(self, vector: Vector):

        # this function exists as it is a really common operation.

        # due to vector 0, 0 pointing the the bottom left not top left of the 2d array,

        # some correction is needed

        column, row = vector.i, 7-vector.j

        return self.pieces\_matrix[row][column]

    # this function should yield all the pieces on the board

    def generate\_all\_pieces(self):

        # nested loop for i and j to iterate through all possible vectors

        for i, j in iter\_product(range(8), range(8)):

            position\_vector = Vector(i,j)

            # get the contents of the corresponding square

            piece = self.get\_piece\_at\_vector(position\_vector)

            # skip if none: skip if square empty

            if piece:

                yield piece, position\_vector

    # this yields all pieces of a given color:

    # used when examining legal moves of a given player

    def generate\_pieces\_of\_color(self, color=None):

        # be default give pieces of player next to go

        if color is None:

            color = self.next\_to\_go

        # return all piece and position vector pairs

        # filtered by piece must match in color

        yield from filter(

            # lambda piece, \_: piece.color == color,

            lambda piece\_and\_vector: piece\_and\_vector[0].color == color,

            self.generate\_all\_pieces()

        )

    # determines if a given player is in check based on the player's color

    def color\_in\_check(self, color=None):

        # default is to check if the player next to go is in check

        if color is None:

            color = self.next\_to\_go

        # let is now be A's turn

        # I use this player a and b model to keep track of the logic here

        color\_a = color

        color\_b = "W" if color == "B" else "B"

        # we will examine all the movement vectors of B's pieces

        # if any of them could take the A's King then currently A is in check as their king is threatened by 1 or more pieces (which could take it next turn)

        # for each of b's pieces

        for piece, position\_v in self.generate\_pieces\_of\_color(color=color\_b):

            movement\_vs = piece.generate\_movement\_vectors(

                pieces\_matrix=self.pieces\_matrix,

                position\_vector=position\_v

            )

            # for each movement vector that the piece could make

            for movement\_v in movement\_vs:

                resultant = position\_v + movement\_v

                # check the contents of the square

                to\_square = self.get\_piece\_at\_vector(resultant)

                # As\_move\_threatens\_king\_A = isinstance(to\_square, pieces\_mod.King) and to\_square.color == color\_a

                # if the contents is A's king then the a is in check.

                As\_move\_threatens\_king\_A = (to\_square == pieces\_mod.King(color=color\_a))

                # if As\_move\_threatens\_king\_A then return true

                if As\_move\_threatens\_king\_A:

                    return True

        # if none of B's pieces were threatening to take A's king then A isn't in check

        return False

    # this function is a generator to be iterated through.

    # it is responsible for yielding every possible move that a given player can make

    # yields this as a position and a movement vector

    def generate\_legal\_moves(self):

        # iterate through all pieces belonging to player next to go

        for piece, piece\_position\_vector in self.generate\_pieces\_of\_color(color=self.next\_to\_go):

            movement\_vectors = piece.generate\_movement\_vectors(

                pieces\_matrix=self.pieces\_matrix,

                position\_vector=piece\_position\_vector

            )

            # iterate through movement vectors of this piece

            for movement\_vector in movement\_vectors:

                # examine resulting child game state

                child\_game\_state = self.make\_move(from\_position\_vector=piece\_position\_vector, movement\_vector=movement\_vector)

                # determine if current player next to go (different to next to go of child game state) is in check

                is\_check\_after\_move = child\_game\_state.color\_in\_check(color=self.next\_to\_go)

                # only yield the move if it doesn't result in check

                if not is\_check\_after\_move:

                    yield piece\_position\_vector, movement\_vector

    # determines if the game is over

    # returns over, winner

    def is\_game\_over\_for\_next\_to\_go(self):

        # sourcery skip: remove-unnecessary-else, swap-if-else-branches

        # in all cases, the game if over if a player has no legal moves left

        if not list(self.generate\_legal\_moves()):

            # if b in check

            if self.color\_in\_check():

                # checkmate for b, a wins

                winner = "W" if self.next\_to\_go == "B" else "B"

                return True, winner

            else:

                # stalemate

                return True, None

        # game not over

        return False, None

    # this function if responsible for generating a static evaluation for a given board-stat

    # it should be used by a maximiser or minimiser

    # starting position should have and evaluation of 0

    def static\_evaluation(self):

        # if over give an appropriate score for win loss or draw

        over, winner = self.is\_game\_over\_for\_next\_to\_go()

        if over:

            match winner:

                case None: multiplier = 0

                case "W": multiplier = 1

                case "B": multiplier = -1

            # return winner \* ARBITRARILY\_LARGE\_VALUE

            return multiplier \* ARBITRARILY\_LARGE\_VALUE

        # this function takes a piece as an argument and uses its color to decide if its value should be positive or negative

        def get\_piece\_value(piece: pieces\_mod.Piece, position\_vector: Vector):

            # this function assumes white it maximizer and so white pieces have a positive score and black negative

            match piece.color:

                case "W": multiplier = 1

                case "B": multiplier = (-1)

            value = multiplier \* piece.get\_value(position\_vector)

            # print(f"{piece.symbol()} at {position\_vector.to\_square} has value {value}")

            return value

        # for each piece, get the value (+/-)

        values = map(

            lambda x: get\_piece\_value(\*x),

            self.generate\_all\_pieces()

        )

        # sum values for static eval

        return sum(values)

    def make\_move(self, from\_position\_vector: Vector, movement\_vector: Vector):

        resultant\_vector = from\_position\_vector + movement\_vector

        # make a coppy of the position vector, deep coppy is used to ensure no parts are shared be reference

        new\_pieces\_matrix = deepcopy(self.pieces\_matrix)

        # convert to list

        new\_pieces\_matrix = list(map(list, new\_pieces\_matrix))

        # look at square with position vecotor

        row, col = 7-from\_position\_vector.j, from\_position\_vector.i

        # get piece thats moving

        piece: pieces\_mod.Piece = new\_pieces\_matrix[row][col]

        # set from square to blank

        new\_pieces\_matrix[row][col] = None

        # update piece to keep track of its last move

        piece.last\_move = movement\_vector

        # set to square to this piece

        row, col = 7-resultant\_vector.j, resultant\_vector.i

        new\_pieces\_matrix[row][col] = piece

        # convert back to tuple

        new\_pieces\_matrix = tuple(map(tuple, new\_pieces\_matrix))

        # update next to go

        new\_next\_to\_go = "W" if self.next\_to\_go == "B" else "B"

        # return new board state instance

        return Board\_State(

            next\_to\_go=new\_next\_to\_go,

            pieces\_matrix=new\_pieces\_matrix

        )

I wrote the functions and tested them in the order of dependency.

Starting with:

* Get\_piece\_at\_vecotor as it had not dependency to other functions

Then in order

* Generate\_all\_pieces
* Generate\_pieces\_of\_color
* Color\_in\_check
* Generate\_legal\_moves
* Game\_over
* Static\_Evaluation

I had a significant issue in testing this module, specifically the generate legal moves function and the static evaluation function. I will show all the testing code and data and then explain the issue.

**Test\_board\_state.py**

import unittest

import ddt

from board\_state import Board\_State

# tested so assumed correct

import pieces

from vector import Vector

def test\_dir(file\_name): return f"test\_data/board\_state/{file\_name}.yaml"

# code used to deserialize

# code repeated from test pieces, opportunity to reduce redundancy

def list\_map(function, iterable): return list(map(function, iterable))

def tuple\_map(function, iterable): return tuple(map(function, iterable))

def descriptor\_to\_piece(descriptor) -> pieces.Piece:

    # converts WN to knight object with a color attribute of white

    if descriptor is None:

        return None

    color, symbol = descriptor

    piece\_type: pieces.Piece = pieces.PIECE\_TYPES[symbol]

    return piece\_type(color=color)

def deserialize\_pieces\_matrix(pieces\_matrix, next\_to\_go="W") -> Board\_State:

    def row\_of\_symbols\_to\_pieces(row):

        return list\_map(descriptor\_to\_piece, row)

    # update pieces\_matrix replacing piece descriptors to piece objects

    pieces\_matrix = list\_map(row\_of\_symbols\_to\_pieces, pieces\_matrix)

    board\_state: Board\_State = Board\_State(pieces\_matrix=pieces\_matrix, next\_to\_go=next\_to\_go)

    return board\_state

@ddt.ddt

class Test\_Case(unittest.TestCase):

    # this test isn't data driven,

    # it tests that the static evaluation is 0 for starting positions

    def test\_static\_eval\_starting\_positions(self):

        self.assertEqual(

            Board\_State().static\_evaluation(),

            0

        )

    # this test is testing that a list of all pieces and there position vectors can be generated

    @ddt.file\_data(test\_dir('generate\_all\_pieces'))

    def test\_generate\_all\_pieces(self, pieces\_matrix, pieces\_and\_squares):

        pieces\_and\_squares = tuple\_map(tuple, pieces\_and\_squares)

        board\_state: Board\_State = deserialize\_pieces\_matrix(pieces\_matrix)

        # use set as order irrelevant

        all\_pieces: set[pieces.Piece, Vector] = set(board\_state.generate\_all\_pieces())

        # convert square to vector, allowed as this is tested

        def deserialize(data\_unit):

            # unpack test data unit

            descriptor, square = data\_unit

            # return [descriptor\_to\_piece(descriptor), Vector.construct\_from\_square(square)]

            return (descriptor\_to\_piece(descriptor), Vector.construct\_from\_square(square))

        def serialize(data\_unit):

            piece, vector = data\_unit

            # return [piece.symbol(), vector.to\_square()]

            return (piece.symbol(), vector.to\_square())

        all\_pieces\_expected: set[pieces.Piece, Vector] = set(map(deserialize, pieces\_and\_squares))

        # legal\_moves\_expected: list[pieces.Piece, Vector] = sorted(

        #     map(deserialize, pieces\_and\_squares),

        #     key=repr

        # )

        self.assertEqual(

            all\_pieces,

            all\_pieces\_expected,

            msg=f"\nactual {list\_map(serialize, all\_pieces)}  !=  expected {list\_map(serialize, all\_pieces\_expected)}"

        )

    # this test is to test the function responsible for getting the piece at a given position vector

    @ddt.file\_data(test\_dir('piece\_at\_vector'))

    def test\_piece\_at\_vector(self, pieces\_matrix, vectors\_and\_expected\_piece):

        board\_state: Board\_State = deserialize\_pieces\_matrix(pieces\_matrix)

        # could use all method and one assert but this would have been less readable, also hard to make useful message,

        # wanting different messages implies multiple asserts should be completed

        for vector, expected\_piece in vectors\_and\_expected\_piece:

            # deserialize vector / cast to Vector

            vector: Vector = Vector(\*vector)

            # repeat for piece

            expected\_piece: pieces.Piece = descriptor\_to\_piece(expected\_piece)

            # actual

            piece: piece.Piece = board\_state.get\_piece\_at\_vector(vector)

            msg = f"Piece at vector {repr(vector)} is {repr(piece)} not expected piece {repr(expected\_piece)}"

    # this test ensures that all the pieces belonging to a specific color and there position vectors can be identified

    @ddt.file\_data(test\_dir('generate\_pieces\_of\_color'))

    def test\_generate\_pieces\_of\_color(self, pieces\_matrix, color, pieces\_and\_squares):

        pieces\_and\_squares = tuple\_map(tuple, pieces\_and\_squares)

        board\_state: Board\_State = deserialize\_pieces\_matrix(pieces\_matrix)

        # use set as order irrelevant

        legal\_moves\_actual: set[pieces.Piece, Vector] = set(board\_state.generate\_pieces\_of\_color(color))

        # convert square to vector, allowed as this is tested

        def deserialize(data\_unit):

            # unpack test data unit

            descriptor, square = data\_unit

            # return [descriptor\_to\_piece(descriptor), Vector.construct\_from\_square(square)]

            return (descriptor\_to\_piece(descriptor), Vector.construct\_from\_square(square))

        def serialize(data\_unit):

            piece, vector = data\_unit

            # return [piece.symbol(), vector.to\_square()]

            return (piece.symbol(), vector.to\_square())

        legal\_moves\_expected: set[pieces.Piece, Vector] = set(map(deserialize, pieces\_and\_squares))

        self.assertEqual(

            legal\_moves\_actual,

            legal\_moves\_expected,

            msg=f"\nactual {list\_map(serialize, legal\_moves\_actual)}  !=  expected {list\_map(serialize, legal\_moves\_expected)}"

        )

    # this test ensures that the chess engine can determine if a specified player is currently in check

    @ddt.file\_data(test\_dir('color\_in\_check'))

    def test\_color\_in\_check(self, pieces\_matrix, white\_in\_check, black\_in\_check):

        board\_state: Board\_State = deserialize\_pieces\_matrix(pieces\_matrix)

        self.assertEqual(

            board\_state.color\_in\_check("W"),

            white\_in\_check,

            msg=f"white {'should' if white\_in\_check else 'should not'} be in check but {'is' if board\_state.color\_in\_check('W') else 'is not'}"

        )

        self.assertEqual(

            board\_state.color\_in\_check("B"),

            black\_in\_check,

            msg=f"black {'should' if black\_in\_check else 'should not'} be in check but {'is' if board\_state.color\_in\_check('B') else 'is not'}"

        )

    # this test ensures that a game over situation can be identified and its nature discerned

    @ddt.file\_data(test\_dir("game\_over"))

    def test\_game\_over(self, pieces\_matrix, expected\_over, expected\_outcome, next\_to\_go):

        board\_state: Board\_State = deserialize\_pieces\_matrix(pieces\_matrix, next\_to\_go=next\_to\_go)

        self.assertEqual(

            board\_state.is\_game\_over\_for\_next\_to\_go(),

            (expected\_over, expected\_outcome),

            msg=f"\nactual {board\_state.is\_game\_over\_for\_next\_to\_go()} != expected {(expected\_over, expected\_outcome)}"

        )

    # this is one of the most important functions to test

    # this function determines all the possible legal moves a player can make with their pieces, accounting for check

    # this test checks this for many inputs

    @ddt.file\_data(test\_dir("generate\_legal\_moves"))

    def test\_generate\_legal\_moves(self, pieces\_matrix, next\_to\_go, expected\_legal\_moves):

        board\_state: Board\_State = deserialize\_pieces\_matrix(pieces\_matrix=pieces\_matrix, next\_to\_go=next\_to\_go)

        actual\_legal\_moves = set(board\_state.generate\_legal\_moves())

        # convert square to vector, allowed as this is tested

        def deserialize\_expected\_legal\_moves():

            # unpack test data unit

            for test\_datum in expected\_legal\_moves:

                from\_square, all\_to\_squares = test\_datum

                for to\_square in all\_to\_squares:

                    position\_vector: Vector = Vector.construct\_from\_square(from\_square)

                    movement\_vector: Vector = Vector.construct\_from\_square(to\_square) - position\_vector

                    yield (position\_vector, movement\_vector)

        expected\_legal\_moves = set(deserialize\_expected\_legal\_moves())

        self.assertEqual(

            actual\_legal\_moves,

            expected\_legal\_moves,

        )

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

    unittest.main()

**Color\_in\_check.yaml**

test1:

  pieces\_matrix: [

    [BR,   BK,   BB,   BK,   BQ,   BB,   BK,   BR  ],

    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],

    [WR,   WK,   WB,   WK,   WQ,   WB,   WK,   WR  ]

  ]

  white\_in\_check: false

  black\_in\_check: false

test2:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, WK, WQ, BK],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  white\_in\_check: false

  black\_in\_check: true

test3:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, BQ],

    [null, null, null, null, null, null, null, null],

    [null, null, null, WP, WP, null, null, null],

    [null, null, null, WB, WK, WB, null, null],

  ]

  white\_in\_check: true

  black\_in\_check: false

test4:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, WK, null, BK],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, WR]

  ]

  white\_in\_check: false

  black\_in\_check: true

test5:

  pieces\_matrix: [

    [null, null, null, BK, null, null, null, null],

    [null, null, null, WP, null, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  white\_in\_check: false

  black\_in\_check: false

test6:

  pieces\_matrix: [

    [null, null, BK, null, null, null, null, null],

    [null, null, null, WP, null, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  white\_in\_check: false

  black\_in\_check: true

test7:

  pieces\_matrix: [

    [null, null, null, null, BK, null, null, null],

    [null, null, null, WP, null, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  white\_in\_check: false

  black\_in\_check: true

test8:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, WP, BK, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  white\_in\_check: true

  black\_in\_check: true

test9:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, BK, WP, null, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  white\_in\_check: true

  black\_in\_check: true

**Game\_over.yaml**

test1:

  next\_to\_go: W

  pieces\_matrix: [

    [BR,   BK,   BB,   BK,   BQ,   BB,   BK,   BR  ],

    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],

    [WR,   WK,   WB,   WK,   WQ,   WB,   WK,   WR  ]

  ]

  expected\_over: false

  expected\_outcome: null

test2:

  next\_to\_go: B

  pieces\_matrix: [

    [null, null, null, BK,   null, null, null, null],

    [null, null, null, WP,   null, null, null, null],

    [null, null, null, WK,   null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  expected\_over: true

  expected\_outcome: null

test3:

  next\_to\_go: B

  pieces\_matrix: [

    [WR,   null, null, null, null, null, BK,   null],

    [null, WR,   null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, WK  ],

  ]

  expected\_over: true

  expected\_outcome: W

test4:

  next\_to\_go: B

  pieces\_matrix: [

    [null, null, null, BK,   null, null, null, null],

    [null, null, null, WP,   null, null, null, null],

    [null, null, WP,   WK,   null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  expected\_over: true

  expected\_outcome: null

test5:

  next\_to\_go: B

  pieces\_matrix: [

    [null, null, null, BK,   null, null, null, null],

    [null, null, WP,   WP,   null, null, null, null],

    [null, null, null, WK,   null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  expected\_over: true

  expected\_outcome: W

test6:

  next\_to\_go: W

  pieces\_matrix: [

    [BR,   null, null, null, null, BR,   BK,   null],

    [BP,   null, null, null, null, BP,   BP,   null],

    [null, BP,   null, null, null, null, null, null],

    [null, null, BQ,   null, null, null, null, null],

    [null, null, null, null, null, null, null, WQ  ],

    [null, null, null, null, null, null, WP,   null],

    [WP,   null, null, null, null, WP,   WB,   null],

    [WR,   null, null, null, null, null, null, WR  ],

  ]

  expected\_over: false

  expected\_outcome: null

test7:

  next\_to\_go: B

  pieces\_matrix: [

    [BR,   null, null, null, null, BR,   BK,   WQ  ],

    [BP,   null, null, null, null, BP,   BP,   null],

    [null, BP,   null, null, null, null, null, null],

    [null, null, BQ,   null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, WP,   null],

    [WP,   null, null, null, null, WP,   WB,   null],

    [WR,   null, null, null, null, null, null, WR  ],

  ]

  expected\_over: true

  expected\_outcome: W

**generate\_all\_pieces.yaml**

test1:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [BP,   null, BP,   null, null, null, null, null],

    [null, WP,   null, null, null, null, null, null]

  ]

  pieces\_and\_squares: [

    [WP, B1],

    [BP, A2],

    [BP, C2]

  ]

test2:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, BP,   null, null, null, null],

    [null, null, null, null, BR,   null, null, null],

    [null, BR,   null, BR,   null, null, WB,   null],

    [null, null, BP,   null, null, null, BP,   null],

    [null, WR,   null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null]

  ]

  pieces\_and\_squares: [

    [WR, B2],

    [BR, B4],

    [BP, C3],

    [BR, D4],

    [BP, D6],

    [BP, G3],

    [WB, G4],

    [BR, E5]

  ]

**Generate\_legal\_moves.yaml**

test1:

  next\_to\_go: B

  pieces\_matrix: [

    [null, null, null, BK, null, null, null, null],

    [null, null, null, WP, null, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  expected\_legal\_moves: []

test2:

  next\_to\_go: W

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [BP,   null, BP,   null, null, null, null, null],

    [null, WP,   null, null, null, null, null, null]

  ]

  expected\_legal\_moves: [

    [B1, [B2, B3, A2, C2]]

  ]

test3:

  next\_to\_go: B

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [BP,   null, BP,   null, null, null, null, null],

    [null, WP,   null, null, null, null, null, null]

  ]

  expected\_legal\_moves: [

    [A2, [A1, B1]],

    [C2, [C1, B1]],

  ]

test4:

  next\_to\_go: W

  pieces\_matrix: [

    [BR,   BN,   BB,   BK,   BQ,   BB,   BN,   BR  ],

    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],

    [WR,   WN,   WB,   WK,   WQ,   WB,   WN,   WR  ]

  ]

  expected\_legal\_moves: [

    [A2, [A3, A4]],

    [B2, [B3, B4]],

    [C2, [C3, C4]],

    [D2, [D3, D4]],

    [E2, [E3, E4]],

    [F2, [F3, F4]],

    [G2, [G3, G4]],

    [H2, [H3, H4]],

    [B1, [A3, C3]],

    [G1, [F3, H3]]

  ]

test5:

  next\_to\_go: W

  pieces\_matrix: [

    [BK, BP, null, null, null, null, WP, WK],

    [BP, BP, null, null, null, null, WP, WP],

    [null, null, null, BP,   null, null, null, null],

    [null, null, null, null, BR,   null, null, null],

    [null, BR,   null, BR,   null, null, WB,   null],

    [null, null, BP,   null, null, null, BP,   null],

    [null, WN,   null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null]

  ]

  expected\_legal\_moves: [

    [B2, [A4, C4, D1, D3]],

    [G4, [

      H3, F5, E6, D7, C8,

      H5, F3, E2, D1

    ]],

  ]

**Generate\_pieces\_of\_color.yaml**

test1:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [BP,   null, BP,   null, null, null, null, null],

    [null, WP,   null, null, null, null, null, null]

  ]

  color: W

  pieces\_and\_squares: [

    [WP, B1]

  ]

test2:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [BP,   null, BP,   null, null, null, null, null],

    [null, WP,   null, null, null, null, null, null]

  ]

  color: B

  pieces\_and\_squares: [

    [BP, A2],

    [BP, C2]

  ]

test3:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, BP,   null, null, null, null],

    [null, null, null, null, BR,   null, null, null],

    [null, BR,   null, BR,   null, null, WB,   null],

    [null, null, BP,   null, null, null, BP,   null],

    [null, WR,   null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null]

  ]

  color: W

  pieces\_and\_squares: [

    [WR, B2],

    [WB, G4]

  ]

test4:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, BP,   null, null, null, null],

    [null, null, null, null, BR,   null, null, null],

    [null, BR,   null, BR,   null, null, WB,   null],

    [null, null, BP,   null, null, null, BP,   null],

    [null, WR,   null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null]

  ]

  color: B

  pieces\_and\_squares: [

    [BR, B4],

    [BP, C3],

    [BR, D4],

    [BP, D6],

    [BP, G3],

    [BR, E5]

  ]

**Piece\_at\_vector.yaml**

test1:

  pieces\_matrix: [

    [BR,   BN,   BB,   BK,   BQ,   BB,   BN,   BR  ],

    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],

    [WR,   WN,   WB,   WK,   WQ,   WB,   WN,   WR  ]

  ]

  vectors\_and\_expected\_piece: [

    [[0, 0], WR],

    [[1, 1], WP],

    [[2, 4], null],

    [[7, 3], null],

    [[5, 7], BB],

    [[2, 6], BP]

  ]

test2:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, BP,   null, null, null, null],

    [null, null, null, null, BR,   null, null, null],

    [null, BR,   null, BR,   null, null, WB,   null],

    [null, null, BP,   null, null, null, BP,   null],

    [null, WR,   null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null]

  ]

  vectors\_and\_expected\_piece: [

    [[0, 0], null],

    [[1, 1], WR],

    [[1, 3], BR],

    [[0, 7], null],

    [[7, 0], null],

    [[7, 7], null],

    [[3, 3], BR],

    [[3, 5], BP],

    [[6, 3], WB],

    [[6, 1], null]

  ]

The tests now work fine:

Text

Description automatically generated

But I was extremely stumped as I was testing my legal moves generator function when I got this issue:

Text

Description automatically generated

(other tests disables to show only this issue)

The test that had failed was (**generate\_legal\_moves.yaml**)

test1:

  next\_to\_go: B

  pieces\_matrix: [

    [null, null, null, BK, null, null, null, null],

    [null, null, null, WP, null, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  expected\_legal\_moves: []

There are supposed to be no legal moves as black in in checkmate. Instead the restriction of check was ignored and my function said that the black king could move to all 5 adjacent squares. I added this test to the check tests:

(**color\_in\_check.yaml**)

test5:

  pieces\_matrix: [

    [null, null, null, BK, null, null, null, null],

    [null, null, null, WP, null, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  white\_in\_check: false

  black\_in\_check: false

test6:

  pieces\_matrix: [

    [null, null, BK, null, null, null, null, null],

    [null, null, null, WP, null, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  white\_in\_check: false

  black\_in\_check: true

test7:

  pieces\_matrix: [

    [null, null, null, null, BK, null, null, null],

    [null, null, null, WP, null, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  white\_in\_check: false

  black\_in\_check: true

test8:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, null, WP, BK, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  white\_in\_check: true

  black\_in\_check: true

test9:

  pieces\_matrix: [

    [null, null, null, null, null, null, null, null],

    [null, null, BK, WP, null, null, null, null],

    [null, null, null, WK, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

    [null, null, null, null, null, null, null, null],

  ]

  white\_in\_check: true

  black\_in\_check: true

All these tests passed which really confused me. The check function correctly determined that the child game states resulting from each of these moves were check for black.

To further analyse this I created a jupyter notebook. I tried to replicate the exact function call being made by the test with a version of the legal moves generator that included a lot of print statements to reveal the inner workings.

I was able to recreate the contradictory outcomes

Cell 1:

Text

Description automatically generated

Cell 2 (copied as too big to screenshot)

STARTING\_POSITIONS: tuple[tuple[pieces\_mod.Piece]] = (

    (

        pieces\_mod.Rook(color="B"),

        pieces\_mod.Knight(color="B"),

        pieces\_mod.Bishop(color="B"),

        pieces\_mod.Queen(color="B"),

        pieces\_mod.King(color="B"),

        pieces\_mod.Bishop(color="B"),

        pieces\_mod.Knight(color="B"),

        pieces\_mod.Rook(color="B")

    ),

    (pieces\_mod.Pawn(color="B"),)\*8,

    (None,)\*8,

    (None,)\*8,

    (None,)\*8,

    (None,)\*8,

    (pieces\_mod.Pawn(color="W"),)\*8,

    (

        pieces\_mod.Rook(color="W"),

        pieces\_mod.Knight(color="W"),

        pieces\_mod.Bishop(color="W"),

        pieces\_mod.Queen(color="W"),

        pieces\_mod.King(color="W"),

        pieces\_mod.Bishop(color="W"),

        pieces\_mod.Knight(color="W"),

        pieces\_mod.Rook(color="W")

    )

)

@dataclass(frozen=True)

class Board\_State():

    pieces\_matrix: tuple[tuple[pieces\_mod.Piece]]

    next\_to\_go: int = "W"

    # pieces\_matrix: tuple[tuple[pieces\_mod.Piece]] = STARTING\_POSITIONS

    def print\_board(self, print\_function=pprint):

        print\_function(

            list(map(

                lambda row: list(map(

                    lambda piece: None if piece is None else piece.symbol(),

                    row

                )),

                self.pieces\_matrix

            ))

        )

    def get\_piece\_at\_vector(self, vector: Vector):

        column, row = vector.i, 7-vector.j

        # column, row = vector.i, vector.j

        return self.pieces\_matrix[row][column]

    def generate\_all\_pieces(self):

        # for i, j in zip(range(8), range(8)):

        # should be product

        for i, j in iter\_product(range(8), range(8)):

            position\_vector = Vector(i,j)

            piece = self.get\_piece\_at\_vector(position\_vector)

            # skip if none

            if piece:

                yield piece, position\_vector

            # if piece:

            #     dev\_print(f"not skipping {position\_vector}")

            #     yield piece, position\_vector

            # else:

            #     dev\_print(f"skipping {position\_vector}")

    def generate\_pieces\_of\_color(self, color=None):

        if color is None:

            color = self.next\_to\_go

        def same\_color(data\_item):

            piece, \_ = data\_item

            return piece.color == color

        yield from filter(

            same\_color,

            self.generate\_all\_pieces()

        )

    def color\_in\_check(self, color=None):

        if color is None:

            color = self.next\_to\_go

        # it is now A's turn

        color\_a = color

        color\_b = "W" if color == "B" else "B"

        # we will examine all the movement vectors of B's pieces

        # if any of them could take the A's King then currently A is in check as their king is threatened by 1 or more pieces (which could take it next turn)

        for piece, position\_v in self.generate\_pieces\_of\_color(color=color\_b):

            movement\_vs = piece.generate\_movement\_vectors(

                pieces\_matrix=self.pieces\_matrix,

                position\_vector=position\_v

            )

            for movement\_v in movement\_vs:

                resultant = position\_v + movement\_v

                to\_square = self.get\_piece\_at\_vector(resultant)

                # print(f"to\_square   -->   {to\_square!r}")

                # dev\_print(f"piece at square {resultant.to\_square()} is {to\_square.symbol() if to\_square else '<empty>'}")

                # dev\_print(f"piece at square {resultant.to\_square()} is {repr(to\_square) if to\_square else '<empty>'}")

                # As\_move\_threatens\_king\_A = isinstance(to\_square, pieces\_mod.King) and to\_square.color == color\_a

                As\_move\_threatens\_king\_A = (to\_square == pieces\_mod.King(color=color\_a))

                # print(f"to\_square == pieces\_mod.King(color='B')   -->   {to\_square!r} == {pieces\_mod.King(color='B')!r}   -->    {to\_square == pieces\_mod.King(color='B')}")

                # dev\_print(f"therefore king {'IS' if As\_move\_threatens\_king\_A else 'NOT'} threatened as square {'DOES' if As\_move\_threatens\_king\_A else 'DOES NOT'} contain {pieces\_mod.King(color=color\_a)!r} instead containing {repr(to\_square) if to\_square else '<empty>'}")

                # if As\_move\_threatens\_king\_A break out of all 3 loops

                if As\_move\_threatens\_king\_A:

                    # dev\_print(f"color\_in\_check(color='{color}') returning True")

                    # dev\_print(f"piece {piece.symbol()} at {position\_v.to\_square()} moving to {resultant.to\_square()} IS threatening king")

                    return True

                # else:

                    # dev\_print(f"piece {piece.symbol()} at {position\_v.to\_square()} moving to {resultant.to\_square()} NOT threatening king")

        # dev\_print(f"color\_in\_check(color='{color}') returning False")

        return False

    def generate\_legal\_moves(self):

        # dev\_print(f"analysing legal moves for next to go {self.next\_to\_go}")

        for piece, piece\_position\_vector in self.generate\_pieces\_of\_color(color=self.next\_to\_go):

            # dev\_print(f"analysing moves for piece: {piece.symbol()}")

            movement\_vectors = piece.generate\_movement\_vectors(

                pieces\_matrix=self.pieces\_matrix,

                position\_vector=piece\_position\_vector

            )

            for movement\_vector in movement\_vectors:

                # dev\_print(f"\tpiece {piece.symbol()}:\_analysing movement vector {repr(movement\_vector)}")

                child\_game\_state: Board\_State = self.make\_move(from\_position\_vector=piece\_position\_vector, movement\_vector=movement\_vector)

                # dev\_print(f"\t\tmovement vector {repr(movement\_vector)}: results in child game state")

                # child\_game\_state.print\_board(

                #     print\_function=lambda rows: list(map(

                #         lambda item: print(f"\t\t{item}"),

                #         rows

                #     ))

                # )

                # dev\_print(repr(child\_game\_state))

                is\_check\_after\_move = child\_game\_state.color\_in\_check(color=self.next\_to\_go)

                dev\_print(f"\t\tThis game state in check? for {self.next\_to\_go}:  {is\_check\_after\_move}")

                if not is\_check\_after\_move:

                    yield piece\_position\_vector, movement\_vector

    # def generate\_legal\_moves(self):

    #     for piece, piece\_position\_vector in self.generate\_pieces\_of\_color(color=self.next\_to\_go):

    #         movement\_vectors = piece.generate\_movement\_vectors(

    #             pieces\_matrix=self.pieces\_matrix,

    #             position\_vector=piece\_position\_vector

    #         )

    #         for movement\_vector in movement\_vectors:

    #             child\_game\_state = self.make\_move(from\_position\_vector=piece\_position\_vector, movement\_vector=movement\_vector)

    #             is\_check\_after\_move = child\_game\_state.color\_in\_check(color=self.next\_to\_go)

    #             if not is\_check\_after\_move:

    #                 yield piece\_position\_vector, movement\_vector

    def is\_game\_over\_for\_next\_to\_go(self):

        # check if in checkmate

        # for player a

        # if b has no moves

        if not list(self.generate\_legal\_moves()):

            # if b in check

            if self.color\_in\_check():

                # checkmate for b, a wins

                return True, self.next\_to\_go

            else:

                # stalemate

                return True, None

        return False, None

    def static\_evaluation(self):

        """Give positive static evaluation if white is winning"""

        def generate\_all\_pieces():

            for i, j in iter\_product(range(8), range(8)):

                piece\_position\_vector = Vector(i, j)

                piece: pieces\_mod.Piece = self.get\_piece\_at\_vector(piece\_position\_vector)

                if not piece:

                    continue

                yield piece, piece\_position\_vector

        over, winner = self.is\_game\_over\_for\_next\_to\_go()

        if over:

            match winner:

                case None: multiplier = 0

                case "W": multiplier = 1

                case "B": multiplier = -1

            return winner \* ARBITRARILY\_LARGE\_VALUE

        else:

            return sum(piece.get\_value(position\_vector) \* multiplier for piece, position\_vector in generate\_all\_pieces())

    def make\_move(self, from\_position\_vector: Vector, movement\_vector: Vector):

        to\_position\_vector = from\_position\_vector + movement\_vector

        # poor code, this below line can cause infinite recursion when legal moves generator called post check changes

        # assert (from\_position\_vector, movement\_vector) in self.generate\_legal\_moves()

        new\_pieces\_matrix = deepcopy(self.pieces\_matrix)

        # convert to list

        new\_pieces\_matrix = list(map(list, new\_pieces\_matrix))

        # set from to blank

        row, col = 7-from\_position\_vector.j, from\_position\_vector.i

        piece: pieces\_mod.Piece = new\_pieces\_matrix[row][col]

        piece.last\_move = movement\_vector

        new\_pieces\_matrix[row][col] = None

        # set to square to this piece

        row, col = 7-to\_position\_vector.j, to\_position\_vector.i

        new\_pieces\_matrix[row][col] = piece

        # convert back to tuple

        new\_pieces\_matrix = tuple(map(tuple, new\_pieces\_matrix))

        new\_next\_to\_go = "W" if self.next\_to\_go == "B" else "B"

        return Board\_State(

            next\_to\_go=new\_next\_to\_go,

            pieces\_matrix=new\_pieces\_matrix

        )

Cell 3:

# tested so assumed correct

import pieces

from vector import Vector

def test\_dir(file\_name): return f"test\_data/board\_state/{file\_name}.yaml"

# code repeated from test pieces, opportunity to reduce redundancy

def list\_map(function, iterable): return list(map(function, iterable))

def tuple\_map(function, iterable): return tuple(map(function, iterable))

def descriptor\_to\_piece(descriptor) -> pieces.Piece:

    # converts WN to knight object with a color attribute of white

    if descriptor is None:

        return None

    color, symbol = descriptor

    piece\_type: pieces.Piece = pieces.PIECE\_TYPES[symbol]

    return piece\_type(color=color)

def deserialize\_pieces\_matrix(pieces\_matrix, next\_to\_go="W") -> Board\_State:

    def row\_of\_symbols\_to\_pieces(row):

        return list\_map(descriptor\_to\_piece, row)

    # update pieces\_matrix replacing piece descriptors to piece objects

    pieces\_matrix = list\_map(row\_of\_symbols\_to\_pieces, pieces\_matrix)

    board\_state: Board\_State = Board\_State(pieces\_matrix=pieces\_matrix, next\_to\_go=next\_to\_go)

    return board\_state

cell 4 though 6

A screenshot of a computer

Description automatically generated

cell 7-8

Graphical user interface, text

Description automatically generated

Cell 9-11

Graphical user interface, text

Description automatically generated

This highlights the issue and apparent contradiction in logic. My chess program is checking each child game state and reporting that black isn’t in check, yielding all 5 moves. Then for one of the child game states the color in check functions seen to directly contradict this by correctly identifying that black is in check.

I later realised the issue and why all my unit tests for the check function had worked and yet a practical test vie the legal moves generator function failed.

The line that was the issue what part of the Piece.\_\_eq\_\_ method

A screenshot of a computer

Description automatically generated with medium confidence

Line 113 was the issue. Shown commented out.

This line meant that a king that had been moved was not equal to a king that had just been initialised for the purpose of a test. This is because they would have different last moves attributes. The comparison in the check function is to a king of the same color but with last\_moves set to None. This worked for the unit tests as the kings in the tests also had not moved. Once this line was removed, all the tests passed.

This was an issue with the pieces testing ad my assumption that the module was robust, but it was fixed.

Next I moved onto a module that would contain a game class that was able to keep track of the whole game.

It should enable user moves to be implemented and computer moves to be generated and implemented (I will get to minimax).

Here was my code:

**Game.py**

# import other modules

from board\_state import Board\_State

from minimax import minimax

from vector import Vector

from assorted import ARBITRARILY\_LARGE\_VALUE, dev\_print, NotUserTurn, InvalidMove

# the game class is used to keep track of a chess game between a user and the computer

class Game(object):

    # it keeps track of:

    # the player's color's

    player\_color\_key: dict

    # the difficulty or depth of the game

    depth: int

    # the current board state

    board\_state: Board\_State

    # the number of moves so far

    move\_counter: int

    # a table of game state hashes and there frequency

    piece\_matrix\_occurrence\_hash\_table: list

    # this adds a new game state to the frequency table

    def add\_new\_piece\_matrix\_to\_hash\_table(self, piece\_matrix):

        # the pieces matrix is hashed

        matrix\_hash = hash(piece\_matrix)

        # if this pieces matrix has been encountered before, add 1 to the frequency

        if matrix\_hash in self.piece\_matrix\_occurrence\_hash\_table.keys():

            self.piece\_matrix\_occurrence\_hash\_table[matrix\_hash] += 1

        # else set frequency to 1

        else:

            self.piece\_matrix\_occurrence\_hash\_table[matrix\_hash] = 1

    # determines if there is a 3 repeat stalemate

    def is\_3\_board\_repeats\_in\_game\_history(self):

        # if any of the board states have been repeated 3 or more times: stalemate

        return any(value >= 3 for value in self.piece\_matrix\_occurrence\_hash\_table.values())

    # constructor for game object

    def \_\_init\_\_(self, depth=2, user\_color="W") -> None:

        # based on user's color, determine color key

        self.player\_color\_key = {

            "W": 1 if user\_color=="W" else -1,

            "B": -1 if user\_color=="W" else 1

        }

        # set depth property from parameters

        self.depth = depth

        # set attributes for game at start

        self.board\_state = Board\_State()

        self.move\_counter = 0

        self.piece\_matrix\_occurrence\_hash\_table = {}

    # this function validates if the user's move is allowed and if so, makes it

    def implement\_user\_move(self, from\_square, to\_square) -> None:

        # check that the user is allowed to move

        which\_player\_next\_to\_go = self.player\_color\_key.get(

            self.board\_state.next\_to\_go

        )

        if which\_player\_next\_to\_go != 1:

            raise NotUserTurn(game=self)

        # unpack move into vector form

        # invalid square syntaxes will cause a value error here

        try:

            position\_vector = Vector.construct\_from\_square(from\_square)

            movement\_vector = Vector.construct\_from\_square(to\_square) - position\_vector

        except Exception:

            raise ValueError("Square's not in valid format")

        # if the move is not in the set of legal moves, raise and appropriate exception

        if (position\_vector, movement\_vector) not in self.board\_state.generate\_legal\_moves():

            raise InvalidMove(game=self)

        # implement move

        self.board\_state = self.board\_state.make\_move(from\_position\_vector=position\_vector, movement\_vector=movement\_vector)

        # adjust other properties that keep track of the game

        self.move\_counter += 1

        self.add\_new\_piece\_matrix\_to\_hash\_table(self.board\_state.pieces\_matrix)

        return (position\_vector, movement\_vector), self.board\_state.static\_evaluation()

    # this function determines if the game is over and if so, what is the nature of the outcome

    def check\_game\_over(self):

        # returns: over: bool, winning\_player: (1/-1), classification: str

        # check for 3 repeat stalemate

        if self.is\_3\_board\_repeats\_in\_game\_history():

            return True, None, "Stalemate"

        # determine if board state is over for next player

        over, winner = self.board\_state.is\_game\_over\_for\_next\_to\_go()

        # switch case statement to determine the appropriate values to be returned in each case

        match (over, winner):

            case False, \_:

                victory\_classification = None

                winning\_player = None

            case True, None:

                victory\_classification = "Stalemate"

                winning\_player = None

            case True, winner:

                victory\_classification = "Checkmate"

                winning\_player = self.player\_color\_key[winner]

        # return appropriate values

        return over, winning\_player, victory\_classification

    # this function determines and implements the computer move

    def implement\_computer\_move(self, best\_move\_function=None):

        # for use with testing bots, a best move function can be provided for use, but minimax if the default

        # get next to go player (1/-1)

        which\_player\_next\_to\_go = self.player\_color\_key.get(

            self.board\_state.next\_to\_go

        )

        # check that is it the computer's turn

        if which\_player\_next\_to\_go != -1:

            raise ValueError(f"Next to go is user: {self.board\_state.next\_to\_go} not computer")

        # if no function provided, default to minimax

        if best\_move\_function is None:

            score, best\_child, best\_move = minimax(

                board\_state = self.board\_state,

                # assume white / user always maximizer

                # is\_maximizer = (self.board\_state.next\_to\_go == "W"),

                is\_maximizer = False,

                # depth is based of difficulty of game based on depth parameter

                depth = self.depth,

                # default values for alpha and beta

                alpha = (-1)\*ARBITRARILY\_LARGE\_VALUE,

                beta = ARBITRARILY\_LARGE\_VALUE

            )

        else:

            # otherwise use provided function,

            # the provided function should take game as an argument and then return data in the same format as the minimax function

            score, best\_child, best\_move = best\_move\_function(self)

        # adjust properties that keep track of the game state

        self.board\_state = best\_child

        self.move\_counter += 1

        self.add\_new\_piece\_matrix\_to\_hash\_table(self.board\_state.pieces\_matrix)

        # incase is it wanted for a print out ect, return move and score

        return best\_move, score

in tandem with this I created a minimax function which I will go into more detail about shortly.

Rather than make a test for this I created a rudementry console chess game to play chess using this library.

from game import Game

from assorted import InvalidMove

# create new chess game

# difficulty set to depth 2

game = Game(depth=2)

# this function informs the user of the details when the game is over

def handle\_game\_over(winner, classification):

    print(f"The game is over, the {'user' if winner==1 else 'computer'} has won in a {classification}")

# print out the starting board

game.board\_state.print\_board()

print()

# keep game going until loop broken

while True:

    # user goes first

    print("Your go USER:")

    # while loop and error checking used to ensure move input

    while True:

        try:

            print("Please enter move in 2 parts")

            from\_square = input("From square:  ")

            to\_square = input("To square:   ")

            # check

            game.implement\_user\_move(from\_square, to\_square)

        except InvalidMove:

            print("This isn't a legal move, try again")

        except ValueError:

            print("This isn't valid input, try again")

        else:

            # if it worked break out of the loop

            break

    # if move results in check then output this

    if game.board\_state.color\_in\_check():

            print("CHECK!")

    # print out the current board\_state

    game.board\_state.print\_board()

    print()

    # check if game over after user's move

    over, winner, classification = game.check\_game\_over()

    # if over, handle it.

    if over:

        handle\_game\_over(winner=winner, classification=classification)

        break

    # alternate, it is now the computers go

    print("Computer's go: ")

    # get the computers move

    move, \_ = game.implement\_computer\_move()

    # print out the board again

    game.board\_state.print\_board()

    # print out the computer's move in terms of squares

    position\_vector, movement\_vector = move

    resultant\_vector = position\_vector + movement\_vector

    piece\_symbol = game.board\_state.get\_piece\_at\_vector(resultant\_vector).symbol()

    print(f"Computer Moved  {piece\_symbol}:  {position\_vector.to\_square()} to {resultant\_vector.to\_square()}")

    # print check if applicable

    if game.board\_state.color\_in\_check():

            print("CHECK!")

    # check if the game is over, if so handle it

    over, winner, classification = game.check\_game\_over()

    if over:

        handle\_game\_over(winner=winner, classification=classification)

        break

    # create a new line to separate for the user's next move

    print()

This allowed the user to play check against the computer. The game wasn’t configurable as the user was white against depth 2 minimax but it was playable.

Here is an example of it running:

PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python console\_chess.py

8| BR  BN  BB  BQ  BK  BB  BN  BR

7| BP  BP  BP  BP  BP  BP  BP  BP

6| ·   ·   ·   ·   ·   ·   ·   ·

5| ·   ·   ·   ·   ·   ·   ·   ·

4| ·   ·   ·   ·   ·   ·   ·   ·

3| ·   ·   ·   ·   ·   ·   ·   ·

2| WP  WP  WP  WP  WP  WP  WP  WP

1| WR  WN  WB  WQ  WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Your go USER:

Please enter move in 2 parts

From square:  D2

To square:   D4

8| BR  BN  BB  BQ  BK  BB  BN  BR

7| BP  BP  BP  BP  BP  BP  BP  BP

6| ·   ·   ·   ·   ·   ·   ·   ·

5| ·   ·   ·   ·   ·   ·   ·   ·

4| ·   ·   ·   WP  ·   ·   ·   ·

3| ·   ·   ·   ·   ·   ·   ·   ·

2| WP  WP  WP  ·   WP  WP  WP  WP

1| WR  WN  WB  WQ  WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer's go:

8| BR  ·   BB  BQ  BK  BB  BN  BR

7| BP  BP  BP  BP  BP  BP  BP  BP

6| ·   ·   BN  ·   ·   ·   ·   ·

5| ·   ·   ·   ·   ·   ·   ·   ·

4| ·   ·   ·   WP  ·   ·   ·   ·

3| ·   ·   ·   ·   ·   ·   ·   ·

2| WP  WP  WP  ·   WP  WP  WP  WP

1| WR  WN  WB  WQ  WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer Moved  BN:  B8 to C6

Your go USER:

Please enter move in 2 parts

From square:  rubbish that is invalid input

To square:   more rubbish

This isn't valid input, try again

Please enter move in 2 parts

From square:  a0

To square:   i9

This isn't a legal move, try again

Please enter move in 2 parts

From square:  e1

To square:   e2

This isn't a legal move, try again

Please enter move in 2 parts

From square:  E2

To square:   E3

8| BR  ·   BB  BQ  BK  BB  BN  BR

7| BP  BP  BP  BP  BP  BP  BP  BP

6| ·   ·   BN  ·   ·   ·   ·   ·

5| ·   ·   ·   ·   ·   ·   ·   ·

4| ·   ·   ·   WP  ·   ·   ·   ·

3| ·   ·   ·   ·   WP  ·   ·   ·

2| WP  WP  WP  ·   ·   WP  WP  WP

1| WR  WN  WB  WQ  WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer's go:

8| BR  ·   BB  BQ  BK  BB  ·   BR

7| BP  BP  BP  BP  BP  BP  BP  BP

6| ·   ·   BN  ·   ·   BN  ·   ·

5| ·   ·   ·   ·   ·   ·   ·   ·

4| ·   ·   ·   WP  ·   ·   ·   ·

3| ·   ·   ·   ·   WP  ·   ·   ·

2| WP  WP  WP  ·   ·   WP  WP  WP

1| WR  WN  WB  WQ  WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer Moved  BN:  G8 to F6

Your go USER:

Please enter move in 2 parts

From square:  F2

To square:   F3

8| BR  ·   BB  BQ  BK  BB  ·   BR

7| BP  BP  BP  BP  BP  BP  BP  BP

6| ·   ·   BN  ·   ·   BN  ·   ·

5| ·   ·   ·   ·   ·   ·   ·   ·

4| ·   ·   ·   WP  ·   ·   ·   ·

3| ·   ·   ·   ·   WP  WP  ·   ·

2| WP  WP  WP  ·   ·   ·   WP  WP

1| WR  WN  WB  WQ  WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer's go:

8| BR  ·   BB  BQ  BK  BB  ·   BR

7| BP  BP  BP  ·   BP  BP  BP  BP

6| ·   ·   BN  ·   ·   BN  ·   ·

5| ·   ·   ·   BP  ·   ·   ·   ·

4| ·   ·   ·   WP  ·   ·   ·   ·

3| ·   ·   ·   ·   WP  WP  ·   ·

2| WP  WP  WP  ·   ·   ·   WP  WP

1| WR  WN  WB  WQ  WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer Moved  BP:  D7 to D5

Your go USER:

Please enter move in 2 parts

From square:  C2

To square:   C3

8| BR  ·   BB  BQ  BK  BB  ·   BR

7| BP  BP  BP  ·   BP  BP  BP  BP

6| ·   ·   BN  ·   ·   BN  ·   ·

5| ·   ·   ·   BP  ·   ·   ·   ·

4| ·   ·   ·   WP  ·   ·   ·   ·

3| ·   ·   WP  ·   WP  WP  ·   ·

2| WP  WP  ·   ·   ·   ·   WP  WP

1| WR  WN  WB  WQ  WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer's go:

8| BR  ·   ·   BQ  BK  BB  ·   BR

7| BP  BP  BP  ·   BP  BP  BP  BP

6| ·   ·   BN  ·   BB  BN  ·   ·

5| ·   ·   ·   BP  ·   ·   ·   ·

4| ·   ·   ·   WP  ·   ·   ·   ·

3| ·   ·   WP  ·   WP  WP  ·   ·

2| WP  WP  ·   ·   ·   ·   WP  WP

1| WR  WN  WB  WQ  WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer Moved  BB:  C8 to E6

Your go USER:

Please enter move in 2 parts

From square:  D1

To square:   D4

This isn't a legal move, try again

Please enter move in 2 parts

From square:  D1

To square:   A4

8| BR  ·   ·   BQ  BK  BB  ·   BR

7| BP  BP  BP  ·   BP  BP  BP  BP

6| ·   ·   BN  ·   BB  BN  ·   ·

5| ·   ·   ·   BP  ·   ·   ·   ·

4| WQ  ·   ·   WP  ·   ·   ·   ·

3| ·   ·   WP  ·   WP  WP  ·   ·

2| WP  WP  ·   ·   ·   ·   WP  WP

1| WR  WN  WB  ·   WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer's go:

8| BR  ·   ·   BQ  BK  BB  ·   BR

7| BP  BP  BP  ·   ·   BP  BP  BP

6| ·   ·   BN  ·   BB  BN  ·   ·

5| ·   ·   ·   BP  BP  ·   ·   ·

4| WQ  ·   ·   WP  ·   ·   ·   ·

3| ·   ·   WP  ·   WP  WP  ·   ·

2| WP  WP  ·   ·   ·   ·   WP  WP

1| WR  WN  WB  ·   WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer Moved  BP:  E7 to E5

Your go USER:

Please enter move in 2 parts

From square:  A4

To square:   B4

8| BR  ·   ·   BQ  BK  BB  ·   BR

7| BP  BP  BP  ·   ·   BP  BP  BP

6| ·   ·   BN  ·   BB  BN  ·   ·

5| ·   ·   ·   BP  BP  ·   ·   ·

4| ·   WQ  ·   WP  ·   ·   ·   ·

3| ·   ·   WP  ·   WP  WP  ·   ·

2| WP  WP  ·   ·   ·   ·   WP  WP

1| WR  WN  WB  ·   WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer's go:

8| BR  ·   ·   BQ  BK  ·   ·   BR

7| BP  BP  BP  ·   ·   BP  BP  BP

6| ·   ·   BN  ·   BB  BN  ·   ·

5| ·   ·   ·   BP  BP  ·   ·   ·

4| ·   BB  ·   WP  ·   ·   ·   ·

3| ·   ·   WP  ·   WP  WP  ·   ·

2| WP  WP  ·   ·   ·   ·   WP  WP

1| WR  WN  WB  ·   WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer Moved  BB:  F8 to B4

Your go USER:

Please enter move in 2 parts

From square:  C3

To square:   B4

8| BR  ·   ·   BQ  BK  ·   ·   BR

7| BP  BP  BP  ·   ·   BP  BP  BP

6| ·   ·   BN  ·   BB  BN  ·   ·

5| ·   ·   ·   BP  BP  ·   ·   ·

4| ·   WP  ·   WP  ·   ·   ·   ·

3| ·   ·   ·   ·   WP  WP  ·   ·

2| WP  WP  ·   ·   ·   ·   WP  WP

1| WR  WN  WB  ·   WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer's go:

8| BR  ·   ·   BQ  BK  ·   ·   BR

7| BP  BP  BP  ·   ·   BP  BP  BP

6| ·   ·   BN  ·   BB  BN  ·   ·

5| ·   ·   ·   BP  ·   ·   ·   ·

4| ·   WP  ·   BP  ·   ·   ·   ·

3| ·   ·   ·   ·   WP  WP  ·   ·

2| WP  WP  ·   ·   ·   ·   WP  WP

1| WR  WN  WB  ·   WK  WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer Moved  BP:  E5 to D4

Your go USER:

Please enter move in 2 parts

From square:  E1

To square:   D2

8| BR  ·   ·   BQ  BK  ·   ·   BR

7| BP  BP  BP  ·   ·   BP  BP  BP

6| ·   ·   BN  ·   BB  BN  ·   ·

5| ·   ·   ·   BP  ·   ·   ·   ·

4| ·   WP  ·   BP  ·   ·   ·   ·

3| ·   ·   ·   ·   WP  WP  ·   ·

2| WP  WP  ·   WK  ·   ·   WP  WP

1| WR  WN  WB  ·   ·   WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer's go:

8| BR  ·   ·   BQ  BK  ·   ·   BR

7| BP  BP  BP  ·   ·   BP  BP  BP

6| ·   ·   BN  ·   BB  BN  ·   ·

5| ·   ·   ·   BP  ·   ·   ·   ·

4| ·   WP  ·   ·   ·   ·   ·   ·

3| ·   ·   ·   ·   BP  WP  ·   ·

2| WP  WP  ·   WK  ·   ·   WP  WP

1| WR  WN  WB  ·   ·   WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer Moved  BP:  D4 to E3

CHECK!

Your go USER:

Please enter move in 2 parts

From square:  D2

To square:   E3

8| BR  ·   ·   BQ  BK  ·   ·   BR

7| BP  BP  BP  ·   ·   BP  BP  BP

6| ·   ·   BN  ·   BB  BN  ·   ·

5| ·   ·   ·   BP  ·   ·   ·   ·

4| ·   WP  ·   ·   ·   ·   ·   ·

3| ·   ·   ·   ·   WK  WP  ·   ·

2| WP  WP  ·   ·   ·   ·   WP  WP

1| WR  WN  WB  ·   ·   WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer's go:

8| BR  ·   ·   BQ  BK  ·   ·   BR

7| BP  BP  BP  ·   ·   BP  BP  BP

6| ·   ·   BN  ·   BB  BN  ·   ·

5| ·   ·   ·   ·   ·   ·   ·   ·

4| ·   WP  ·   BP  ·   ·   ·   ·

3| ·   ·   ·   ·   WK  WP  ·   ·

2| WP  WP  ·   ·   ·   ·   WP  WP

1| WR  WN  WB  ·   ·   WB  WN  WR

  (A   B   C   D   E   F   G   H )

Computer Moved  BP:  D5 to D4

CHECK!

Your go USER:

Please enter move in 2 parts

From square:  E3

To square:   D4

This isn't a legal move, try again

You can see 1 problem and some sophisticated behaviours.

* It prints out the board after each move, makes it clear who’s turn it is and what move the computer has made.
* We can see that the validation works as I try in my second move to move in invalid ways or to input jargon and the program asks me again.
* We can also see that, when given the opportunity the computer sacrificed a rook to take a queen.
* We see that I cannot move in a way that causes check. This is after I moved out my king to present an easy check. We also see that it printed check which is useful

The issue we see is when the computer moves a pawn from E7 to E5 over the top of a bishop. This isn’t a legal move but I forgot to check that the middle square and more to squares were empty when coding in the pawns movement. **This will be corrected for future.**

I created the following minimax function

# import local modules

# cannot import game as causes circular import, if necessary put in same file

from board\_state import Board\_State

from assorted import ARBITRARILY\_LARGE\_VALUE

from vector import Vector

# my minimax function takes as arguments:

# Board\_State, is\_maximiser, alpha and beta (used for pruning) and check\_extra\_depth (produces better outcome but slower)

# it returns

# score, child, move

def minimax(board\_state: Board\_State, is\_maximizer: bool, depth, alpha, beta, check\_extra\_depth=True):

    # sourcery skip: low-code-quality, remove-unnecessary-else, swap-if-else-branches

    # assume white is maximizer

    # when calling, if give appropriate max min arg

    # base case

    # if over or depth==0 return static evaluation

    over, \_ = board\_state.is\_game\_over\_for\_next\_to\_go()

    if depth == 0 or over:

        # special recursive case 1

        # examine terminal nodes that are check to depth 2 (variable depth)

        # to avoid goose chaises, extra resources are allowed if check not already explored

        if board\_state.color\_in\_check() and check\_extra\_depth and not over:

            # print(f"checking board state {hash(board\_state)} at additional depth due to check")

            return minimax(

                board\_state=board\_state,

                is\_maximizer=is\_maximizer,

                depth=2,

                alpha=alpha,

                beta=beta,

                check\_extra\_depth=False

            )

        else:

            # static eval works for game over to

            return board\_state.static\_evaluation(), None, None

    # define variables used to return more that just score (move and child)

    best\_child\_game\_state: Board\_State | None = None

    best\_move\_vector: Vector | None = None

    # function yields move ordered by how favorable they are (low depth minimax approximation)

    def gen\_ordered\_child\_game\_states():

        # this function does a low depth minimax recursive call (special recursive case 2) to give a move a score

        def approx\_score\_move(move):

            child\_game\_state = board\_state.make\_move(\*move)

            return minimax(

                board\_state=child\_game\_state,

                depth=depth-2,

                is\_maximizer=not is\_maximizer,

                alpha=alpha,

                beta=beta,

                check\_extra\_depth=False

            )[0]

            # print(f"approx\_score\_move(move={move!r})  ->  {result!r}")

        # if depth is 1 or less just yield moves form legal moves

        if depth <= 1:

            yield from board\_state.generate\_legal\_moves()

        # else sort them

        else:

            # sort best to worse

            # sort ascending order if minimizer, descending if maximizer

            yield from sorted(

                board\_state.generate\_legal\_moves(),

                key=approx\_score\_move,

                reverse=is\_maximizer

            )

    if is\_maximizer:

        # set max to -infinity

        maximum\_evaluation = (-1)\*ARBITRARILY\_LARGE\_VALUE

        # iterate through moves and resulting game states

        for position\_vector, movement\_vector in gen\_ordered\_child\_game\_states():

            child\_game\_state = board\_state.make\_move(from\_position\_vector=position\_vector, movement\_vector=movement\_vector)

            # evaluate each one

            # general recursive case 1

            evaluation, \_, \_ = minimax(

                board\_state=child\_game\_state,

                is\_maximizer=not is\_maximizer,

                depth=depth-1,

                alpha=alpha,

                beta=beta,

                check\_extra\_depth=check\_extra\_depth

            )

            # update alpha and max evaluation

            if evaluation > maximum\_evaluation:

                maximum\_evaluation = evaluation

                best\_child\_game\_state = child\_game\_state

                best\_move\_vector = (position\_vector, movement\_vector)

                alpha = max(alpha, evaluation)

            # where possible, prune

            if beta <= alpha:

                # print("Pruning!")

                break

        # once out of loop, return result

        return maximum\_evaluation, best\_child\_game\_state, best\_move\_vector

    else:

        minimum\_evaluation = ARBITRARILY\_LARGE\_VALUE

        for position\_vector, movement\_vector in gen\_ordered\_child\_game\_states():

            child\_game\_state = board\_state.make\_move(from\_position\_vector=position\_vector, movement\_vector=movement\_vector)

            evaluation, \_, \_ = minimax(

                board\_state=child\_game\_state,

                is\_maximizer=not is\_maximizer,

                depth=depth-1,

                alpha=alpha,

                beta=beta,

                check\_extra\_depth=check\_extra\_depth

            )

            if evaluation < minimum\_evaluation:

                minimum\_evaluation = evaluation

                best\_child\_game\_state = child\_game\_state

                best\_move\_vector = (position\_vector, movement\_vector)

                beta = min(beta, evaluation)

            if beta <= alpha:

                # print("Pruning!")

                break

        return minimum\_evaluation, best\_child\_game\_state, best\_move\_vector

it features some efficiency upgrades:

* Alpha beta pruning
* Child nodes examined best first to enhance pruning
* Variable depth in check to close out games

I then created a test for the minimax function

# this test is responsible for testing various mutations of the minimax function and how they play, it is not a data driven test

# imports

from random import choice as random\_choice

import unittest

# from functools import wraps

import multiprocessing

import os.path

import csv

from game import Game

from minimax import minimax

from assorted import ARBITRARILY\_LARGE\_VALUE

# from board\_state import Board\_State

# from vector import Vector

# # was not needed in the end, this decorator would have repeated a given function a given number of times

# def repeat\_decorator\_factory(times):

#     def decorator(func):

#         @wraps(func)

#         def wrapper(\*args, \*\*kwargs):

#             for \_ in range(times):

#                 func(\*args, \*\*kwargs)

#         return wrapper

#     return decorator

# this is a utility function that maps a function across an iterable but also converts the result to a list data structure

def list\_map(func, iter):

    return list(map(func, iter))

# this function is used to serialize a pieces matrix for output in a message

# it converts pieces to symbols

def map\_pieces\_matrix\_to\_symbols(pieces\_matrix):

    return list\_map(

        lambda row: list\_map(

            lambda square: square.symbol() if square else None,

            row

        ),

        pieces\_matrix

    )

# this functions updates a CSV file with the moves and scores of a chess game for graphical analysis in excel

def csv\_write\_move\_score(file\_path, move, score):

    # convert move to a pair of squares

    position\_vector, movement\_vector = move

    resultant\_vector = position\_vector + movement\_vector

    from\_square = position\_vector.to\_square()

    to\_square = resultant\_vector.to\_square()

    # if file doesn't exist, create is and add the headers

    if not os.path.exists(file\_path):

        with open(file\_path, "w", newline="") as file:

            writer = csv.writer(file, delimiter=",")

            writer.writerow(("from\_square", "to\_square", "score"))

    # add data as a new row

    with open(file\_path, "a", newline="") as file:

        writer = csv.writer(file, delimiter=",")

        writer.writerow((from\_square, to\_square, score))

# this contains the majority of the logic to do a bot vs bot test with the game class

# it is a component as it isn't the whole test

def minimax\_test\_component(description, good\_bot, bad\_bot, success\_criteria, write\_to\_csv=False):

    # sourcery skip: extract-duplicate-method

    # good bot and bad bot make decisions about moves,

    # the test is designed to assert that good bot wins (and or draws in some cases)

    # generate csv path

    if write\_to\_csv:

        # not sure why but the description sometimes contains an erroneous colon, this is caught and removed

        # was able to locate bug to here, as it is a test I added a quick fix

        # bug located, some description stings included them

        csv\_path = f"test\_reports/{description}.csv".replace(" ", "\_").replace(":", "")

    # start a new blank game

    # depth irrelevant as computer move function passed as parameter

    game: Game = Game()

    # keep them making moves until return statement breaks loop

    while True:

        # get move choice from bad bot

        \_, \_, move\_choice = bad\_bot(game)

        # serialised to is can be passed as a user move (reusing game class)

        position\_vector, movement\_vector = move\_choice

        resultant\_vector = position\_vector + movement\_vector

        from\_square, to\_square = position\_vector.to\_square(), resultant\_vector.to\_square()

        # implement bad bot move and update csv

        move, score = game.implement\_user\_move(from\_square=from\_square, to\_square=to\_square)

        if write\_to\_csv:

            csv\_write\_move\_score(

                file\_path=csv\_path,

                move=move,

                score=score

            )

        # see if this move causes the test to succeed or fail or keep going

        success, msg, board\_state = success\_criteria(game, description=description)

        if success is not None:

            return success, msg, board\_state

        # providing good bot function, implement good bot move and update csv

        move, score = game.implement\_computer\_move(best\_move\_function=good\_bot)

        if write\_to\_csv:

            csv\_write\_move\_score(

                file\_path=csv\_path,

                move=move,

                score=score

            )

        # again check if this affects the test

        success, msg, board\_state = success\_criteria(game, description=description)

        if success is not None:

            return success, msg, board\_state

        # # if needed provide console output to clarify that slow bot hasn't crashed

        # if game.move\_counter % 10 == 0 or depth >= 3:

        # print(f"Moves {game.move\_counter}:  static evaluation -> {game.board\_state.static\_evaluation()}, Minimax evaluation -> {score} by turn {description}")

# below are some function that have been programmed as classes with a \_\_call\_\_ method.

# these are basically fancy functions that CAN BE HASHED.

# I had to manually do this under the hood hashing as it is needed to allow communication between the threads

# a job must be hashable to be piped to a thread (separate python instance)

class Random\_Bot():

    # picks a random move

    def \_\_call\_\_(self, game):

        # determine move at random

        legal\_moves = list(game.board\_state.generate\_legal\_moves())

        assert len(legal\_moves) != 0

        # match minimax output structure

        # score, best\_child, best\_move

        return None, None, random\_choice(legal\_moves)

    def \_\_hash\_\_(self) -> int:

        return hash("I am random bot, I am a unique singleton so each instance can share a hash")

# picks a good move

# has constructor to allow for configuration

class Good\_Bot():

    # configure for depth and allow variable depth

    def \_\_init\_\_(self, depth, check\_extra\_depth):

        self.depth = depth

        self.check\_extra\_depth = check\_extra\_depth

    # make minimax function call given config

    def \_\_call\_\_(self, game):

        return minimax(

            board\_state=game.board\_state,

            is\_maximizer=(game.board\_state.next\_to\_go == "W"),

            depth=self.depth,

            alpha=(-1)\*ARBITRARILY\_LARGE\_VALUE,

            beta=ARBITRARILY\_LARGE\_VALUE,

            check\_extra\_depth=self.check\_extra\_depth

        )

    def \_\_hash\_\_(self) -> int:

        return hash(f"Good\_Bot(depth={self.depth}, check\_extra\_depth={self.check\_extra\_depth})")

# used to look at a game and decide if the test should finish

class Success\_Criteria():

    # constructor allow config for stalemates to sill allow test to pass

    def \_\_init\_\_(self, allow\_stalemate\_3\_states\_repeated: bool):

        self.allow\_stalemate\_3\_states\_repeated = allow\_stalemate\_3\_states\_repeated

    def \_\_call\_\_(self, game: Game, description):

        # returns: success, message, serialised pieces matrix

        # call game over and use a switch case to decide what to do

        match game.check\_game\_over():

            # if 3 repeat stalemate, check with config wether is is allows

            case True, None, "Stalemate":

                # game.board\_state.print\_board()

                if game.is\_3\_board\_repeats\_in\_game\_history and self.allow\_stalemate\_3\_states\_repeated:

                    # game.board\_state.print\_board()

                    # print(f"Success: Stalemate at {game.moves} moves in test {description}: 3 repeat board states, outcome specify included in allowed outcomes")

                    return True, f"Success: Stalemate at {game.move\_counter} moves in test {description}: 3 repeat board states, outcome specify included in allowed outcomes", map\_pieces\_matrix\_to\_symbols(game.board\_state.pieces\_matrix)

                else:

                    # game.board\_state.print\_board()

                    # print(f"FAILURE: ({description}) stalemate caused (3 repeats?  -> {game.is\_3\_board\_repeats\_in\_game\_history()})")

                    return False, f"FAILURE: ({description}) stalemate caused (3 repeats?  -> {game.is\_3\_board\_repeats\_in\_game\_history()})", map\_pieces\_matrix\_to\_symbols(game.board\_state.pieces\_matrix)

            # good bot loss causes test to fail

            case True, 1, "Checkmate":

                # game.board\_state.print\_board()

                # print(f"Failure: ({description}) computer lost")

                return False, f"Failure: ({description}) computer lost", map\_pieces\_matrix\_to\_symbols(game.board\_state.pieces\_matrix)

            # good bot win causes test to pass

            case True, -1, "Checkmate":

                # game.board\_state.print\_board()

                # print(f"SUCCESS: ({description}) Game has finished and been won in {game.move\_counter} moves")

                return True, f"SUCCESS: ({description}) Game has finished and been won in {game.move\_counter} moves", map\_pieces\_matrix\_to\_symbols(game.board\_state.pieces\_matrix)

            # if the game isn't over, return success as none and test will continue

            case False, \_, \_:

                return None, None, None

    def hash(self):

        return hash(f"Success\_Criteria(allow\_stalemate\_3\_states\_repeated={self.allow\_stalemate\_3\_states\_repeated})")

# given a test package (config for one test), carry it out

def execute\_test\_job(test\_data\_package):

    # deal with unexplained bug where argument is tuple / list length 1 containing relevant dict (quick fix as only a test)

    # I was able to identify that this is where it occurs and add a correction but I am not sure what the cause of the bug is

    if any(isinstance(test\_data\_package, some\_type) for some\_type in (tuple, list)):

        if len(test\_data\_package) == 1:

            test\_data\_package = test\_data\_package[0]

    # print(f"test\_data\_package   -->   {test\_data\_package}")

    # really simple, call minimax test component providing all keys in package as keyword arguments

    return minimax\_test\_component(\*\*test\_data\_package)

# this function takes an iterable of hashable test\_packages

# it all 8 logical cores on my computer to multitask to finish the test sooner

def pool\_jobs(test\_data):

    # counts logical cores

    # my CPU is a 10th gen i7

    # it has 4 cores and 8 logical cores due to hyper threading

    # with 4-8 workers I can use 100% of my CPU

    cores = multiprocessing.cpu\_count()

    # create a pool

    with multiprocessing.Pool(cores) as pool:

        # map the execute\_test\_job function across the set of test packages using multitasking

        # return the result

        return pool.map(

            func = execute\_test\_job,

            iterable = test\_data

        )

# test case contains unit tests

# multitasking only occurs within a test, tests are themselves executed sequentially

# I could pool all tests into one test function but this way multiple failures can occur in different tests

# (one single test would stop at first failure)

class Test\_Case(unittest.TestCase):

    # this function takes the results of the tests from the test pool and checks the results with a unittest

    # a failure is correctly identified to correspond to the function that called this function

    # reduces repeated logic

    def check\_test\_results(self, test\_results):

        for success, msg, final\_pieces\_matrix in test\_results:

            # print()

            # for row in final\_pieces\_matrix:

            #     row = "  ".join(map(

            #         lambda square: str(square).replace("None", ". "),

            #         row

            #     ))

            #     print(row)

            # print(msg)

            # i choose to iterate rather than assert all as this allows me to have the appropriate message on failure

            self.assertTrue(

                success,

                msg=msg

            )

    # tests basic minimax vs random moves

    def test\_vanilla\_depth\_1\_vs\_randotron(self):

        # 10 trials as outcome is linked to a random behaviour

        trials = 10

        # test package generated to include relevant data and logic (bots and success criteria)

        test\_data = {

            "description": "test: depth 1 vanilla vs randotron",

            "good\_bot": Good\_Bot(depth=1, check\_extra\_depth=False),

            "bad\_bot": Random\_Bot(),

            "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=True),

            "write\_to\_csv": False

        }

        # only trial will write to a csv

        test\_data\_but\_to\_csv = test\_data

        test\_data\_but\_to\_csv["write\_to\_csv"] = True

        self.check\_test\_results(

            pool\_jobs(

                (trials-1) \* [test\_data] +  [test\_data\_but\_to\_csv]

            )

        )

    def test\_advanced\_depth\_1\_vs\_randotron(self):

        trials = 10

        test\_data = {

            "description": "test: depth 1 advanced vs randotron",

            "good\_bot": Good\_Bot(depth=1, check\_extra\_depth=True),

            "bad\_bot": Random\_Bot(),

            "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=True),

            "write\_to\_csv": False

        }

        test\_data\_but\_to\_csv = test\_data

        test\_data\_but\_to\_csv["write\_to\_csv"] = True

        self.check\_test\_results(

            pool\_jobs(

                (trials-1) \* [test\_data] + [test\_data\_but\_to\_csv]

            )

        )

    def test\_depth\_2\_vs\_randotron(self):

        trials = 10

        test\_data = {

            "description": "test: depth 2 advanced vs randotron",

            "good\_bot": Good\_Bot(depth=2, check\_extra\_depth=True),

            "bad\_bot": Random\_Bot(),

            "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=False),

            "write\_to\_csv": False

        }

        test\_data\_but\_to\_csv = test\_data

        test\_data\_but\_to\_csv["write\_to\_csv"] = True

        self.check\_test\_results(

            pool\_jobs(

                (trials-1) \* [test\_data] + [test\_data\_but\_to\_csv]

            )

        )

    def test\_depth\_1\_advanced\_vs\_depth\_1\_vanilla(self):

        # only one trial needed as outcome is deterministic

        trials = 1

        test\_data = {

            "description": "test: depth 1 vanilla vs depth 1 variable check",

            "good\_bot": Good\_Bot(depth=1, check\_extra\_depth=True),

            "bad\_bot": Good\_Bot(depth=1, check\_extra\_depth=False),

            # they aren't different enough in efficacy to guarantee no draws

            "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=True),

            "write\_to\_csv": True

        },

        self.check\_test\_results(

            pool\_jobs(trials\*[test\_data])

        )

    def test\_depth\_2\_vs\_depth\_1(self):

        trials = 1

        test\_data = {

            "description": "test: depth 2 vs depth 1",

            "good\_bot": Good\_Bot(depth=2, check\_extra\_depth=True),

            "bad\_bot": Good\_Bot(depth=1, check\_extra\_depth=True),

            "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=False),

            "write\_to\_csv": True

        }

        self.check\_test\_results(

            pool\_jobs(trials\*[test\_data])

        )

    def test\_depth\_3\_vs\_depth\_2(self):

        trials = 1

        test\_data = {

            "description": "test: depth 3 vs depth 2",

            "good\_bot": Good\_Bot(depth=3, check\_extra\_depth=True),

            "bad\_bot": Good\_Bot(depth=2, check\_extra\_depth=True),

            "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=False),

            "write\_to\_csv": True

        }

        self.check\_test\_results(

            pool\_jobs(trials\*[test\_data])

        )

    def test\_depth\_3\_vs\_depth\_1(self):

        trials = 1

        test\_data = {

            "description": "test: depth 3 vs depth 1",

            "good\_bot": Good\_Bot(depth=3, check\_extra\_depth=True),

            "bad\_bot": Good\_Bot(depth=1, check\_extra\_depth=True),

            "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=False),

            "write\_to\_csv": True

        }

        self.check\_test\_results(

            pool\_jobs(trials\*[test\_data])

        )

    def test\_depth\_3\_vs\_randotron(self):

        trials = 4

        test\_data = {

            "description": "test: depth 3 vs randotron",

            "good\_bot": Good\_Bot(depth=3, check\_extra\_depth=True),

            "bad\_bot": Random\_Bot(),

            "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=False),

            "write\_to\_csv": False

        }

        test\_data\_but\_to\_csv = test\_data

        test\_data\_but\_to\_csv["write\_to\_csv"] = True

        self.check\_test\_results(

            pool\_jobs(

                (trials-1) \* [test\_data] + [test\_data\_but\_to\_csv]

            )

        )

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

    unittest.main()

I went to great effort to improve the tests efficiency and this wasn’t wasted. It allowed me to perform tests with many trials simultaneously in order to fully utilise my CPU.

Currently all the test I have run today were successful **but** even given a whole afternoon I haven’t been able to run all the tests. This is due to the depth 3 tests. Due to variable depth, the number of static evaluations can be (branching factor)^5 (really slow). This combinatorial explosion means that it takes more than 4 hours to run the tests.

The tests I have run successfully are:

* test\_depth\_1\_vanilla\_vs\_depth\_1\_variable\_check
* test\_depth\_2\_vs\_depth\_1
* test\_depth\_2\_advanced\_vs\_randotron
* test\_depth\_3\_vs\_depth\_1
* test\_depth\_1\_advanced\_vs\_randotron

Here is me showing how multitasking has allowed me to fully utilise my CPU

Let me run depth 2 vs randotron with 10 trials to demonstrate:

Text

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

Before doing this my CPU was only ever able to devote 25% power to my python program.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

As tasks finish the number of workers decreases. As my computer has 4 physical cores and 8 logical cores, near 100% cpu usage can continue until there are less than 4 threads. At this point each thread can only use 25% of the available CPU power. This means that the test doesn’t use the CPU fully at the end. This is only problematic is one of the games is particularly long and tends to stalemate as the next test cannot begin until this one has finished.

The first time I created the randobot and ran the test for depth 1 I found the issue where the game was resulting in a stalemate where it was endlessly repeating. I looked it up and in chess if a game state is repeated 3 times a stalemate occurs. I implemented this using the hash and frequency table in the Game class. This does end the game when this occurs. However it does mean that that the minimax function is unaware that a stalemate can occur this way as the logic isn’t happening in the board state class. This shouldn’t be an issue however as, when an infinite loop occurs the minimax will anticipate that its mean evaluation won’t change and so the stalemate wouldn’t affect its behaviour greatly anyway.

I have added this to my unit test as bots of similar skill level may sometimes draw. This can be specified in the test criteria.

I also recorded the utility values of the various tests in CSV files so that it could be graphed. I had a problem with this where the workers from each thread were all adding to the file, making it a useless mess. The fix was to add a .copy()

    # tests basic minimax vs random moves

    def test\_vanilla\_depth\_1\_vs\_randotron(self):

        # 10 trials as outcome is linked to a random behaviour

        # trials = 10

        trials = 5

        # test package generated to include relevant data and logic (bots and success criteria)

        test\_data = {

            "description": "test: depth 1 vanilla vs randotron",

            "good\_bot": Good\_Bot(depth=1, check\_extra\_depth=False),

            "bad\_bot": Random\_Bot(),

            "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=True),

            "write\_to\_csv": False

        }

        # only trial will write to a csv

        test\_data\_but\_to\_csv = test\_data.copy()

        test\_data\_but\_to\_csv["write\_to\_csv"] = True

        self.check\_test\_results(

            pool\_jobs(

                (trials-1) \* [test\_data] +  [test\_data\_but\_to\_csv]

            )

        )

The issue was that I was passing the dictionary by reference to a new variable and so I wasn’t mutating one test to use the CSV file but all of them.

This issue was particularly frustrating as I was wating hours for a set of corrupted data.

I have left my unit test running overnight and in 9 hours it still hasn’t finished. The depth 3 vs depth 2 and depth 3 vs randotron tests haven’t finished. This highlights an issue with the minimax testing as the depth 3 tests far too long. I should still be able to analyse some of the other data:

Here is the data to illustrate, I added columns D and E and then created a graph

Chart, line chart

Description automatically generated

It shows the static evaluations and how it improves for the computer until a win

I will now show some of the graphs:

Depth 1 advanced vs Ranotron (ends in a win)

Depth 1 advanced vs depth 1 vanilla (ends in stalemate 3 repeat)

Depth 2 advanced vs Ranotron (ends in win):

Depth 2 advanced vs depth 1 advanced (deterministic: win every time):

Depth 3 advanced vs depth 1 advanced (deterministic: win every time)

Depth 3 vs depth 2:

Depth 1 vanilla vs Ranotron:

I found analysing these graphs to be very interesting. It seems that both depth 1 and depth 2 can occasionally stalemate with Ranotron. It also seems that depth 1 vanilla vs advanced can stalemate. I believe that this is because they are quite well matched algorithms therefore the better one can only guarantee win or draw.

I thought It was also nice to see how depth 3 vs depth 1 got to checkmate quicker (25 moves) than depth 2 (37 moves). This shows that these algorithms are at there best when playing a rational opponent they can predict.

The antithesis of this is that, while depth 2 wins against depth 1, depth 1 wins quicker against Ranotron. This is because the minimax function assumes the opponent will play optimally. If the opponent isn’t playing rationally then carefully selected move will result in the game lasting longer than necessary.

We also see that depth 3 vs depth 2 resulted in a stalemate however depth 3 was able to maximise its score. This shows how the algorithms are relatively similar and are a match for each other (able to draw)

I hope to get the graphs for the other tests. To do this I will run the tests that haven’t finished again another time. However not finishing is still a outcome. It shows that it would not currently be possible to play a chess game vs depth 3 as it takes too long. This is due to the combinatorial explosion at a higher depth and the exponential time complexity of the algorithm.

We can also see that depth 1 vanilla beat Ranotron faster (29moves vs 37 moves) than vanilla depth 1. We also saw that even though these 2 algorithms drew in a 3 repeat stalemate, the advanced version was able to have a higher static evaluation.

All of these results show that:

* my minimax algorithm works
* a higher depth beat lower depth in effectiveness
* variable depth checking beats without in effectiveness

I also saw from manually running the algorithm with and without, that alpha beta pruning was improving performance. It is unclear weather pre-sorting child nodes has improved performance as this performance increase would only kick in at higher depths which I was unable to run before.

**Validation**:

With regards to validation, some was added in the game class and console chess program. This included:

* invalid input for users move (not chess squares) 🡪 value error
* illegal move 🡪 illegal move error
* not the users turn to go 🡪not user turn error

This validation worked well enough to allow for the console chess game to be robust.

However, the intended use for this engine is in a webserver as part of a full stack website. This means that I will need both client side and server side validation. The server side validation will be added to the flask server.

I intend to add more robust validation when I write the flask server, the chess engine will assume that the inputs it gets are valid.

Feedback from Stakeholder

I received feedback from a stakeholder who played the chess game:

They were very impressed that is could play chess and, in their game, the pawn forward 2 bug didn’t arise.

Regarding the user interface they liked:

* the fact you could see the letters and numbers at the side of the board
* the fact that if printed out the computer’s move
* the fact that it printed out check
* the fact that it correctly prevented illegal moves when in check
* the fact that it took pieces of high value when given the opportunity

They suggested:

* that I find a way of clearing the console / text output after the computer’s move so that the user only sees the one relevant board and not many older ones if they scroll up.

What I took from this feedback:

I will keep this suggestion in mind if I am unable to get the GUI to work in the next protype. I will also consider adding letters and squares to the edges of the chess board in my GUI. While the user will input their moves by clicking, this will help contextualise the move history.

I did have an issue with the minimax function where the static evaluations were not what they should have been. I realised that the static evaluation for the starting positions was not 0. To correct this I made a test and then reviewed my code. The corrections was to flip the value matrices for black pieces as the matrices were not symmetrical and were from white’s perspective. This corrections was easy as it could be made just in the pieces class.

    # this should use the position vector and value matrix to get the value of the piece

    def get\_value(self, position\_vector: Vector):

        # flip if black as matrices are all for white pieces

        if self.color == "W":

            row, column = 7-position\_vector.j, position\_vector.i

        else:

            row, column = position\_vector.j, position\_vector.i

        # return sum of inherent value + value relative to positon on board

        return self.\_value + self.\_value\_matrix[row][column]

I also attempted to refactor the King.movement\_vecotor method, accidently breaking it. This was an easy fix as I ran the unit tests which highlighted the issue.

Changes/Fixes that I now plan to make to the design or code as a result of testing and feedback

* I intend to make changes to the command line interface (CLI) to make it clearer by removing old boards
* I intend to make the minimax function more efficient so that I can run at depth 3 in a reasonable time frame.
* I intend to fix the pawn forward 2 issue and add a test to ensure that it is fixed.

Evaluation

Overall I am extremely happy with this iteration.

I feel that in this iteration I accomplished my ambitions goals and created a robust chess engine complete with a chess bot that can beat me (I believe minimax depth 3 with variable depth surpasses my average ability). With odd exceptions such as the pawn move forward 2 bug, everything is fully functional and tested. This is a lesson in how testing isn’t always perfect however, without the automated testing, I don’t think I would have succeeded.

My stakeholders seem very impressed that my program can actually play chess in an intelligent manner. I am also pleasantly surprised by how well the final product has turned out.

While my testing was not a complete success (as expected) it was invaluable. My pieces module ended up having the most undiagnosed issues so in hindsight I should have tested it more thoroughly. I realised that most of the time testing was not writing tests but debugging functions. Because tests were quick to write I approached the problem like an engineer and wrote many. I was extremely happy with the results. They allowed me to debug essential function to allow my minimax function to work.

Evidence of testing (excluding minimax tests)

Text

Description automatically generated

The interface was not a focus of the project and so it received limited time investment. However, I was able to make the interface clear to my stakeholders by using standard chess symbols for pieces and standard coordinates. Because of this I think that the interface was also, considering requirements and expectations at this state, a success.

This prototype was a success as it was able to achieve the specified aims beyond the bare minimum. This will make producing the next prototype much easier as I have already created a robust chess engine.