**Faculty of Organization and Informatics Varaždin**

**Faculty of Organizational Sciences Beograd**

**University of Žilina**

**An application for playing a social game over the network - Object-oriented databases - ZODB**

Professors: Students:

Markus Schatten Goran Filinić

Michael Kvet Martin Gajdoš

Srđa Lj. Bjeladinović Jakov Gatarić

Milica S. Škembarević Ilija Radojković

Luka Zrinski

Contents

[1. Introduction 1](#_Toc125575224)

[2. About the game 1](#_Toc125575225)

[3. Technologies 5](#_Toc125575226)

[3.1 Python 6](#_Toc125575227)

[3.2 Pygame 7](#_Toc125575228)

[4. Use cases 8](#_Toc125575229)

[5. Backend 8](#_Toc125575230)

[5.1 Moving pieces 9](#_Toc125575231)

[5.2 Attacking pieces 9](#_Toc125575232)

[5.3 Special moves 10](#_Toc125575233)

[5.4 Checkmate 10](#_Toc125575234)

[15. Design 11](#_Toc125575235)

[16. UML diagram 13](#_Toc125575236)

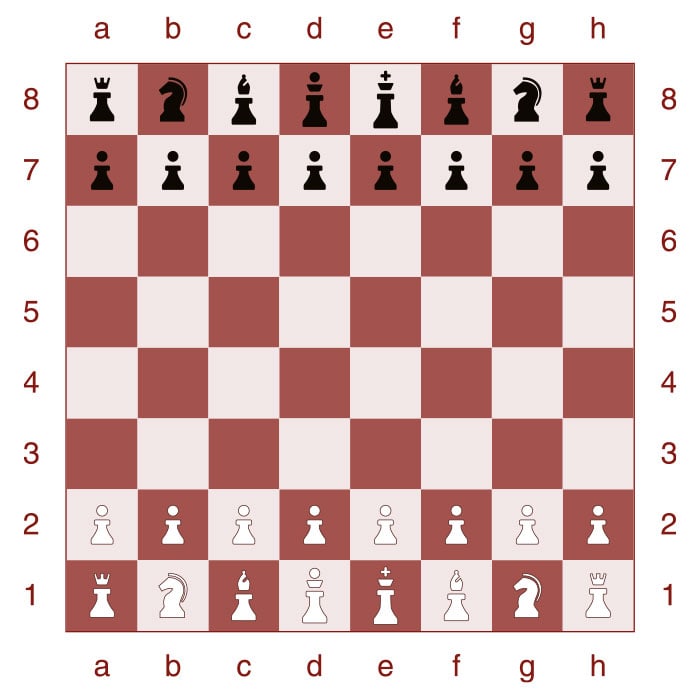
# Introduction

This document specifies a design for the gameplay with title “Chess”.Chess is a board game between two players. Each player chooses between two colored chess pieces. Chess pieces are divided into two different colored sets. While the sets might not be literally white and black. They are always referred to as “white” and “black”. Each set consists of sixteen pieces: one king, one queen, two rooks, two bishops, two knights, and eight pawns. The game is played on a square board of eight rows (called ranks) and eight columns (called files).

In this project we used Python as our main programing language, and his library Pygame that helped us build basic game logic. We integrated that with ZODB that we have persisted data between two player, and have a history of player moves.

# About the game

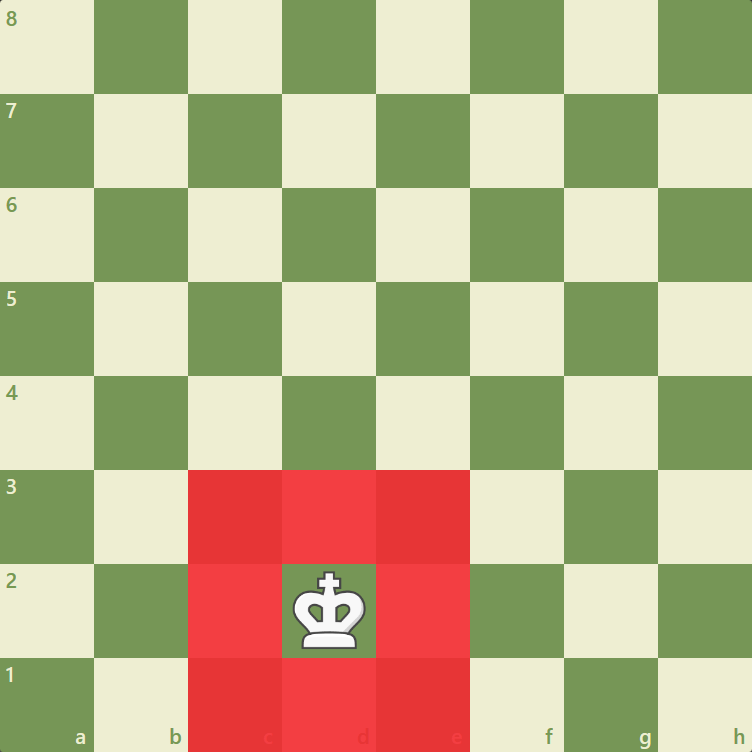
Each player chooses between two colored chess pieces. Chess pieces are divided into two different colored sets. While the sets might not be literally white and black. They are always referred to as “white” and “black”. Each set consists of sixteen pieces: one king, one queen, two rooks, two bishops, two knights, and eight pawns. The game is played on a square board of eight rows (called ranks) and eight columns (called files).



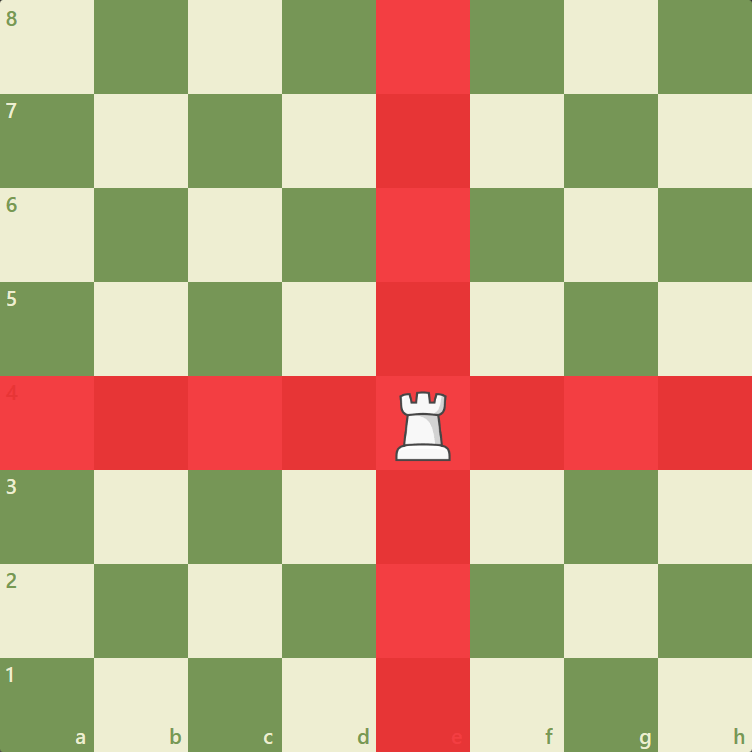
The pieces are set out as shown in the  photo. Thus, on White's first rank, from left to right, the pieces are placed in the following order: rook, knight, bishop, queen, king, bishop, knight, rook. On the second rank is placed a row of eight pawns. Black's position mirrors White's, with an equivalent piece on the same file. The board is placed with a light square at the right-hand corner nearest to each player. The correct positions of the king and queen may be remembered by the phrase "queen on her own color" – the white queen begins on a light square, and the black queen on a dark square.

White moves first, after which players alternate turns, moving one piece per turn, except for castling, when two pieces are moved. A piece is moved to either an unoccupied square or one occupied by an opponent's piece. Each piece has its own way of moving. In the diagrams, the dots mark the squares to which the piece can move if there are no intervening piece(s) of either color (except the knight, which leaps over any intervening pieces). All pieces except the pawn can capture an enemy piece if it is located on a square to which they would be able to move if the square was unoccupied. The squares on which pawns can capture enemy pieces are marked in the diagram with black crosses.

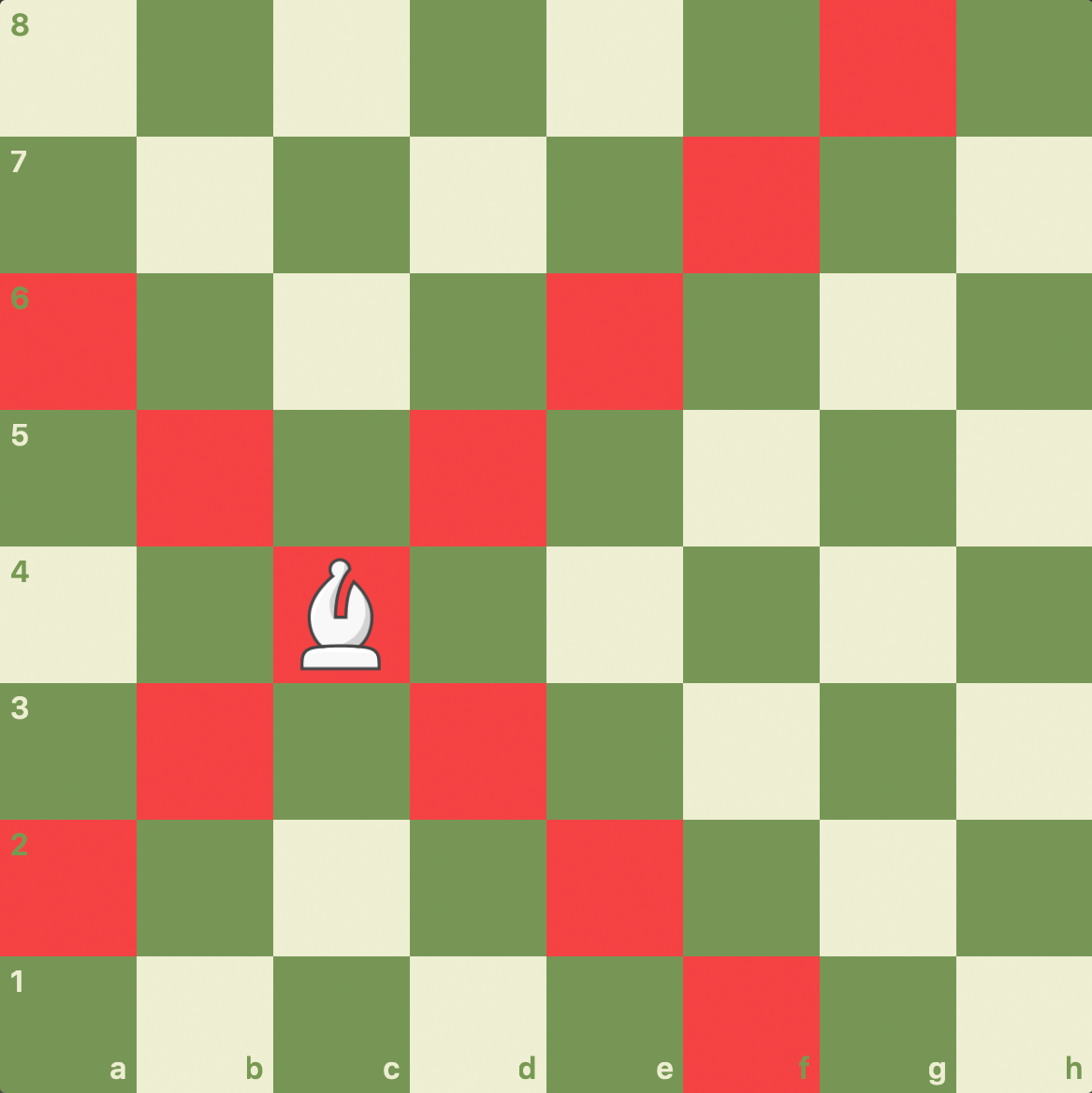
* The **king** moves one square in any direction. There is also a special move called castling that involves moving the king and a rook. The king is the most valuable piece — attacks on the king must be immediately countered, and if this is impossible, immediate loss of the game ensues.



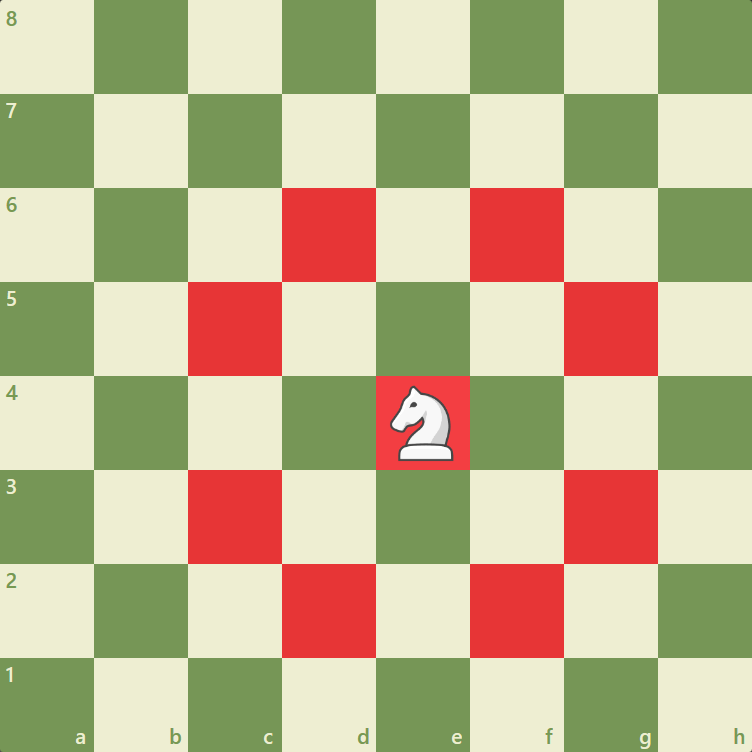
* A **rook** can move any number of squares along a rank or file, but cannot leap over other pieces. Along with the king, a rook is involved during the king's castling move.



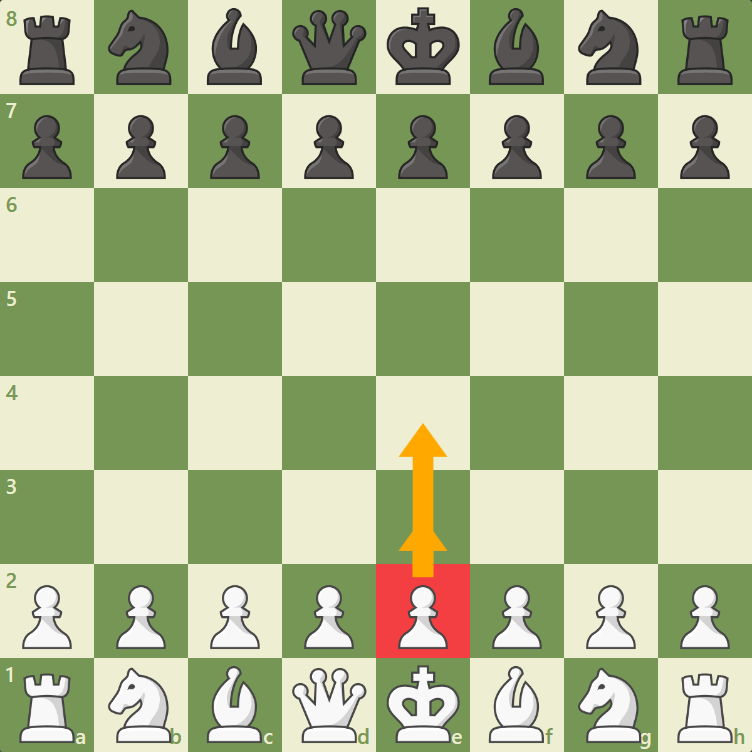
* A **bishop** can move any number of squares diagonally, but cannot leap over other pieces.



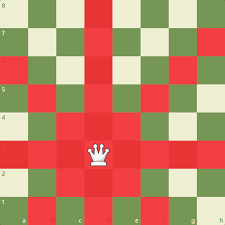
* A **knight** moves to any of the closest squares that are not on the same rank, file, or diagonal. (Thus the move forms an "L"-shape: two squares vertically and one square horizontally, or two squares horizontally and one square vertically.) The knight is the only piece that can leap over other pieces.



* A **pawn** can move forward to the unoccupied square immediately in front of it on the same file, or on its first move it can advance two squares along the same file, provided both squares are unoccupied (black dots in the diagram). A pawn can capture an opponent's piece on a square diagonally in front of it by moving to that square (black crosses). It can*not* capture a piece while advancing along the same file. A pawn has two special moves: the *[en passant](https://en.wikipedia.org/wiki/Chess" \l "En_passant)* capture and [promotion](https://en.wikipedia.org/wiki/Chess#Promotion).



* A **queen** combines the power of a rook and bishop and can move any number of squares along a rank, file, or diagonal, but cannot leap over other pieces.



When a king is under immediate attack, it is said to be in **check**. A move in response to a check is legal only if it results in a position where the king is no longer in check. This can involve capturing the checking piece; interposing a piece between the checking piece and the king (which is possible only if the attacking piece is a queen, rook, or bishop and there is a square between it and the king); or moving the king to a square where it is not under attack. Castling is not a permissible response to a check.[[1]](https://en.wikipedia.org/wiki/Chess#cite_note-FideLawsOfChess-1)

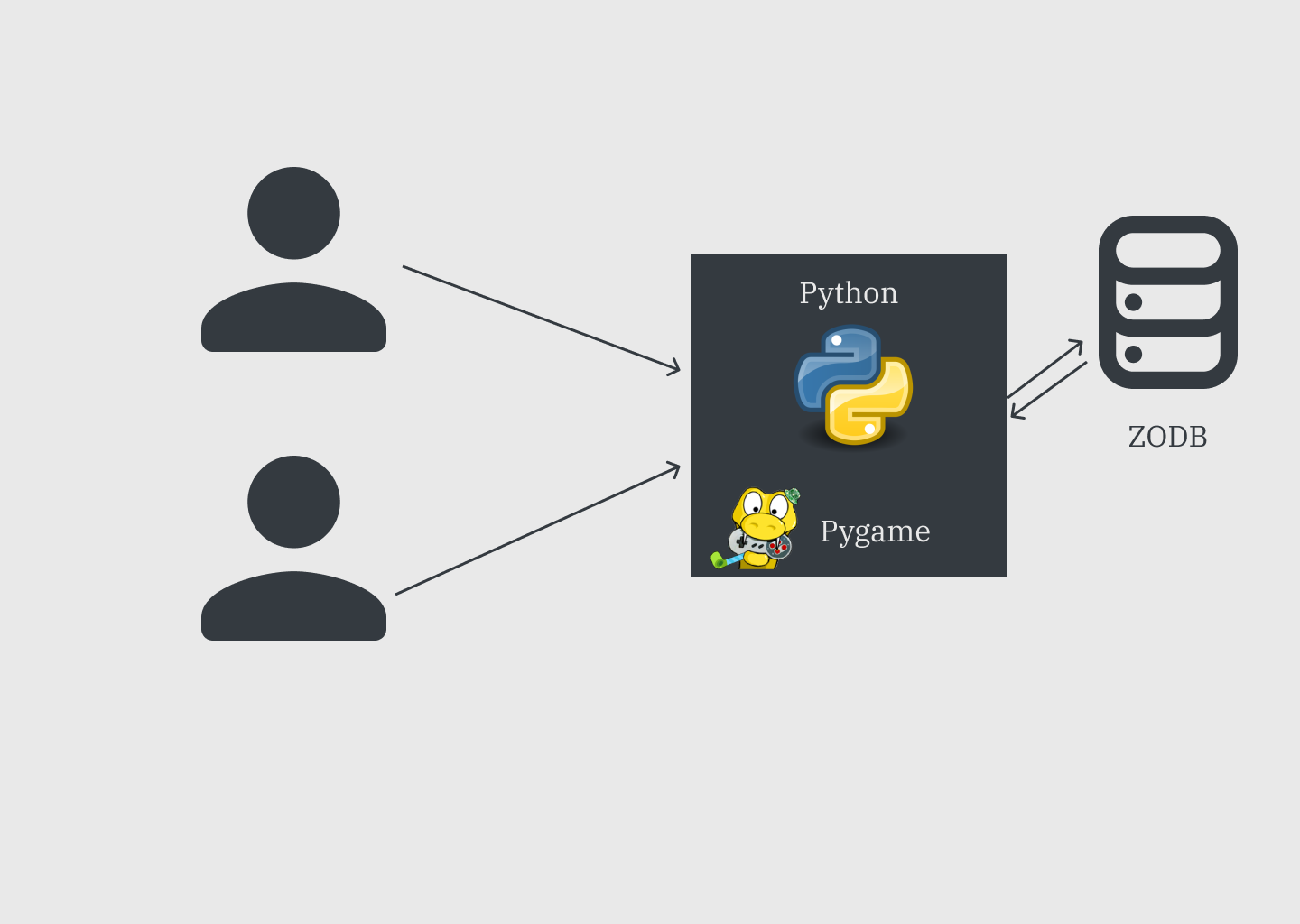
The object of the game is to **checkmate** the opponent; this occurs when the opponent's king is in check, and there is no legal way to get it out of check.

# Technologies

Our project is done in Python programming language using ZODB for storage. We created fully functional chess game using Python like backend with specific game libraries that helped us simplify the game logic and manipulating objects. Our final touch was adding our objects in ZODB. We persisted game moves played by players, and their final result of the game, by that we have complete history of players match.

Main technologies:

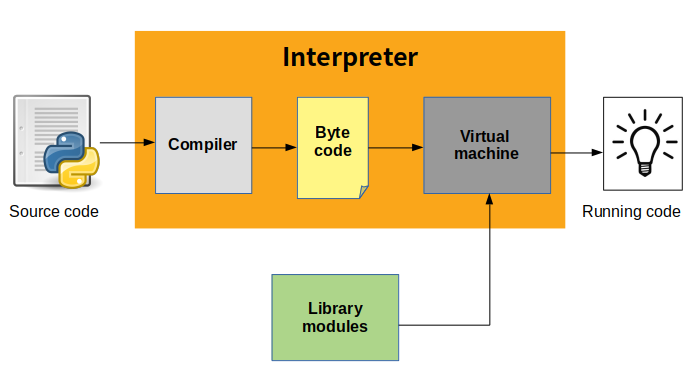
* Python
* ZODB(Zope Object Database)



## 3.1 Python

Python is a [high-level](https://en.wikipedia.org/wiki/High-level_programming_language), [general-purpose programming language](https://en.wikipedia.org/wiki/General-purpose_programming_language). Its design philosophy emphasizes [code readability](https://en.wikipedia.org/wiki/Code_readability) with the use of [significant indentation](https://en.wikipedia.org/wiki/Off-side_rule).

Python is [dynamically typed](https://en.wikipedia.org/wiki/Type_system#DYNAMIC) and [garbage-collected](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)). It supports multiple [programming paradigms](https://en.wikipedia.org/wiki/Programming_paradigm), including [structured](https://en.wikipedia.org/wiki/Structured_programming) (particularly [procedural](https://en.wikipedia.org/wiki/Procedural_programming)), [object-oriented](https://en.wikipedia.org/wiki/Object-oriented_programming) and [functional programming](https://en.wikipedia.org/wiki/Functional_programming). It is often described as a "batteries included" language due to its comprehensive [standard library](https://en.wikipedia.org/wiki/Standard_library)

Python is meant to be an easily readable language. Its formatting is visually uncluttered and often uses English keywords where other languages use punctuation. Unlike many other languages, it does not use [curly brackets](https://en.wikipedia.org/wiki/Curly_bracket_programming_language) to delimit blocks, and semicolons after statements are allowed but rarely used. It has fewer syntactic exceptions. Python uses [whitespace](https://en.wikipedia.org/wiki/Whitespace_character) indentation, rather than [curly brackets](https://en.wikipedia.org/wiki/Curly_bracket_programming_language) or keywords, to delimit [blocks](https://en.wikipedia.org/wiki/Block_(programming)). An increase in indentation comes after certain statements; a decrease in indentation signifies the end of the current block. Thus, the program's visual structure accurately represents its semantic structure. This feature is sometimes termed the [off-side rule](https://en.wikipedia.org/wiki/Off-side_rule). Some other languages use indentation this way, but in most, indentation has no semantic meaning. The recommended indent size is four spaces. 

Python is both compiled and interpreted language. Python language code is actually compiled into bytecodes and then executed in Python virtual machine by the interpreter.

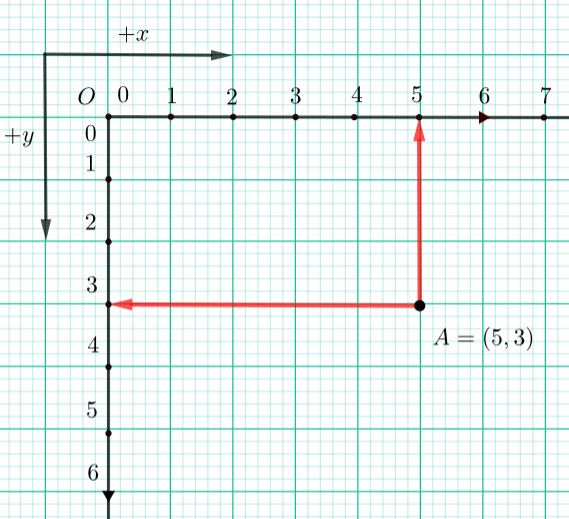
## 3.2 Pygame

Pygame is a cross-platform set of Python modules designed for writing video games.It includes computer graphics and sound libraries designed to be used with Python programing language. The Pygame library has been developing since the early 2000s. The authors themselves said that this is not the best library for game programming (not even the second, nor the third by quality), but its main advantage is that it is simpler to use than the other libraries and it is suitable for learning programming through the interesting world of computer graphics and computer game.

In order to run programs written using PyGame in your application development environment, you must first install this library. When Python is installed on your computer, we can go to the PyGame library installation. It’s really very simple. Just type pip3 install pygame in the command prompt. In order for the programs we write to use the PyGame library (module), the first thing we need to do is import the PyGame module at the beginning of the program. This allows us to use all the functions and constants defined in the PyGame module.

Coordinates are a very important term for us and we will encounter them in just about every PyGame program. The position of all objects (points, line segments, circles, text, imported images, etc.) in the window is determined by their coordinates in the coordinate system of the window.The coordinate system of a window is similar, but still slightly different from the one used in mathematics. The position of a point is determined by an ordered pair of its coordinates in this case as well (coordinate x, i.e. abscissa and coordinate y, i.e. ordinate). The unit of measurement is one pixel.

In computer graphics, origin of a coordinate system is in the upper left corner of a window.



# Running the application

The first step is to run the ZEO server with the command**: runzeo -a localhost:2709 -f db.fs** , the next step is to start the application server so that a communication would be enabled between the database and the client application. We are starting the server with the command: **python server.py** . The last step is to run the app with the command**: python main.py**.

# Backend

In our project we used Python and his game making library Pygame. Our complete backend and game logic is built in Python, using Pygame that enables us to easily build game engine for chess game. We start our game with *main()* function that is being called first.

def main():

    screen=setUpScreen()

    clock=p.time.Clock()

    loadModels()

    active=True

    while active:

        checkEventsAndUpdatetheBoard(active,screen,clock)

First of all we set up our screen with method *setUpScreen()* , the result of this is that we can see our board loaded. Code shown bellow is our *setUpScreen()* function

def setUpScreen():

    screen=p.display.set\_mode((BOARD\_WIDTH + MOVE\_LOG\_PANEL\_WIDTH,BOARD\_HEIGHT))

    screen.fill(p.Color("white"))

    return screen

Then we define clock with Pygame library. Each player will have clock attach to it, so each player have limited time to play their moves. We get this object from Pygame library with:

p.time.Clock()

where p is reference to Pygame library.

After that we can load our models. Our models represent chess pieces, with black and white color. For this we use array of String objects where each string has star letter of “W” or “B”, where W is white, and B is black.

 def loadModels():

    chessPieces=['wP','wR','wN','wB','wK','wQ','bP','bR','bN','bB','bK','bQ']

    for piece in chessPieces:

        Models[piece]=p.transform.scale(p.image.load("BackEnd\Models\\"+piece+".png"),(SqSize,SqSize))

We assign to each model its image, and we scale it to appropriate dimensions.

As the final line of code we run the game while its active.

### 5.1 Moving pieces

Moving pieces is our core game logic. Each chess piece have limited movement, for each piece we have defined a function that restricts wrong movement. For pawns we defined *getpawnMoves(self,row,column,moves)* ,*getRookMoves(self,wor,column,moves)* for rook,*getKnightMoves(self,row,column,moves)* for *knight,getBishopMoves(self,row,column,moves)* for bishop,*getQueenMoves(self,row,column,moves)* for queen,and *getKingMoves(self,row,column,moves)* for king.

We have special moves like castle moves defined in our *getKingssideCastleMoves(self,row,column,moves)* and *getQueensideCastleMoves(self,row,column,moves).*We defined list of lists which is our board variable:

    self.board=[

            ['bR','bN','bB','bQ','bK','bB','bN','bR'],

            ['bP','bP','bP','bP','bP','bP','bP','bP'],

            ['\_\_','\_\_','\_\_','\_\_','\_\_','\_\_','\_\_','\_\_'],

            ['\_\_','\_\_','\_\_','\_\_','\_\_','\_\_','\_\_','\_\_'],

            ['\_\_','\_\_','\_\_','\_\_','\_\_','\_\_','\_\_','\_\_'],

            ['\_\_','\_\_','\_\_','\_\_','\_\_','\_\_','\_\_','\_\_'],

            ['wP','wP','wP','wP','wP','wP','wP','wP'],

            ['wR','wN','wB','wQ','wK','wB','wN','wR'],

        ]

When we move our pieces we update our board, so we have the current game state available.

### 5.2 Attacking pieces

We have defined function that indicate us if square is under attack:

def squareUnderAttack(self, row, column):

        self.whiteTurn = not self.whiteTurn

        opponentMoves = self.getAllPossibleMoves()

        self.whiteTurn = not self.whiteTurn

        for move in opponentMoves:

            if move.endRow == row and move.endColumn == column:

                return True

        return False

### 5.3 Special moves

Special moves in chess game are castle moves. There are 2 special castle moves, queens castle move and king castle move. Each of this move is defined in special function bellow. First of all we need to check if our king is under attack if we want to perform castle move, we permitted special move when king is under attack.

def getCastleMoves(self, row, column, moves):

        if self.inCheck:

            return

        if (self.whiteTurn and self.currentCastlingRights.wKside) or (not self.whiteTurn and self.currentCastlingRights.bKside):

            self.getKingsideCastleMoves(row, column, moves)

        if (self.whiteTurn and self.currentCastlingRights.wQside) or (not self.whiteTurn and self.currentCastlingRights.bQside):

            self.getQueensideCastleMoves(row, column, moves)

    def getKingsideCastleMoves(self, row, column, moves):

        if self.board[row][column + 1] == "\_\_" and self.board[row][column +2] == "\_\_":

            if not self.squareUnderAttack(row, column + 1) and not self.squareUnderAttack(row, column + 2):

                moves.append(Movement((row, column), (row, column + 2), self.board, isCastleMove = True))

    def getQueensideCastleMoves(self, row, column, moves):

        if self.board[row][column - 1] == "\_\_" and self.board[row][column - 2] == "\_\_" and self.board[row][column - 3] == "\_\_":

            if not self.squareUnderAttack(row, column - 1) and not self.squareUnderAttack(row, column - 2):

                moves.append(Movement((row, column), (row, column - 2), self.board, isCastleMove = True))



### 5.4 Checkmate

Purpose of this game is that one of player get other in checkmate position, where king has no space to move. We simply check for state of the board to be in checkmate position, and then if it is white turn black was in checkmate and lost the game, and the same applies for black.

1. def checkForGameOver(boardState, screen):
2. if boardState.checkMate:
3. if boardState.whiteTurn:
4. drawText(screen, "Black wins by checkmate")
5. return True
6. else:
7. drawText(screen, "White wins by checkmate")
8. elif boardState.staleMate:
9. drawText(screen, "Stalemate!")



## Design

As for the pieces we see when we play the game, we used images found on internet for each piece and manipulate them like objects using Python. We downloaded images and corrected them if needed in photoshop. Our images are stored in BackEnd/Models, each image has .png extension, and each piece we load with Pygame library, specifically with :

p.image.load(“path to image”)

where p is reference to Pygame library. We can use this build-in function to load images, and then manipulate them with our game logic.

As for our board and look of it we draw it manually with:

def drawSquares(screen):

    global colors

    colors=[p.Color("light gray"),p.Color("#999999")]

    for row in range(DIMENSIONS):

        for column in range(DIMENSIONS):

            color=colors[((row+column)%2)]

            p.draw.rect(screen,color,p.Rect(column\*SqSize,row\*SqSize,SqSize,SqSize))

def drawPieces(screen,board):

    for row in range(DIMENSIONS):

        for column in range(DIMENSIONS):

            piece=board[row][column]

            if piece != '\_\_':

                screen.blit(Models[piece],p.Rect(column\*SqSize,row\*SqSize,SqSize,SqSize))

def drawBoard(screen,boardState, validMoves):

    drawSquares(screen)

    highlightMoves(screen, boardState, validMoves)

    drawPieces(screen,boardState.board)

    drawMoveLog(screen,boardState)

And each piece when moved is animated with our function:

ef animeteMove(move, screen, boardState, clock):

    global colors

    coords = []

    deltaRow = move.endRow - move.startRow

    deltaColumn = move.endColumn - move.startColumn

    framesPerSquare = 5

    frameCount = (abs(deltaRow) + abs(deltaColumn)) \* framesPerSquare

    for frame in range(frameCount + 1):

        row, column = ((move.startRow + deltaRow \* frame/frameCount), move.startColumn + deltaColumn \* frame/frameCount)

        drawSquares(screen)

        drawPieces(screen, boardState)

        color=colors[(move.endRow + move.endColumn) % 2]

        endSquare = p.Rect(move.endColumn \* SqSize, move.endRow \* SqSize, SqSize, SqSize)

        p.draw.rect(screen, color, endSquare)

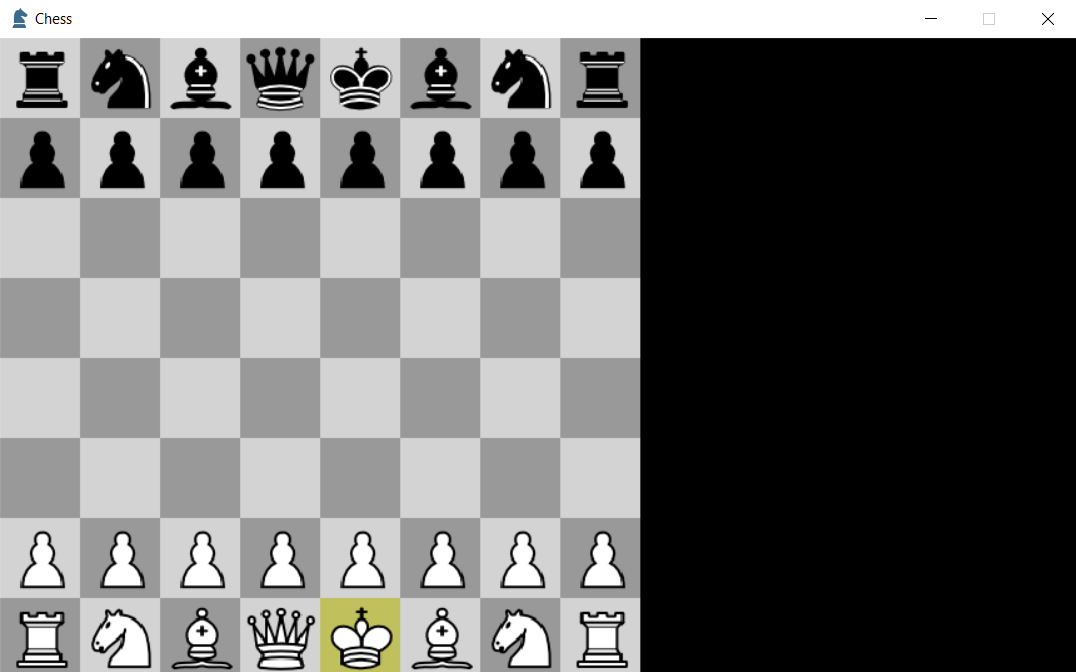
        if move.pieceMovedTo != "\_\_":

            screen.blit(Models[move.pieceMovedTo], endSquare)

        screen.blit(Models[move.pieceMovedFrom], p.Rect(column \* SqSize, row \* SqSize, SqSize, SqSize))

        p.display.flip()

        clock.tick(60)

****

## UML diagram

We connected our backend with storage database called ZODB (Zope Object Database). ZODB is object-oriented database that stores Python objects. ZODB stores Python objects using an extended version of Python's built-in object persistence (pickle). A ZODB database has a single root object (normally a dictionary), which is the only object directly made accessible by the database. All other objects stored in the database are reached through the root object. Objects referenced by an object stored in the database are automatically stored in the database as well.

