**Prototype 2 log:**

This is a log of my progress with the 2nd prototype, some of the initial progress is noted on the ‘overall plan’ document. One of the most important reasons for doing this is so that I can keep track of the various issues and problems I encounter and record what I tried and how I solved them.

One such problem I could talk about is the difficult to use syntax of web sockets that divided functions in 2. I intend to overcome this using promises which I will document.

I have created a user interface for my 6\*6 pawn based game using the VUE.js framework. I have currently just linked to an external JavaScript file that decides how to handle a square click event.

I now have a decision for how to approach crafting the solution that I have planned. I could do what I did for noughts and crosses where I completed the frontend and the connection with the backend and then added an text input in the backend for what move should be made. I then wrote my minimax function and substituted it into the backend adding in the brains. I think that this would be harder to do for this game as there is more information to send to the front end. This includes a larger board state matrix and an array of legal moves. I think instead I will park the frontend and work on the backend brains. I will write this program in python first so that the user inputs their move in text and then the python program responds with a move.

I would like to make a separate game module for this that emulates the frontend. It should start with an array of legal moves and then validate the users input move. If valid and not game ending it should ask another python module for the computer move, resulting game state and the new array of legal moves.

This way I can prefect the backend brains and then add them to the frontend as a final step.

In this way my game emulator text based chess game stepping stone should closely match this flow diagram for the logical flow of the frontend.

A picture containing polygon

Description automatically generated

I intend to make an abstract base class for a piece and then make a class for a pawn.

Ok update. Currently I have created a Vue frontend (just the html not the JavaScript) and now I am not sure that this is what I want.

I found a source for a login system and it seemed very hard to adapt to vue

I think that I may want to pursue and alternative.

But first I want to finish the backend, then the frontend.

I had the idea of making some skeleton code to define what method and classes should exist then creating tests for them. I want to start with tests for all legal moves as this should be easier. I will then work on testing the minimax function. I am going to look into chess libraries for python. Perhaps I can use stock fish to create a test where I test my function vs stockfish to see that my function is getting in the to 95% of moves.

I created a *function\_lib.py*:

from dataclasses import dataclass

@dataclass(frozen=True)

class Vector():

    x: int

    y: int

    # @classmethod

    # def vector\_addition(cls, v1, v2):

    #     assert all(isinstance(v, cls) for v in (v1, v2)), "Both arguments must be vector objects"

    #     return cls(

    #         x= v1.x + v2.x,

    #         y= v1.y + v2.y

    #     )

    # def \_\_add\_\_(self, other):

    #     return self.vector\_addition(self, other)

    def \_\_add\_\_(self, other):

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

        return Vector(

            x=self.x + other.x,

            y=self.y + other.y

        )

    def position\_vector\_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.x in range(8) and self.y in range(8)

I then created a *app.py* file

# imports

# built in libraries

import flask

# import os

from flask\_socketio import SocketIO

from copy import deepcopy

# my local code

# import logger as logger\_module

from game\_engine import game\_engine as ge

# define constants

# setup logger

# basedir = os.getcwd()

# LOG\_DIR = os.path.join(basedir, 'server.log')

# lgr = logger\_module.setup\_logger('server', LOG\_DIR, level='INFO')

# log\_dec = logger\_module.logging\_decorator\_factory(lgr)

# setup flask app

app = flask.Flask(\_\_name\_\_)

# setup sockets

socketio = SocketIO(app, async\_mode=None)

# define socket handlers

@socketio.on("server\_move\_request")

def handle\_server\_move\_request(msg):

    pass

# define flask api route handlers

def run\_app():

    app.run(host='127.0.0.1', port=5000, debug=True)

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

    run\_app()

I then created my *game\_engine.py* file

from dataclasses import dataclass

from function\_lib import Vector

@dataclass

class Board\_State():

    # due to the unicode characters for pieces not being on my keyboard I will use None, BP and WP to represent squares

    pieces\_matrix: list[list[str]]

    def execute\_move(self, from\_sq, to\_sq):

        return Board\_State()

    def legal\_moves\_generator(self):

        pass

    def static\_eval(self) -> float:

        pass

    def minimax\_best\_move(self):

        pass

    def outcome\_if\_over(self):

        pass

def main(board\_state: list[list[str]]) -> list:

    """This function will take the current board state as a parameter

    Its output mirror the API diagram

    move, new board state matrix, array of legal moves and outcome, is game over

    I am yet to decide if I will return all of this in one object"""

    board\_state: Board\_State = Board\_State(board\_state)

    best\_move = board\_state.minimax\_best\_move()

    new\_board\_state: Board\_State = board\_state.execute\_move(best\_move)

    over, outcome = new\_board\_state.outcome\_if\_over()

    legal\_moves = new\_board\_state.legal\_moves\_enumerator()

    legal\_moves\_and\_outcome = legal\_moves.map(

        lambda move: move, new\_board\_state.execute\_move(move).outcome\_if\_over()

    )

    return (

        best\_move, new\_board\_state, (over, outcome), tuple(legal\_moves\_and\_outcome)

    )

I then wrote a unit test that would fail *test\_game\_engine.py*

from function\_lib import Vector

from game\_engine import Board\_State

def test\_legal\_moves():

    # result

    profile = [None, 'passive pawn', None, None, None, None, 'active pawn', None]

    board\_state = Board\_State(8\*[e] for e in profile)

    legal\_moves = board\_state.legal\_moves\_generator()

    # correct

    expected\_legal\_moves = [

        [Vector(0, 1), Vector(0, 2)],

        [Vector(1, 1), Vector(1, 2)],

        [Vector(2, 1), Vector(2, 2)],

        [Vector(3, 1), Vector(3, 2)],

        [Vector(4, 1), Vector(4, 2)],

        [Vector(5, 1), Vector(5, 2)],

        [Vector(6, 1), Vector(6, 2)],

        [Vector(7, 1), Vector(7, 2)]

    ]

    assert sorted(legal\_moves) == sorted(expected\_legal\_moves)

This resulted in the following failure

PS C:\Users\henry\Documents\computing coursework\prototype 2\socket server> python -m pytest

===

platform win32 -- Python 3.10.0, pytest-7.1.3, pluggy-1.0.0

rootdir: C:\Users\henry\Documents\computing coursework\prototype 2\socket server

collected 1 item

test\_game\_engine.py F                                                                                                                                                                          [100%]

============================================================================================= FAILURES ==============================================================================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_legal\_moves \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

    def test\_legal\_moves():

        # result

        profile = [None, 'passive pawn', None, None, None, None, 'active pawn', None]

        board\_state = Board\_State(8\*[e] for e in profile)

        legal\_moves = board\_state.legal\_moves\_generator()

        # correct

        expected\_legal\_moves = [

            [Vector(0, 1), Vector(0, 2)],

            [Vector(1, 1), Vector(1, 2)],

            [Vector(2, 1), Vector(2, 2)],

            [Vector(3, 1), Vector(3, 2)],

            [Vector(4, 1), Vector(4, 2)],

            [Vector(5, 1), Vector(5, 2)],

            [Vector(6, 1), Vector(6, 2)],

            [Vector(7, 1), Vector(7, 2)]

        ]

>       assert sorted(legal\_moves) == sorted(expected\_legal\_moves)

E       TypeError: 'NoneType' object is not iterable

test\_game\_engine.py:23: TypeError

====================================================================================== short test summary info ======================================================================================

FAILED test\_game\_engine.py::test\_legal\_moves - TypeError: 'NoneType' object is not iterable

========================================================================================= 1 failed in 0.30s =========================================================================================

PS C:\Users\henry\Documents\computing coursework\prototype 2\socket server>

I then wrote some code that I thought would work to make the test succeed.

    def legal\_moves\_generator(self):

        for i, row in enumerate(self.pieces\_matrix):

            for j, square in enumerate(row):

                if square == 'AP':

                    position\_vector = Vector(i, j)

                    # print(f"CALLED:  get\_pawn\_movement\_vectors(position\_vector={position\_vector}, board\_state={repr(self)})")

                    movement\_vectors = get\_pawn\_movement\_vectors(position\_vector=position\_vector, board\_state=self)

                    print(f"RETURNED:  get\_pawn\_movement\_vectors(position\_vector={position\_vector})  RETURNED  {list(movement\_vectors)}")

                    if movement\_vectors is not None:

                        yield from movement\_vectors

def get\_pawn\_movement\_vectors(position\_vector, board\_state: Board\_State):

    foreword\_v = (position\_vector + Vector(0, 1))

    if foreword\_v.position\_vector\_in\_board():

        square = board\_state.pieces\_matrix[foreword\_v.i][foreword\_v.j]

        if square is None:

            yield (position\_vector, foreword\_v)

The test still failed.

PS C:\Users\henry\Documents\computing coursework\prototype 2\socket server> python -m pytest

============================================================================ test session starts ==================================================================

platform win32 -- Python 3.10.0, pytest-7.1.3, pluggy-1.0.0

rootdir: C:\Users\henry\Documents\computing coursework\prototype 2\socket server

collected 1 item

test\_game\_engine.py F

================================================================================= FAILURES ========================================================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_legal\_moves\_initial\_positions \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

    def test\_legal\_moves\_initial\_positions():

        # result

        profile = [None, 'AP', None, None, None, None, 'AP', None]

        board\_state = Board\_State([8\*[e] for e in profile])

        legal\_moves = board\_state.legal\_moves\_generator()

        # correct

        expected\_legal\_moves = [

            [Vector(0, 1), Vector(0, 2)],

            [Vector(1, 1), Vector(1, 2)],

            [Vector(2, 1), Vector(2, 2)],

            [Vector(3, 1), Vector(3, 2)],

            [Vector(4, 1), Vector(4, 2)],

            [Vector(5, 1), Vector(5, 2)],

            [Vector(6, 1), Vector(6, 2)],

            [Vector(7, 1), Vector(7, 2)]

        ]

        # sort by alphabetical order of the repr form of the vector (arbitrary but should work)

        # key = lambda v: repr(v)

  []

  []

RETURNED:  get\_pawn\_movement\_vectors(position\_vector=Vector(i=1, j=6))  RETURNED  []

RETURNED:  get\_pawn\_movement\_vectors(position\_vector=Vector(i=1, j=7))  RETURNED  []

RETURNED:  get\_pawn\_movement\_vectors(position\_vector=Vector(i=6, j=0))  RETURNED  []

RETURNED:  get\_pawn\_movement\_vectors(position\_vector=Vector(i=6, j=1))  RETURNED  []

RETURNED:  get\_pawn\_movement\_vectors(position\_vector=Vector(i=6, j=2))  RETURNED  []

RETURNED:  get\_pawn\_movement\_vectors(position\_vector=Vector(i=6, j=3))  RETURNED  []

RETURNED:  get\_pawn\_movement\_vectors(position\_vector=Vector(i=6, j=4))  RETURNED  []

RETURNED:  get\_pawn\_movement\_vectors(position\_vector=Vector(i=6, j=5))  RETURNED  []

RETURNED:  get\_pawn\_movement\_vectors(position\_vector=Vector(i=6, j=6))  RETURNED  []

RETURNED:  get\_pawn\_movement\_vectors(position\_vector=Vector(i=6, j=7))  RETURNED  []

========================================================================== short test summary info ==========================================================================

FAILED test\_game\_engine.py::test\_legal\_moves\_initial\_positions - assert [] == [[Vector(i=0,...5, j=2)], ...]

============================================================================= 1 failed in 0.34s =============================================================================

PS C:\Users\henry\Documents\computing coursework\prototype 2\socket server>

The print statement I added reviled that I had some issue with the position vectors being invalid.

I can see a problem. When I enumerate I say that the row index is x and the column index is y. This is the logic error. This was fixed as below

    def legal\_moves\_generator(self):

        for j, row in enumerate(self.pieces\_matrix):

            for i, square in enumerate(row):

                if square == 'AP':

                    position\_vector = Vector(i, j)

                    # print(f"CALLED:  get\_pawn\_movement\_vectors(position\_vector={position\_vector}, board\_state={repr(self)})")

                    movement\_vectors = get\_pawn\_movement\_vectors(position\_vector=position\_vector, board\_state=self)

                    print(f"RETURNED:  get\_pawn\_movement\_vectors(position\_vector={position\_vector})  RETURNED  {list(movement\_vectors)}")

                    if movement\_vectors is not None:

                        yield from movement\_vectors

Now the test result is:

    def legal\_moves\_generator(self):

        for j, row in enumerate(self.pieces\_matrix):

            for i, square in enumerate(row):

                print(f"examined square {(i, j)}")

I then checked that the squares were all being examined in a systematic way and they were.

examined square (0, 0)

examined square (1, 0)

examined square (2, 0)

examined square (3, 0)

examined square (4, 0)

examined square (5, 0)

examined square (6, 0)

examined square (7, 0)

examined square (0, 1)

examined square (1, 1)

examined square (2, 1)

examined square (3, 1)

examined square (4, 1)

examined square (5, 1)

examined square (6, 1)

examined square (7, 1)

examined square (0, 2)

examined square (1, 2)

examined square (2, 2)

examined square (3, 2)

examined square (4, 2)

examined square (5, 2)

examined square (6, 2)

examined square (7, 2)

examined square (0, 3)

examined square (1, 3)

examined square (2, 3)

examined square (3, 3)

examined square (4, 3)

examined square (5, 3)

examined square (6, 3)

examined square (7, 3)

examined square (0, 4)

examined square (1, 4)

examined square (2, 4)

examined square (3, 4)

examined square (4, 4)

examined square (5, 4)

examined square (6, 4)

examined square (7, 4)

examined square (0, 5)

examined square (1, 5)

examined square (2, 5)

examined square (3, 5)

examined square (4, 5)

examined square (5, 5)

examined square (6, 5)

examined square (7, 5)

examined square (0, 6)

examined square (1, 6)

examined square (2, 6)

examined square (3, 6)

examined square (4, 6)

examined square (5, 6)

examined square (6, 6)

examined square (7, 6)

examined square (0, 7)

examined square (1, 7)

examined square (2, 7)

examined square (3, 7)

examined square (4, 7)

examined square (5, 7)

examined square (6, 7)

examined square (7, 7)

this shows that the squares are being successfully examined. I now need to see if it correctly identifies all the squares that contain active pawns.

    def legal\_moves\_generator(self):

        for j, row in enumerate(self.pieces\_matrix):

            for i, square in enumerate(row):

                if square == 'AP':

                    print(f"square with active pawn {(i, j)}")

The result was

square with active pawn (0, 1)

square with active pawn (1, 1)

square with active pawn (2, 1)

square with active pawn (3, 1)

square with active pawn (4, 1)

square with active pawn (5, 1)

square with active pawn (6, 1)

square with active pawn (7, 1)

square with active pawn (0, 6)

square with active pawn (1, 6)

square with active pawn (2, 6)

square with active pawn (3, 6)

square with active pawn (4, 6)

square with active pawn (5, 6)

square with active pawn (6, 6)

square with active pawn (7, 6)

This doesn’t seem to be correct. This could be the source of the logic error



I used the debugging function to see the value of the legal\_moves function.

I added some code to store the variable content in a json file so I could more easily examine it.

    serialised\_legal\_moves = tuple(((v1.i, v1.j), (v2.i, v2.j)) for v1, v2 in legal\_moves)

    with open("legal\_moves.json", "w") as file:

        file.write(

            json.dumps(

                serialised\_legal\_moves

            )

        )

This was the result

[

    [[0, 1], [0, 2]],

    [[0, 6], [0, 7]],

    [[1, 1], [1, 2]],

    [[1, 6], [1, 7]],

    [[2, 1], [2, 2]],

    [[2, 6], [2, 7]],

    [[3, 1], [3, 2]],

    [[3, 6], [3, 7]],

    [[4, 1], [4, 2]],

    [[4, 6], [4, 7]],

    [[5, 1], [5, 2]],

    [[5, 6], [5, 7]],

    [[6, 1], [6, 2]],

    [[6, 6], [6, 7]],

    [[7, 1], [7, 2]],

    [[7, 6], [7, 7]]

]

For each pawn there is an additional vector.

Oh, I realise, I accidently added active pawns to both sides.

Let me rectify this by changing the test data to represent the test it is meant to

    profile = [None, 'AP', None, None, None, None, 'PP', None]

    board\_state = Board\_State([8\*[e] for e in profile])

this prevented the program from thinking there were multiple active pawns in each column. Now it still isn’t working.

Here is precisely what it failing.

The test for legal moves from initial positions is failing.

Here was the test input and expected output

    board\_state = Board\_State([

        [None, None, None, None, None, None, None, None],

        ['PP', 'PP', 'PP', 'PP', 'PP', 'PP', 'PP', 'PP'],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        ['AP', 'AP', 'AP', 'AP', 'AP', 'AP', 'AP', 'AP'],

        [None, None, None, None, None, None, None, None]

    ])

    legal\_moves = board\_state.legal\_moves\_generator()

    # correct

    expected\_legal\_moves = (

        (Vector(0, 1), Vector(0, 2)),

        (Vector(1, 1), Vector(1, 2)),

        (Vector(2, 1), Vector(2, 2)),

        (Vector(3, 1), Vector(3, 2)),

        (Vector(4, 1), Vector(4, 2)),

        (Vector(5, 1), Vector(5, 2)),

        (Vector(6, 1), Vector(6, 2)),

        (Vector(7, 1), Vector(7, 2)),

    )

Here was the actual output

[

    "Vector(i=0, j=6)  -->  Vector(i=0, j=7)",

    "Vector(i=1, j=6)  -->  Vector(i=1, j=7)",

    "Vector(i=2, j=6)  -->  Vector(i=2, j=7)",

    "Vector(i=3, j=6)  -->  Vector(i=3, j=7)",

    "Vector(i=4, j=6)  -->  Vector(i=4, j=7)",

    "Vector(i=5, j=6)  -->  Vector(i=5, j=7)",

    "Vector(i=6, j=6)  -->  Vector(i=6, j=7)",

    "Vector(i=7, j=6)  -->  Vector(i=7, j=7)"

]

For the test with random positions. Here was the test with the input and expected output.

    legal\_moves = Board\_State([

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, 'AP'],

        [None, None, None, 'AP', None, None, 'AP', None],

        [None, None, None, None, 'AP', None, None, None],

        [None, 'AP', None, None, None, None, None, None],

        ['AP', None, 'AP', None, None, 'AP', None, None],

        [None, None, None, None, None, None, None, None]

    ]).legal\_moves\_generator()

    expected\_legal\_moves = [

        [Vector(0, 1), Vector(0, 2)],

        [Vector(1, 2), Vector(1, 3)],

        [Vector(2, 1), Vector(2, 2)],

        [Vector(3, 4), Vector(3, 5)],

        [Vector(4, 3), Vector(4, 4)],

        [Vector(5, 1), Vector(5, 2)],

        [Vector(6, 4), Vector(6, 5)],

        [Vector(7, 5), Vector(7, 6)]

    ]

And here was the actual output

[

    "Vector(i=0, j=6)  -->  Vector(i=0, j=7)",

    "Vector(i=1, j=5)  -->  Vector(i=1, j=6)",

    "Vector(i=2, j=6)  -->  Vector(i=2, j=7)",

    "Vector(i=3, j=3)  -->  Vector(i=3, j=4)",

    "Vector(i=4, j=4)  -->  Vector(i=4, j=5)",

    "Vector(i=5, j=6)  -->  Vector(i=5, j=7)",

    "Vector(i=6, j=3)  -->  Vector(i=6, j=4)",

    "Vector(i=7, j=2)  -->  Vector(i=7, j=3)"

]

I can see a pattern in the difference between the expected and actual output.

The program identifies the pawns to be in the right column but in the wrong row. For instance in the initial positions the pawns are in the row second from bottom (1) but the program thinks they are in the row second form the top (6). This makes sense as **the row with index 0 is at the top and not the bottom**. I want to correct this.

I will create a test to insure that the issue is corrected. It will check that the program can correctly identify the position vectors of all pawns.

I wrote some new code in my game engine file to divide the logic of identifying pawn position vectors and identifying their movement vectors

    def 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 pawn\_position\_vector\_generator(self):

        for i, j in product(range(8), range(8)):

            position\_vector = Vector(i, j)

            if self.piece\_at\_vector(position\_vector) == "AP":

                yield position\_vector

    def legal\_moves\_generator(self):

        for pawn\_position\_vector in self.pawn\_position\_vector\_generator():

            movement\_vectors = get\_pawn\_movement\_vectors(pawn\_position\_vector, self)

            # if not None or empty

            if movement\_vectors:

                yield from movement\_vectors

I then wrote a new test

def test\_gen\_position\_vectors():

    result\_pawn\_position\_vectors = board\_state = sorted(

        Board\_State([

            [None, None, None, None, None, None, None, None],

            ['PP', 'PP', 'PP', 'PP', 'PP', 'PP', 'PP', 'PP'],

            [None, None, None, None, None, None, None, None],

            [None, None, None, None, None, None, None, None],

            [None, None, None, None, None, None, None, None],

            [None, None, None, None, None, None, None, None],

            ['AP', 'AP', 'AP', 'AP', 'AP', 'AP', 'AP', 'AP'],

            [None, None, None, None, None, None, None, None]

        ]).pawn\_position\_vector\_generator(),

        key=repr

    )

    expected\_pawn\_position\_vector = sorted(

        [

            Vector(0, 1),

            Vector(1, 1),

            Vector(2, 1),

            Vector(3, 1),

            Vector(4, 1),

            Vector(5, 1),

            Vector(6, 1),

            Vector(7, 1)

        ],

        key=repr

    )

    assert result\_pawn\_position\_vectors == expected\_pawn\_position\_vector

The test fails but I used a breakpoint to determine the contents of the variables

Text

Description automatically generated

This shows the issue. I will now try to fix it by changing the function that gets a piece from the board at a position vector

    def 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]

This flips how i vectors are converted to rows. i=0 now should correspond to the 8th (bottom) row. This should allow me to make vectors from the bottom left corner.

I have made a minor change to ensure that the format of all the array was the list datatype not the tuple data type. Now all the tests pass.

PS C:\Users\henry\Documents\computing coursework\prototype 2\socket server> python -m pytest

============================================================================= test session starts ==============================================================================

platform win32 -- Python 3.10.0, pytest-7.1.3, pluggy-1.0.0

rootdir: C:\Users\henry\Documents\computing coursework\prototype 2\socket server

collected 3 items

test\_game\_engine.py ...                                                                                                                                                   [100%]

============================================================================== 3 passed in 0.07s ===============================================================================

PS C:\Users\henry\Documents\computing coursework\prototype 2\socket server>

Here is the code as it stands currently

Game engine

from dataclasses import dataclass

from itertools import product

from turtle import position

from function\_lib import Vector

@dataclass

class Board\_State():

    # due to the unicode characters for pieces not being on my keyboard I will use None, BP and WP to represent squares

    pieces\_matrix: list[list[str]]

    def execute\_move(self, from\_sq, to\_sq):

        return Board\_State()

    # def legal\_moves\_generator(self):

    #     for j, row in enumerate(self.pieces\_matrix):

    #         for i, square in enumerate(row):

    #             if square == 'AP':

    #                 # print(f"square with active pawn {(i, j)}")

    #                 position\_vector = Vector(i, j)

    #                 # print(f"CALLED:  get\_pawn\_movement\_vectors(position\_vector={position\_vector}, board\_state={repr(self)})")

    #                 movement\_vectors = get\_pawn\_movement\_vectors(position\_vector=position\_vector, board\_state=self)

    #                 # print(f"RETURNED:  get\_pawn\_movement\_vectors(position\_vector={position\_vector})  RETURNED  {list(movement\_vectors)}")

    #                 if movement\_vectors is not None:

    #                     yield from movement\_vectors

    def 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 pawn\_position\_vector\_generator(self):

        for i, j in product(range(8), range(8)):

            position\_vector = Vector(i, j)

            if self.piece\_at\_vector(position\_vector) == "AP":

                yield position\_vector

    def legal\_moves\_generator(self):

        for pawn\_position\_vector in self.pawn\_position\_vector\_generator():

            movement\_vectors = get\_pawn\_movement\_vectors(pawn\_position\_vector, self)

            # if not None or empty

            if movement\_vectors:

                yield from movement\_vectors

    def static\_eval(self) -> float:

        pass

    def minimax\_best\_move(self):

        pass

    def outcome\_if\_over(self):

        pass

def get\_pawn\_movement\_vectors(position\_vector, board\_state: Board\_State):

    foreword\_v = (position\_vector + Vector(0, 1))

    if foreword\_v.position\_vector\_in\_board():

        # square = board\_state.pieces\_matrix[foreword\_v.i][foreword\_v.j]

        square = board\_state.piece\_at\_vector(foreword\_v)

        if square is None:

            yield (position\_vector, foreword\_v)

def main(board\_state: list[list[str]]) -> list:

    """This function will take the current board state as a parameter

    Its output mirror the API diagram

    move, new board state matrix, array of legal moves and outcome, is game over

    I am yet to decide if I will return all of this in one object"""

    board\_state: Board\_State = Board\_State(board\_state)

    best\_move = board\_state.minimax\_best\_move()

    new\_board\_state: Board\_State = board\_state.execute\_move(best\_move)

    over, outcome = new\_board\_state.outcome\_if\_over()

    legal\_moves = new\_board\_state.legal\_moves\_enumerator()

    legal\_moves\_and\_outcome = legal\_moves.map(

        lambda move: (move, new\_board\_state.execute\_move(move).outcome\_if\_over())

    )

    return (

        best\_move, new\_board\_state, (over, outcome), tuple(legal\_moves\_and\_outcome)

    )

Test game engine

import json

from function\_lib import Vector

from game\_engine import Board\_State

def test\_gen\_position\_vectors():

    result\_pawn\_position\_vectors = board\_state = sorted(

        Board\_State([

            [None, None, None, None, None, None, None, None],

            ['PP', 'PP', 'PP', 'PP', 'PP', 'PP', 'PP', 'PP'],

            [None, None, None, None, None, None, None, None],

            [None, None, None, None, None, None, None, None],

            [None, None, None, None, None, None, None, None],

            [None, None, None, None, None, None, None, None],

            ['AP', 'AP', 'AP', 'AP', 'AP', 'AP', 'AP', 'AP'],

            [None, None, None, None, None, None, None, None]

        ]).pawn\_position\_vector\_generator(),

        key=repr

    )

    expected\_pawn\_position\_vector = sorted(

        [

            Vector(0, 1),

            Vector(1, 1),

            Vector(2, 1),

            Vector(3, 1),

            Vector(4, 1),

            Vector(5, 1),

            Vector(6, 1),

            Vector(7, 1)

        ],

        key=repr

    )

    assert result\_pawn\_position\_vectors == expected\_pawn\_position\_vector

def test\_legal\_moves\_initial\_positions():

    # result

    # profile = [None, 'PP', None, None, None, None, 'AP', None]

    # board\_state = Board\_State([8\*[e] for e in profile])

    board\_state = Board\_State([

        [None, None, None, None, None, None, None, None],

        ['PP', 'PP', 'PP', 'PP', 'PP', 'PP', 'PP', 'PP'],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        ['AP', 'AP', 'AP', 'AP', 'AP', 'AP', 'AP', 'AP'],

        [None, None, None, None, None, None, None, None]

    ])

    legal\_moves = board\_state.legal\_moves\_generator()

    # correct

    expected\_legal\_moves = (

        (Vector(0, 1), Vector(0, 2)),

        (Vector(1, 1), Vector(1, 2)),

        (Vector(2, 1), Vector(2, 2)),

        (Vector(3, 1), Vector(3, 2)),

        (Vector(4, 1), Vector(4, 2)),

        (Vector(5, 1), Vector(5, 2)),

        (Vector(6, 1), Vector(6, 2)),

        (Vector(7, 1), Vector(7, 2)),

    )

    # sort by alphabetical order of the repr form of the vector (arbitrary but should work)

    # key = lambda v: repr(v)

    key = repr

    legal\_moves, expected\_legal\_moves = sorted(legal\_moves, key=key), sorted(expected\_legal\_moves, key=key)

    # serialised\_legal\_moves = tuple(((v1.i, v1.j), (v2.i, v2.j)) for v1, v2 in legal\_moves)

    serialised\_legal\_moves = tuple(f"{repr(v1)}  -->  {repr(v2)}" for v1, v2 in legal\_moves)

    with open("legal\_moves\_initial.json", "w") as file:

        file.write(

            json.dumps(

                serialised\_legal\_moves

            )

        )

    assert legal\_moves == expected\_legal\_moves

def test\_legal\_moves\_random\_positions():

    legal\_moves = map(

        list,

        sorted(

            Board\_State([

                [None, None, None, None, None, None, None, None],

                [None, None, None, None, None, None, None, None],

                [None, None, None, None, None, None, None, 'AP'],

                [None, None, None, 'AP', None, None, 'AP', None],

                [None, None, None, None, 'AP', None, None, None],

                [None, 'AP', None, None, None, None, None, None],

                ['AP', None, 'AP', None, None, 'AP', None, None],

                [None, None, None, None, None, None, None, None]

            ]).legal\_moves\_generator(),

            key=repr

        )

    )

    expected\_legal\_moves = sorted(

        [

            [Vector(0, 1), Vector(0, 2)],

            [Vector(1, 2), Vector(1, 3)],

            [Vector(2, 1), Vector(2, 2)],

            [Vector(3, 4), Vector(3, 5)],

            [Vector(4, 3), Vector(4, 4)],

            [Vector(5, 1), Vector(5, 2)],

            [Vector(6, 4), Vector(6, 5)],

            [Vector(7, 5), Vector(7, 6)]

        ],

        key=repr

    )

    for result, expected in zip(legal\_moves, expected\_legal\_moves):

        # assert result == expected, f"at column {i} result != expected:   {result} != {expected}"

        assert result == expected, f"{result} != {expected}"

    # assert legal\_moves == expected\_legal\_moves

I will now add logic and corresponding tests for the pawn movement where they take another piece.

I have created a test that includes both take and foreword movements

def test\_take\_vectors():

    legal\_moves = map(

        list,

        sorted(

            Board\_State([

                [None, None, None, None, None, None, None, None],

                [None, None, None, None, None, None, None, None],

                [None, None, None, None, None, None, "PP", None],

                [None, None, None, "PP", None, None, "AP", "AP"],

                [None, None, "AP", None, "AP", None, None, None],

                ["AP", None, None, None, None, None, None, None],

                [None, None, None, None, None, None, None, None],

                [None, None, None, None, None, None, None, None]

            ]).legal\_moves\_generator(),

            key=repr

        )

    )

    # includes foreword adn diagonal takes

    expected\_legal\_moves = sorted(

        [

            [Vector(0, 2), Vector(0, 3)],

            [Vector(2, 3), Vector(2, 4)],

            [Vector(2, 3), Vector(3, 4)],

            [Vector(4, 3), Vector(4, 4)],

            [Vector(4, 3), Vector(3, 4)],

            [Vector(7, 4), Vector(7, 5)],

            [Vector(7, 4), Vector(6, 5)],

        ],

        key=repr

    )

    for result, expected in zip(legal\_moves, expected\_legal\_moves):

        assert result == expected, f"{result} != {expected}"

It fails at the moment as the functionality hasn’t been added in yet

Let me try to add this

def get\_pawn\_movement\_vectors(position\_vector, board\_state: Board\_State):

    foreword\_v = position\_vector + Vector(0, 1)

    if foreword\_v.position\_vector\_in\_board():

        # square = board\_state.pieces\_matrix[foreword\_v.i][foreword\_v.j]

        square = board\_state.piece\_at\_vector(foreword\_v)

        if square is None:

            yield [position\_vector, foreword\_v]

    diagonal\_takes = (

        position\_vector + Vector(-1, 1),

        position\_vector + Vector(1, 1),

    )

    for diagonal\_take in diagonal\_takes:

        if diagonal\_take.position\_vector\_in\_board():

            square = board\_state.piece\_at\_vector(diagonal\_take)

            if square == "PP":

                yield [position\_vector, diagonal\_take]

Now with some minor changes to the test all 4 pass.

Now I want to make a test for the execute move function

Once I get this test to work I will change all my code to use tuples not lists.

My 5th test also works.

Test:

def test\_execute\_move():

    resulting\_game\_state = Board\_State([

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, 'AP'],

        [None, None, None, 'AP', None, None, 'AP', None],

        [None, None, None, None, 'AP', None, None, None],

        [None, 'AP', None, None, None, None, None, None],

        ['AP', None, 'AP', None, None, 'AP', None, None],

        [None, None, None, None, None, None, None, None]

    ]).execute\_move(

        Vector(0, 1),

        Vector(0, 2)

    )

    expected\_game\_state = Board\_State([

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, 'AP'],

        [None, None, None, 'AP', None, None, 'AP', None],

        [None, None, None, None, 'AP', None, None, None],

        ['AP', 'AP', None, None, None, None, None, None],

        [None, None, 'AP', None, None, 'AP', None, None],

        [None, None, None, None, None, None, None, None]

    ])

    assert resulting\_game\_state == expected\_game\_state

Then here is the code

    def execute\_move(self, from\_sq: Vector, to\_sq: Vector):

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

        # set from to blank

        row, col = 7-from\_sq.j, from\_sq.i

        current\_piece = new\_pieces\_matrix[row][col]

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

        # set to square to this piece

        row, col = 7-to\_sq.j, to\_sq.i

        new\_pieces\_matrix[row][col] = current\_piece

        return Board\_State(

            pieces\_matrix=new\_pieces\_matrix

        )

Now I will change all the code to use tuples

I now need to transitions away from active pawns and passive pawns and towards computer and user pawns.

One of the things that I find funny about this project is that it is amazing how often progress takes the form of pursuing one path and failing, with no useful solution produced, only to gain a deeper understanding of the problem and how better to tackle it.

My first idea for the second prototype was to use VUE.js to create a webpage that would make it easy to render the current game state onto a page dynamically. This would allow me to change one of the variable values in order to change the page itself. My plan was to use JavaScript and WebSocket in order to exchange data and move requests. This may still work.

I then found a page that gave me a better idea of what flask logins could look like. <https://www.digitalocean.com/community/tutorials/how-to-add-authentication-to-your-app-with-flask-login>

This drove me to instead pursue a more HTTP heavy solution that used multiple pages, a flask solution to login and a rest api for communication.

I then tried to tackle the problem from the backend first. My idea was to make a working chess game that only had pawns on the board. I would use the console to play it. Once the backend was working I would then salvage some of the work in VUE to make my html webpage interface. Once I had connected my frontend and backend I would then be able to play the game with the frontend only. I could then work to add in other pieces such as rooks, bishops etc.

This lead me to try to breakdown the problem of a chess engine into submodules to try to tackle it in python. I got some unit tests working but ultimately the approach was flawed. These tests tackled a problem where there were passive pawns at the top of the board and active pawns at the bottom of the board. By iterating through passive pawns and generating vectors I then created a generator for all legal moves. I then realised that I needed to change the object such that there were pieces owned by the computer and pieces owned by the user. There would also need to be an attribute for next to go.

In the process I realised how complicated pawns are.

* Firstly they are the only piece which has different movement vectors based on its colour (0, 1) or (0, -1).
* They are also the only piece that has a concept of memory. They are allowed to move foreword 2 only if this is their fist move.
* They also care about other pieces last move as they can do on passion on a pawn
* They also can turn into a different piece when they reach the end.

I will now again reconsider my approach.

I want to create a system of classes to encode the behaviour of each piece.

I want to have an abstract base class (no instances made for inheritance) for a generic piece. This will dictate the public methods and attributes required. I will then create classes that inherit from these classes to represent each piece.

I will call this prototype v2.3

At this point it is all coming together.

I have used an abstract base class to represent a chess piece It allows me to not repeat functionality that they share and to dictate the attributes and functions (object interface) that each subclass should have. I have also made a class for a game state. It uses a piece matrix and is able to:

* Determine all legal moves (accounting for check)
* Recognise checkmate and stalemate
* Perform static evaluations on the current game
* Make a move to result in a new child game state

I now want to take the logic I have created and the discoveries I have made into Version 3.4: this will allow me to divide the game state class up further.

I want to make a Game class that contains an array of move objects, the current game state, a list of all moves made so far.

A move object will allow me to encode in a standardised way how a move affects the board. It should allow for custom moves like castling ect.

A game with a list of moves will allow me to reverse a move or add a move.

I will change the game state class to allow for movements to be encoded in chess coordinates.

I will try to make all the objects serializable to help with tests.

I will try to write a version of minimax now I have all the subcomponents needed.

I will try to integrate some functions from the chess library to, for example, print out the board.

This should allow me to play a full game against the computer. I may use tkinter to do this.

Here is all the code from version 2.3:

(remains unchanged in folder. Not sure how I will show it)

I realise there are some logic errors. I want to add in the logic into v2.4 testing as I go.

So my last prototype was pretty good. It captured the main logic of:

* Determining legal moves
* Handling check
* Static evaluations

What I now want to do is recreate this functionality and recycle my code for v2.4. I will use a slightly different system of classes with different systems of classes and I will test as I go in order to check that all my functions are working. I found before that test driven development was hard when I was testing as I wrote the code. However this time I am doing a major refactor of the code and testing as I go.

Some of my tests. (such as tests for the vector class) can be done with standard techniques that I have used in the past. This involves defining a separate python file for the tests and defining a function for each test where I use assert statements to ensure all behaviour is as expected. To detect logic errors more easily I may also need to think of adding validation to major functions and classes in addition to trying to validate all user input. This internal validation where I don’t assume that at this point all data and input is valid is not needed everywhere, but should help tackle logic errors efficiently as the project grows in size.

For other tests I want to create some code to allow me to store my test cases in an excel file. The code should then read my standardised excel file and then check that the output is as expected. This should reduce repletion and code volume for testing.

I have found a relevant library to my problem

[**https://ddt.readthedocs.io/en/latest/example.html**](https://ddt.readthedocs.io/en/latest/example.html)

I may not use it but reading the code that you can create with it did give me an idea. I could make a DIY version for my own purposes if necessary that read from an excel file. It relied on the principle of defining a standard unit-test test function but one that took arguments for what data was to be tested. It then executed a single test. Then decorators where applied to use the function to produce a procedure that completed he original function interactively across a given array or file of data.

OK here is the plan.

I am going to create a variety of testcases as I go in excel files. I will write code to read the file as a data frame with pandas. I will then create a decorator to repeat the test for each row. This code will be a challenge but it will be general and reusable. I will then write specific unit test functions using this data. I will also create a function to read a chess board from an excel file and encode it as a string in chess notation to use in tests.

<https://en.wikipedia.org/wiki/Descriptive_notation>

So I will now try to build my own automated testing strategy.

I will use this source to help me read excel sheets

<https://www.geeksforgeeks.org/working-with-excel-files-using-pandas/>

I am still undecided as to how I will represent a chess board state.

Actually considering this problem it would still be hard to use Excel, maybe I will just use yaml or json. That way I can use an existing framework.

<https://ddt.readthedocs.io/en/latest/example.html>

I have now created some unit tests and they work. I am having an issue structuring the directories in a logical way without breaking the python imports. I want to learn how to use an \_\_init\_\_.py to create a python package to make imports easier.

PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\main> tree /f

Folder PATH listing for volume OS

Volume serial number is 4ED8-B070

C:.

│   delete\_me.py

│   v2.4.code-workspace

│

├───test

│   │   test\_vector.py

│   │   \_\_init\_\_.py

│   │

│   ├───test\_vector\_data

│   │       from\_square.yaml

│   │       vector\_add.yaml

│   │       vector\_in\_board.yaml

│   │

│   └───\_\_pycache\_\_

│           test\_vector.cpython-310.pyc

│           \_\_init\_\_.cpython-310.pyc

│

└───vector

    │   vector.py

    │   \_\_init\_\_.py

    │

    └───\_\_pycache\_\_

            vector.cpython-310.pyc

            \_\_init\_\_.cpython-310.pyc

PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\main> python -m unittest test.test\_vector

...

----------------------------------------------------------------------

Ran 3 tests in 0.000s

OK

PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\main> python delete\_me.py

Vector(i=1, j=1)

PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\main>

Above is one valid structure where all the import work.

I will stop vector being a package as it is only one script.

PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\main> tree /f

Folder PATH listing for volume OS

Volume serial number is 4ED8-B070

│   delete\_me.py

│   v2.4.code-workspace

│   vector.py

│

├───test

│   │   \_\_init\_\_.py

│   │

│   ├───test\_vector\_data

│   │       from\_square.yaml

│   │       vector\_add.yaml

│   │       vector\_in\_board.yaml

│   │

│   └───\_\_pycache\_\_

│           test\_vector.cpython-310.pyc

│           \_\_init\_\_.cpython-310.pyc

│

└───\_\_pycache\_\_

        vector.cpython-310.pyc

PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\main> python -m unittest test.test\_vector

...

----------------------------------------------------------------------

Ran 3 tests in 0.002s

OK

PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\main> python delete\_me.py

Vector(i=1, j=1)

PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\main>

Although as the python test are in their own file and they, rather uniquely need to import a file that is in a parent directory, I cannot run the test file directly. The unittest command is used as it adds a path to the sys.path list of import paths.

I am going to adopt the strategy of having tests and python scripts in the same folder. I can use the discover feature to search for tests so I should still be able to run many tests across many directories automatically.

I will now begin adding in pieces and the board with refactoring and tests. I need tests as I know that there were some logic errors that I now need to fix.

So I am glad that I started with the vector module as the changes that I wanted to make to it were minimal. This gave me an opportunity to learn how to make unit tests in the way that I had planned. Now I need to refactor my game state class and then create tests for it.

I want to make some changes.

For one thing. I want to in future differentiate the game, which involves players and the game state which involves simply colours. I also want to be able to instantiate pieces form a string so “BK” will be black king.

So I will need to first make changes to the game state and pieces modules. To do this I will use a jupyter notebook to ensure my changes sort of work (and as part of the experimentation). Then I will write the unit-tests.

I now know that I cannot refactor before I know what my planned changes are.

I have refactored the pieces module.

* I removed redundancy as each of the piece class had each defined a similar function to determine the contents of a square in terms of empty, ally or enemy. I have created on function for this in the base class Piece.
* I have simplified the logic to check vectors for pieces like bishop, queen, rook
* I have removed ownership

I should still be able to write unit tests for this library alone.

So I have created my first test for the pieces module. It has one function to execute the test and a corresponding yaml file to contain the test data. It works by starting with an empty board. Then is supposes that a white piece of a certain class is at a given square. Then the test asserts that the generate movement vectors function is as expected.

Python code

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 TestCase(unittest.TestCase):

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

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

        position\_vector = Vector.position\_vector\_from\_chess\_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).position\_vector\_to\_square(),

                movement\_vectors

            )

        )

        self.assertEqual(

            set(resultant\_squares),

            set(move\_squares),

            msg=f"Expected movement squares {sorted(move\_squares)}  !=  actual movement squares {sorted(resultant\_squares)}\n{repr(piece)} at {square}"

        )

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

    unittest.main()

yaml file of test data:

test:

  piece\_type: 'N'

  square: E4

  move\_squares: [

    D2,

    F2,

    D6,

    F6,

    C5,

    C3,

    G5,

    G3

  ]

test:

  piece\_type: Q

  square: C3

  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

]

I started with the test for the knight and it worked.

But my test for the queen doesn’t work. The function actually returns only [A1, B2].

I cannot easily tell why this is. I suspect that it is down to the break statement within the match case not properly breaking out of the for loop as expected.

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)

        for unit\_vector, multiplier in iter\_product(unit\_vectors, 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

I was about to explain the expected behaviour and then use a jupyter notebook to debug what is actually happening but in documenting it the source of the issue has become obvious to me.

I have used iter\_product to simulate 2 nested for loops

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

I actually need to use 2 distinct for loops as I only intended to break out of the multiple explorer for loop.

Currently the program starts with vector (-1, -1) and then test multiples 1, 2, and 3. As 3 is illegal the whole loop is broken. Let me just fix this.

Now with this fix:

        for unit\_vector in unit\_vectors:

            for multiplier in range(1,8):

There are still some issues.

The current output is (in order of examination)

['B2', 'A1', 'B4', 'A5', 'D2', 'E1', 'D4', 'E5', 'F6', 'G7', 'H8']

With the aid of a diagram of a board:



I have determined that this output means that the vectors that were examined were:

(-1, -1), (-1, 1), (1, -1), (1, 1)

This has prompted me to check that the vector generator is acting appropriately. I can now see the issue. I should use or, not and.

Old code:

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

new code

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

Writing these tests has also helped me detect other silly mistakes that are more obvious as to how to correct.

For the king piece which doesn’t explore many multiples of a vector, I changed the match case so that the loop wasn’t broken

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

        for movement\_vector in unit\_vectors:

            resultant\_vector = position\_vector + movement\_vector

            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

I then created a test for the black pawn including move foreward and take. I wanted to check that the vectors were being reflected correctly. The test was as follows

test:

  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

  ]

It failed. The code failed to recognise take moves C6 and E6. I then looked at the code:

    # override init constructor

    def \_\_init\_\_(self, color):

        self.color = color

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

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

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

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

            # v(0, 1) for foreword

            (Vector(0, 1\*multiplier), lambda square: self.square\_contains(square) == "empty"),

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

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

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

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

        )

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

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

            resultant\_vector = position\_vector + movement\_vector

            # if vector\_out of range continue

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

                continue

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

            piece = pieces\_matrix[row][column]

            if condition(piece):

                yield movement\_vector

I noticed these lines:

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

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

The black vs white pawns differ in the j component of the vector not the I so the multiplier is being applied at the wrong point.

Here was the correction

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

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

This caused the test to pass.

I started by refactoring the board state class so that it didn’t include the concept ownership. Methods such as static evaluation return an evaluation where white is the maximiser. The get pieces of owner of is player in check have been changed to be based on colour.

I will now try to see what else isn’t working by beginning to write unit tests.

The first methods to test will be methods that don’t depend on any other methods. For example legal moves depends on the in checkmate function and the static evaluation function depends on the game over function.

Let me start by testing these functions:

**Get\_piece\_at\_vector**

**Generate\_all\_pieces**

**Generate\_pieces\_of\_color**

**Color\_in\_check**

**Then**

**Generate\_legal\_moves**

**Game\_over**

**Then**

**Static\_evaluation**

**Make\_move**

So I have fixed an issue with the unit tests where only some were running. Now all the tests run. Looking under the hood. The class decorator works in tandem with the decorator file data to take my function for a single test with data as parameters and generate a procedure for each test in my yaml file.

I got partway though making a game over test before I realised it depended on the check test so I set it to skip. I then created a get piece at vector test which works. I used squares in the test data as this functionality is pretested and so assumed correct.

I have created a test for generate legal moves but is wasn’t working. To fix this I ported over the test into a self-contained notebook for easier debugging.

from board\_state import Board\_State

# tested so assumed correct

import pieces

from vector import Vector

# code repeated from test pieces, opportunity to reduce redundancy

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 deserialize\_pieces\_matrix(pieces\_matrix) -> 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='W')

    return board\_state

def test\_generate\_all\_pieces(pieces\_matrix, 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\_legal\_moves())

    # convert square to vector, allowed as this is tested

    def deserialize(data\_unit):

        # unpack test data unit

        piece, square = data\_unit

        return (descriptor\_to\_piece(piece), Vector.position\_vector\_from\_chess\_square(square))

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

    assert legal\_moves\_actual == legal\_moves\_expected, f"\nactual {legal\_moves\_actual}  !=  expected {legal\_moves\_expected}"

test\_data = {

    "pieces\_matrix": [

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        [None, None, None, None, None, None, None, None],

        ["BP",   None, "BP",   None, None, None, None, None],

        [None, "WP",   None, None, None, None, None, None]

    ],

    "pieces\_and\_squares": [

        ["WP", "B1"],

        ["BP", "A2"],

        ["BP", "C2"]

    ]

}

Text

Description automatically generated

This highlighted the issue. I could not use debugging to pier inside the test function and to see at which stage the logical behaviour deviated from expected.

Through adding some print statements

    def generate\_all\_pieces(self):

        for i, j in zip(range(8), range(8)):

            position\_vector = Vector(i,j)

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

            # skip if none

            if piece:

                dev\_print(f"not skipping {position\_vector}")

                yield piece, position\_vector

            else:

                dev\_print(f"skipping {position\_vector}")

I could see the behaviour

legal\_moves\_actual: set[pieces.Piece, Vector] = set(board\_state.generate\_legal\_moves())

legal\_moves\_actual

skipping Vector(i=0, j=0)

skipping Vector(i=1, j=1)

skipping Vector(i=2, j=2)

skipping Vector(i=3, j=3)

skipping Vector(i=4, j=4)

skipping Vector(i=5, j=5)

skipping Vector(i=6, j=6)

skipping Vector(i=7, j=7)

set()

This lead me to see this logic error: this wasn’t present in v2.3 and was reintroduced by refactoring in v2.4

        # for i, j in zip(range(8), range(8)):

        # should be product

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

I then encountered this issue

Text

Description automatically generated

This implied that there was some issue with my equality operator. Those sets are equal.

I then tried to use sorted but then realised that this means I needed either a key or a < operator. I will create an arbitrary key.

This lead me to this issue

Graphical user interface, text

Description automatically generated

(replicated directly from an issue in a test)

I changed the equality operator in the Pieces class to be \_\_eq\_\_ not the incorrect \_\_equal\_\_. I also added an equality operator to the vector class

    def \_\_eq\_\_(self, other):

        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

this means that the tests now work

Using sets was also a good flawed idea as it would mean that I would need to make my objects hash able, comparing sorted lists also accounts for duplicates.

Text

Description automatically generated

I now need to fix a test which is designed to check that the game\_over function can recognise checkmate.

This is the data for the failing test:

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

The black king should be in check for all squares it can move to. Therefore it is stalemate as the king cannot move and isn’t already in check. This concours as white can do nothing to get checkmate either.

    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

now due to earlier testing I will assume that my check and legal move function are fully working. The expected behaviour is (True, None) the actual behaviour is (false, None). This implies that the initial if statement that checks if the legal moves list is empty is not having its condition met resulting in the final line being executed. Therefore this **would** imply that is this case the legal moves generator is incorrectly presenting one or more legal moves. To be sure that this isn’t the case, I will add this board state to the legal move tests and check that there are no legal moves.

This result was due to a flaw in the execution of my testing strategy. I had become confused and had not in fact already tested legal moves generator. As such I will set this game over test to be skipped until the legal moves generator test can be made.

So I created a test for legal moves and started with the previous stalemate scenario.

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: []

Text

Description automatically generated

The test revealed that the legal moves generator thought that the black king could move left or right even though this would mean check from the white pawn.

So now I will add a new test to the check function to see why it doesn’t recognise this as check

test5:

  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

test5:

  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: true

  black\_in\_check: false

These tests both worked confirming that the issue was with the legal moves generator

Text

Description automatically generated

I have created a jupyter notebook to allow me to more closely analyse what exactly is going on here.

I modified the legal moves generator function to add a large amount of print statements. This is not ideal and I will eventually add logging instead

    def generate\_legal\_moves(self):

        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

                    ))

                )

                is\_check\_for\_a = child\_game\_state.color\_in\_check()

                dev\_print(f"\t\tThis game state in check?:  {is\_check\_for\_a}")

                if not is\_check\_for\_a:

                    yield piece\_position\_vector, movement\_vector

then when I manually complete the same test in jupyter notebook this is the output

analysing moves for piece: BK

    piece BK:\_analysing movement vector Vector(i=-1, j=-1)

        movement vector Vector(i=-1, j=-1): results in child game state

        [None, None, None, None, None, None, None, None]

        [None, None, 'BK', 'WP', None, None, None, None]

        [None, None, None, 'WK', None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        This game state in check?:  True

    piece BK:\_analysing movement vector Vector(i=-1, j=0)

        movement vector Vector(i=-1, j=0): results in child game state

        [None, None, 'BK', None, None, None, None, None]

        [None, None, None, 'WP', None, None, None, None]

        [None, None, None, 'WK', None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        This game state in check?:  False

    piece BK:\_analysing movement vector Vector(i=0, j=-1)

        movement vector Vector(i=0, j=-1): results in child game state

        [None, None, None, None, None, None, None, None]

        [None, None, None, 'BK', None, None, None, None]

        [None, None, None, 'WK', None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        This game state in check?:  True

    piece BK:\_analysing movement vector Vector(i=1, j=-1)

        movement vector Vector(i=1, j=-1): results in child game state

        [None, None, None, None, None, None, None, None]

        [None, None, None, 'WP', 'BK', None, None, None]

        [None, None, None, 'WK', None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        This game state in check?:  True

    piece BK:\_analysing movement vector Vector(i=1, j=0)

        movement vector Vector(i=1, j=0): results in child game state

        [None, None, None, None, 'BK', None, None, None]

        [None, None, None, 'WP', None, None, None, None]

        [None, None, None, 'WK', None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        This game state in check?:  False

This is unexpected and highlights 2 distinct issue to be investigated and fixed.

* One issue is that the king is being allowed to move in movement vectors outside the board. This is only being stopped by the checkmate function. This is not the intended way to prevent the king moving in this way as it is highly tenuous and unreliable. I will fix this be added new tests to the pieces module.
* The second issue is that this output suggests that the king moving left and right results in child game states that are not check but this is inconsistent with earlier testing of the check function with the same input. The current hypothesis is that the make move function is not updating the next to go variable in the correct way. The solution to this is to create a test for the make a move function.

Let me tackle the first issue of the king having incorrect movement vectors first.

I have added a test for this to the test empty board yaml file

test7:

  piece\_type: K

  square: D8

  expected\_move\_squares: [

    C7, D7, E7,

  C8, E8

  ]

This test worked, Now let me add the exact scenario to the test populated tests.

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

  ]

This test also passed. This is unexpected. Let me look again at what happened in that jupyter notebook. Well I didn’t look carefully enough. None of the movement vectors result in the king moving off the board. This is therefore not an issue. Oh well, better to have made the tests to be sure.

In the current test I am analysing the legal moves for black. I realised that calling make move to produce child game states would result in the child game states having a next to go property of white. This meant that my is check function call should have specified: is check for black. I corrected this below:

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

It still isn’t working and produces the same result where it incorrectly assert that black isn’t in check. I will add all of these game states to the is check test data.

At this point the is check function has been fully verified and it is working for all of these game states. Here are all the tests that I created to verify this. They all passed.

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

With no other option I continued to be suspicious of the check function and so I added more print statements. They allowed me to further narrow down where the issue was occurring.

    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}")

                # 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} == {pieces\_mod.King(color='B')}   -->    {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 {repr(pieces\_mod.King(color=color\_a))} 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

The resulting output was

analysing legal moves for next to go B

analysing moves for piece: BK

    piece BK:\_analysing movement vector Vector(i=-1, j=-1)

        movement vector Vector(i=-1, j=-1): results in child game state

        [None, None, None, None, None, None, None, None]

        [None, None, 'BK', 'WP', None, None, None, None]

        [None, None, None, 'WK', None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

        [None, None, None, None, None, None, None, None]

Board\_State(pieces\_matrix=((None, None, None, None, None, None, None, None), (None, None, King(color='B'), Pawn(color='W'), None, None, None, None), (None, None, None, King(color='W'), None, None, None, None), (None, None, None, None, None, None, None, None), (None, None, None, None, None, None, None, None), (None, None, None, None, None, None, None, None), (None, None, None, None, None, None, None, None), (None, None, None, None, None, None, None, None)), next\_to\_go='W')

to\_square   -->   None

piece at square C5 is <empty>

to\_square == pieces\_mod.King(color='B')   -->   None == BK   -->    False

therefore king NOT threatened as square DOES NOT contain King(color='B') instead containing <empty>

piece WK at D6 moving to C5 NOT threatening king

to\_square   -->   None

piece at square C6 is <empty>

to\_square == pieces\_mod.King(color='B')   -->   None == BK   -->    False

therefore king NOT threatened as square DOES NOT contain King(color='B') instead containing <empty>

piece WK at D6 moving to C6 NOT threatening king

to\_square   -->   BK

piece at square C7 is King(color='B')

to\_square == pieces\_mod.King(color='B')   -->   BK == BK   -->    False

therefore king NOT threatened as square DOES NOT contain King(color='B') instead containing King(color='B')

piece WK at D6 moving to C7 NOT threatening king

to\_square   -->   None

piece at square D5 is <empty>

to\_square == pieces\_mod.King(color='B')   -->   None == BK   -->    False

therefore king NOT threatened as square DOES NOT contain King(color='B') instead containing <empty>

piece WK at D6 moving to D5 NOT threatening king

to\_square   -->   None

piece at square E5 is <empty>

to\_square == pieces\_mod.King(color='B')   -->   None == BK   -->    False

therefore king NOT threatened as square DOES NOT contain King(color='B') instead containing <empty>

piece WK at D6 moving to E5 NOT threatening king

to\_square   -->   None

piece at square E6 is <empty>

to\_square == pieces\_mod.King(color='B')   -->   None == BK   -->    False

therefore king NOT threatened as square DOES NOT contain King(color='B') instead containing <empty>

piece WK at D6 moving to E6 NOT threatening king

to\_square   -->   None

piece at square E7 is <empty>

to\_square == pieces\_mod.King(color='B')   -->   None == BK   -->    False

therefore king NOT threatened as square DOES NOT contain King(color='B') instead containing <empty>

piece WK at D6 moving to E7 NOT threatening king

to\_square   -->   None

piece at square D8 is <empty>

to\_square == pieces\_mod.King(color='B')   -->   None == BK   -->    False

therefore king NOT threatened as square DOES NOT contain King(color='B') instead containing <empty>

piece WP at D7 moving to D8 NOT threatening king

color\_in\_check(color='B') returning False

        This game state in check? for B:  False

This is the output from only one movement vector but highlights an issue.

In this case the king moved to C7, the check function examined the movement vector of the white pawn to C7 and determined incorrectly that the king wasn’t threatened

to\_square   -->   BK

piece at square C7 is King(color='B')

to\_square == pieces\_mod.King(color='B')   -->   BK == BK   -->    False

therefore king NOT threatened as square DOES NOT contain King(color='B') instead containing King(color='B')

piece WK at D6 moving to C7 NOT threatening king

to\_square   -->   None

To ensure that none of the variables contained string desrciptions of pieces, I used the repr function to make clear that the equality operator was failing

to\_square - -> King(color='B')

piece at square C7 is King(color='B')

to\_square == pieces\_mod.King(color='B') - -> King(color='B') == King(color='B') - -> False

therefore king NOT threatened as square DOES NOT contain King(color='B') instead containing King(color='B')

piece WK at D6 moving to C7 NOT threatening king

I then tried to use jupyter notebook to check that the equality operator was working between 2 kings and it was?

Text

Description automatically generated

I then looked at the equality operator I had written

    def \_\_eq\_\_(self, other):

        try:

            # assert same subclass like rook

            assert isinstance(other, type(self))

            assert self.color == other.color

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

            # value and value\_matrix should never be changes

        except AssertionError:

            return False

        else:

            return True

    def \_\_hash\_\_(self):

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

I immediately realised that the obvious difference was that the one of the kings would have a value for last move. I then removed the last moved part of the operator

    def \_\_eq\_\_(self, other):

        try:

            # assert same subclass like rook

            assert isinstance(other, type(self))

            assert self.color == other.color

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

            # value and value\_matrix should never be changes

        except AssertionError:

            return False

        else:

            return True

I will now try the code.

It works. Thank Chris Evans, it works.

So the issue was that, in the is check code, the equality operator was failing as the king had just moved. This issue wasn’t occurring in my as all of the piece objects tested had a last move value of None. Now I can finish the other unittests.

I then completed some other tests, including the starting positions:

test4:

  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\_legal\_moves: [

    [A2, A3], [A2, A4],

    [B2, B3], [B2, B4],

    [C2, C3], [C2, C4],

    [D2, D3], [D2, D4],

    [E2, E3], [E2, E4],

    [F2, F3], [F2, F4],

    [G2, G3], [G2, G4],

    [H2, H3], [H2, H4],

    [B1, A3], [B1, C3],

    [G1, F3], [G1, H3],

  ]

This test failed as the actual legal move function failed to recognise that the knights could move forward over the pawns. I will now double down on testing the knights in the pieces unittests.

I then realised that the issue was that I had used WK to be knight in this case even though it is meant to be WN. I will correct this. Once this was corrected, all the tests worked.