CHAPTER 1→ **ARTIFICIAL INTELLIGENCE**

INTRODUCTION :

Artificial intelligence (AI) is an area of computer science that emphasizes the creation of intelligent machines that work and react like humans. In general use, the term "artificial intelligence" means a machine which mimics human cognition. It has become an essential part of the technology industry. Some of the activities computers with artificial intelligence are designed for include:

- Speech recognition
- Learning
- Planning
- Problem solving

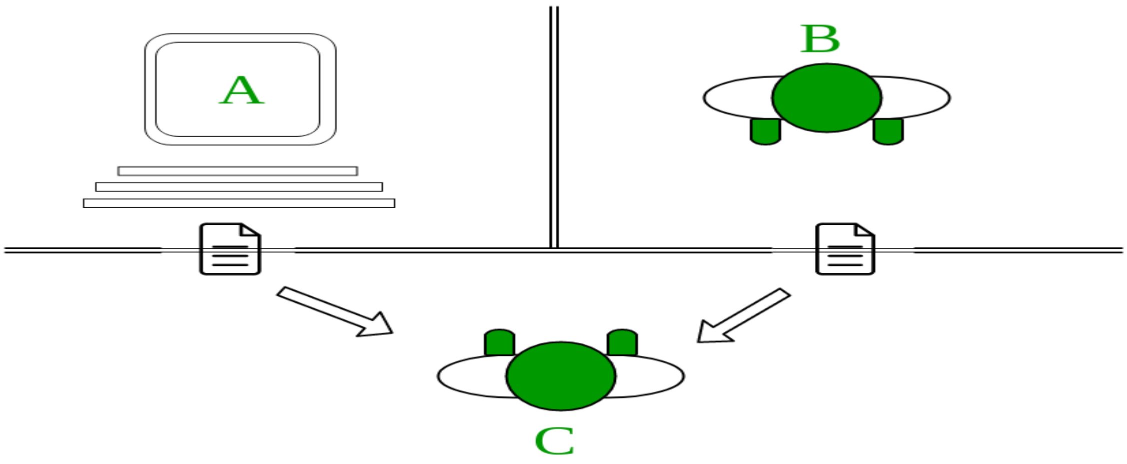
Artificial intelligence in video games

In video games, **artificial intelligence** is used to generate responsive, adaptive or intelligent behaviors primarily in non-player characters(NPCs), similar to human-like intelligence. The techniques used typically draw upon existing methods from the field of artificial intelligence(AI). When you make a game, you often have enemies for the player to combat. You want these enemies to seem intelligent and present a challenge to the player to keep the game fun and engaging.
You can do this through a variety of techniques collectively known as artificial intelligence (AI) programming. This is a huge field – entire books have been written on the subject! There are many subtopics, from pathfinding to steering to planning to state machines to behavior trees and more.
Many algorithms comes under Artificial Intelligence :-

- Depth-First
- Breadth-First
- Uniform Cost
- A*
- Minimax

Turing Test in Artificial Intelligence

The **Turing test** developed by Alan Turing(Computer scientist) in 1950. He proposed that "Turing test is used to determine whether or not computer(machine) can think intelligently like human"?
Imagine a game of three players having two humans and one computer, an interrogator(as human) is isolated from other two players. The interrogator job is to try and figure out which one is human and which one is computer by asking questions from both of them. To make the things harder computer is trying to make the interrogator guess wrongly. In other words computer would try to indistinguishable from human as much as possible.



Goals of Artificial Intelligence:

- To Create Expert Systems − The systems which exhibit intelligent behavior, learn, demonstrate, explain, and advice its users.
- To Implement Human Intelligence in Machines − Creating systems that understand, think, learn, and behave like humans. Goals of AI
- To Create Expert Systems − The systems which exhibit intelligent behavior, learn, demonstrate, explain, and advice its users.
- To Implement Human Intelligence in Machines − Creating systems that understand, think, learn, and behave like humans.

Applications of AI :

- Gaming
- Natural Language Processing
- Expert Systems,Speech Recognition
- Handwriting Recognition
- Intelligent Robots

CHAPTER 2 → **GAME DEVELOPMENT IN PYTHON**

PyGame Library

The pygame library is an open-source module for the Python programming language specifically intended to help you make games and other multimedia applications. Built on top of the highly portable SDL (Simple DirectMedia Layer) development library, pygame can run across many platforms and operating systems.

The fact that Pygame is open source means that bugs are generally fixed very quickly by the community. It also means that you can extend Pygame to suit your needs and maybe even give back to the community.

Pygame is highly portable, as it supports Windows, Linux, Mac OS X, BeOS, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX. In addition to these computing platforms, a Pygame subset for Android exists. Furthermore, Pygame does not require OpenGL and can use DirectX, WinDIB, X11, Linux framebuffer and many other APIs to render graphics. This ensures that a larger number of users can play your game

By using the pygame module, you can control the logic and graphics of your games without worrying about the backend complexities required for working with video and audio.

PYGAME INSTALLATION

- Pygame requires Python; if you don't already have it, you can download it from python.org. **Use python 3.6.1** or greater, because it is much friendlier to newbies, and additionally runs faster.
- The best way to install pygame is with the pip tool (which is what python uses to install packages). Note, this comes with python in recent versions. We use the --user flag to tell it to install into the home directory, rather than globally.

"python3 -m pip install pygame –user"

- To see if it works, run one of the included examples:

"python3 -m pygame.examples.aliens"

TURTLE GRAPHICS

- Turtle graphics is a method of programming "vector"graphics using a relative cursor upon a Cartesian plane
- Python has a built in module that supports turtle graphics called the "turtle" module.
- In turtle graphics you control a cursor, also know as a "turtle".

It has the following properties

- A position in 2D space
- An orientation, or heading
- A pen that can lay down color on the canvas

<u>Setting up your turtle environment</u>

```
import turtle!

#set a title for your canvas window!

turtle.title(”My Awesome Turtle Animation")!

# set up the screen size (in pixels - 425 x 425) !

# set the starting point of the turtle (0, 0)!

turtle.setup(425, 425, 0, 0)!
```

<u>IMPORANT NOTE!</u>

- ➢ Do not name your Python source code file “turtle.py”
- ➢ This will prevent Python from finding the correct “turtle.py”
- ➢ module when you use the “import turtle” statement at the top of your program

Getting rid of the turtle window

This function call will cause your turtle window to deactivate. When you click on it. Place it at the end of your program. “turtle.exitonclick()”

Basic Drawing

You can move your turtle forward by using the following command:

“turtle.forward(pixels)”

And you can have your turtle turn by using the following

commands:

“turtle.right(degrees)”

“turtle.left(degrees)”

Your turtle will continually “paint” while it’s moving.

CHAPTER 3 → TIC- TAC- TOE

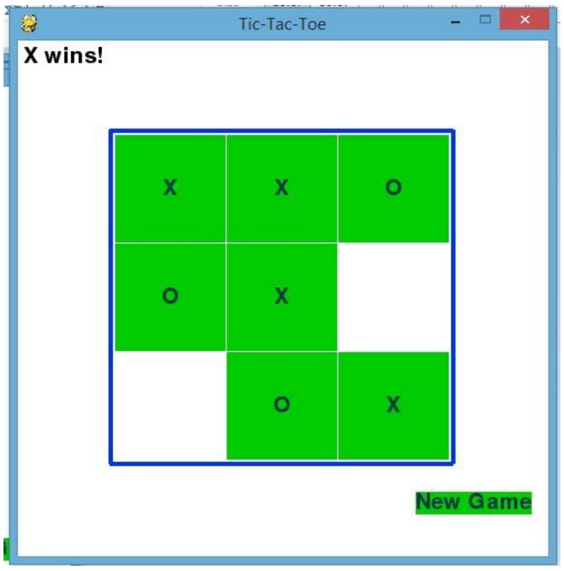
Implementation of Tic-Tac-Toe game

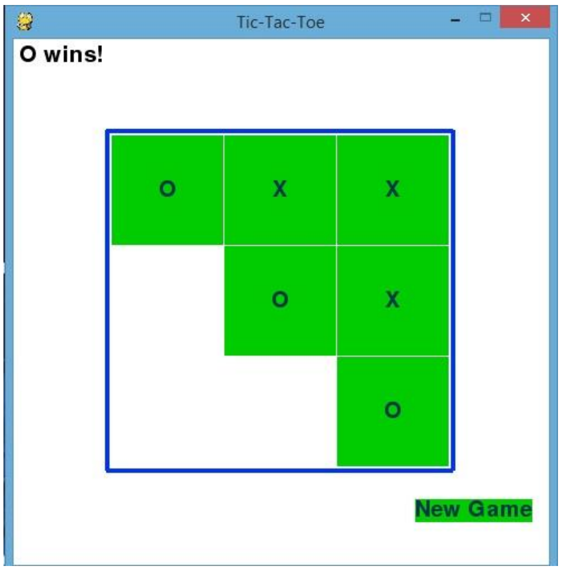
Rules of the Game

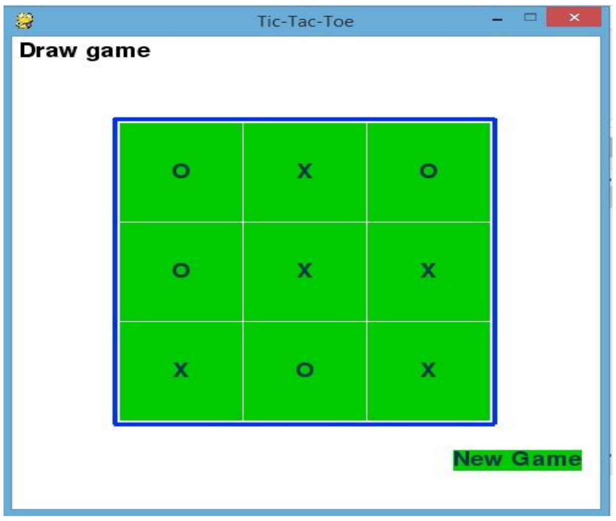
- The game is to be played between two people (in this program between HUMAN and COMPUTER).
- One of the player chooses ‘O’ and the other ‘X’ to mark their respective cells.
- The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character (‘O’ or ‘X’).
- If no one wins, then the game is said to be draw.

Implementation

In our program the computer moves are made in order to prevent user from winning.If X (The human player) wins then game displays “X wins”, if O (the computer) wins the game displayes “O wins”, else it displays “Draw Game”.







Algorithms:

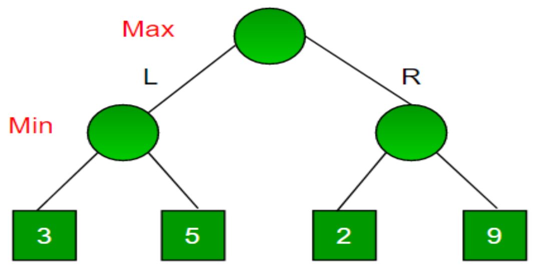
There are mainly two algorithm used in tic tac toe which are:

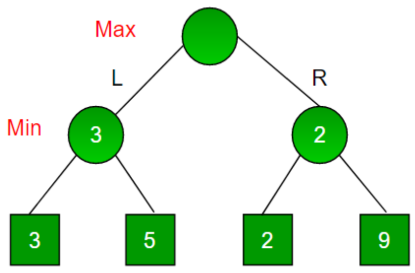
- Minmax algorithm.
- Alpha beta pruning.

1. **Minmax Algorithm-** Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn based games such as Tic-Tac-Toe, Backgamon, Mancala, Chess, etc.

   In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to get the lowest score possible while minimizer tries to do opposite.

   Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.



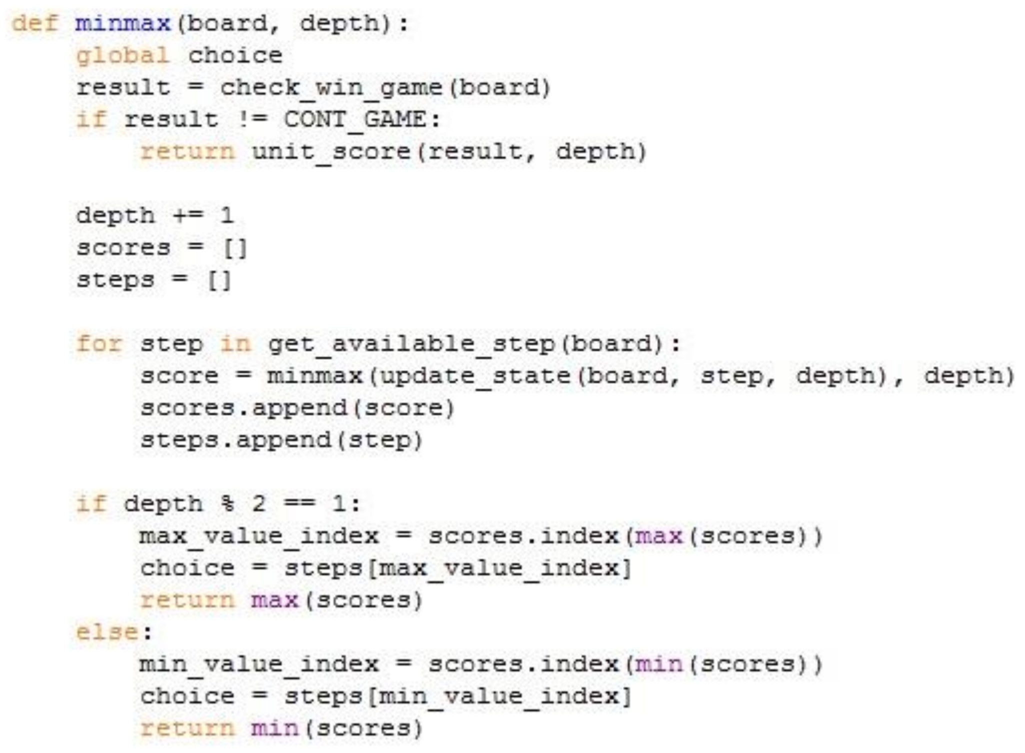


Since this is a backtracking based algorithm, it tries all possible moves, then backtracks and makes a decision.

- Maximizer goes LEFT : It is now the minimizers turn. The minimizer now has a choice between 3 and 5. Being the minimizer it will definitely choose the least among both, that is 3
- Maximizer goes RIGHT : It is now the minimizers turn. The minimizer now has a choice between 2 and 9. He will choose 2 as it is the least among the two values.

Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is 3.

Now the game tree looks like the tree in the right :



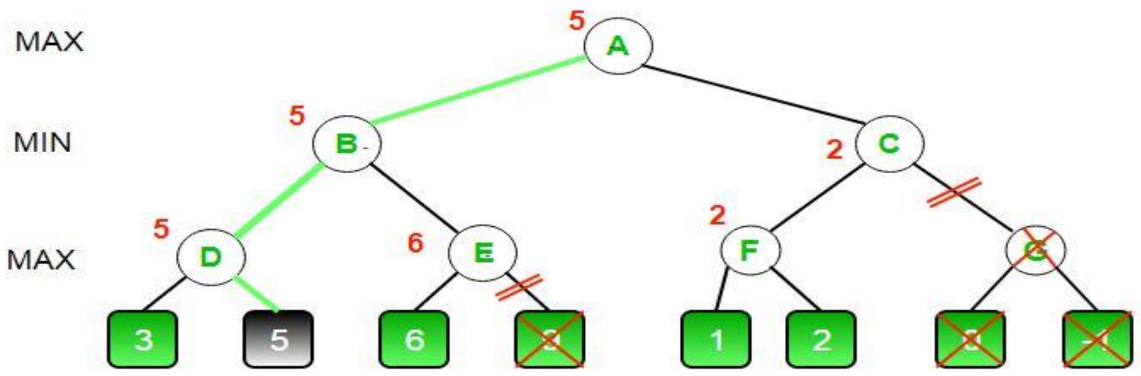
```
def minmax(board, depth):
    global choice
    result = check_win_game(board)
    if result != CONT_GAME:
        return unit_score(result, depth)

    depth += 1
    scores = []
    steps = []

    for step in get_available_step(board):
        score = minmax(update_state(board, step, depth), depth)
        scores.append(score)
        steps.append(step)

    if depth % 2 == 1:
        max_value_index = scores.index(max(scores))
        choice = steps[max_value_index]
        return max(scores)
    else:
        min_value_index = scores.index(min(scores))
        choice = steps[min_value_index]
        return min(scores)
```

2. **Alpha-Beta Pruning-**Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

   Let's define the parameters alpha and beta.
   Alpha is the best value that the **maximizer** currently can guarantee at that level or above.
   Beta is the best value that the **minimizer** currently can guarantee at that level or above.

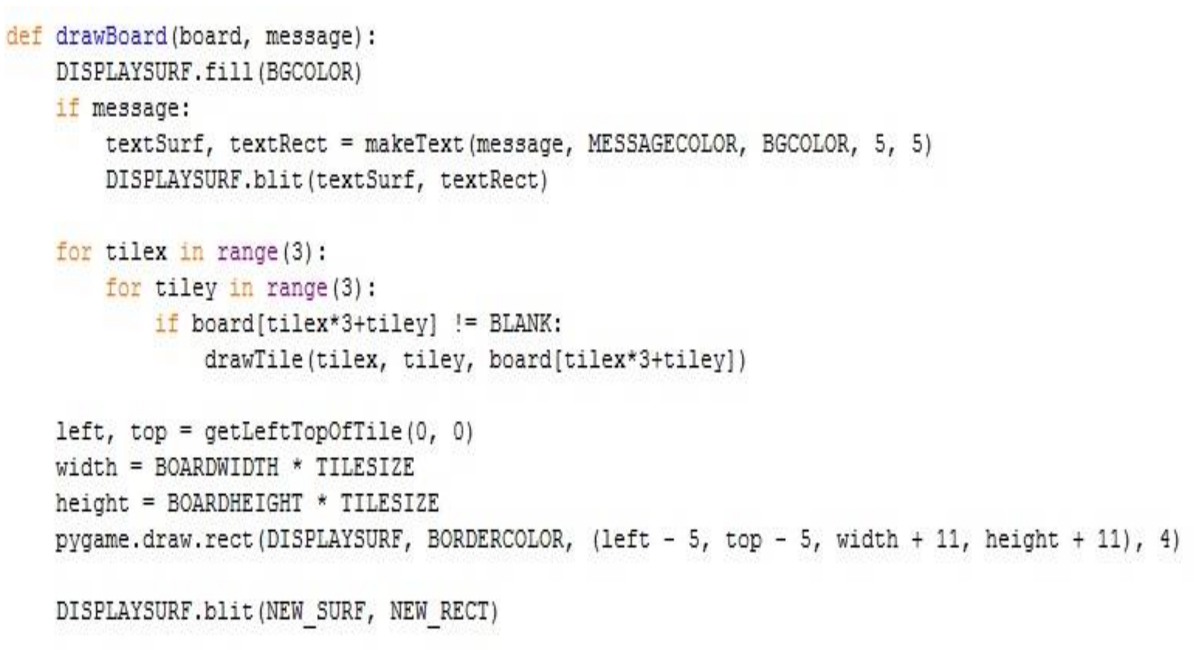
Pseudocode :

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```



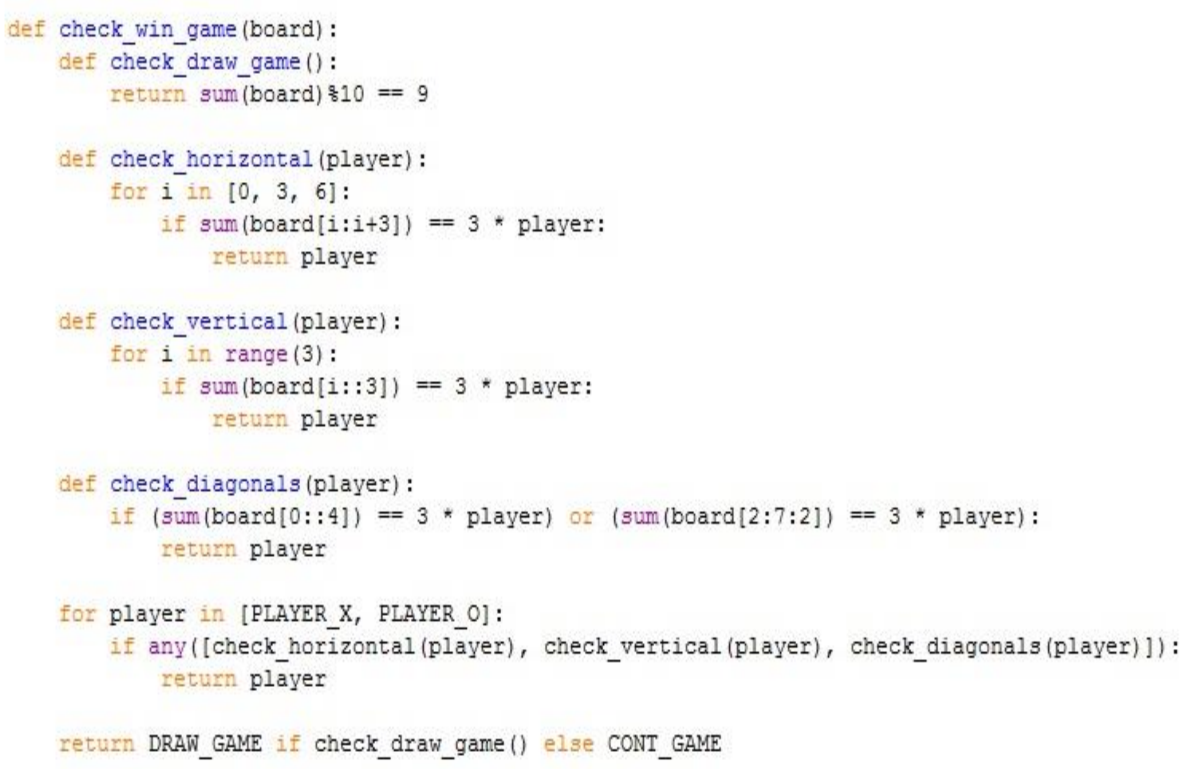
- **B** returns 5 to **A**. At **A**, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater. **A** now calls **C** to see if it can get a higher value than 5.
- At **C**, alpha = 5 and beta = +INF. **C** calls **F**
- At **F**, alpha = 5 and beta = +INF. **F** looks at its left child which is a 1. alpha = max(5, 1) which is still 5.
- **F** looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5
- **F** returns a value of 2 to **C**. At **C**, beta = min(+INF, 2). The condition beta <= alpha becomes false as beta = 2 and alpha = 5. So it breaks and it dose not even have to compute the entire sub-tree of **G**.
- The intuition behind this break off is that, at **C** the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose **B**. So why would the maximizer ever choose **C** and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub tree.
- **C** now returns a value of 2 to **A**. Therefore the best value at **A** is max(5, 2) which is a 5.
- Hence the optimal value that the maximizer can get is 5

Draw and Setup board: Drawing board or the play area in python using pygame involves DisplaySurf and Draw function the colors for background and for the tiles is already defined at the beginning of the source code.

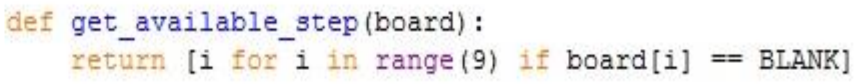


```
def drawBoard(board, message):
    DISPLAYSURF.fill(BGCOLOR)
    if message:
        textSurf, textRect = makeText(message, MESSAGECOLOR, BGCOLOR, 5, 5)
        DISPLAYSURF.blit(textSurf, textRect)

    for tilex in range(3):
        for tiley in range(3):
            if board[tilex*3+tiley] != BLANK:
                drawTile(tilex, tiley, board[tilex*3+tiley])

    left, top = getLeftTopOfTile(0, 0)
    width = BOARDWIDTH * TILESIZE
    height = BOARDHEIGHT * TILESIZE
    pygame.draw.rect(DISPLAYSURF, BORDERCOLOR, (left - 5, top - 5, width + 11, height + 11), 4)

    DISPLAYSURF.blit(NEW_SURF, NEW_RECT)
```

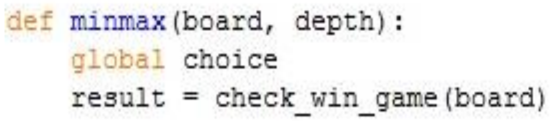
To check Board condition: The game proceeds as long as either one of the player wins or there is no space left for the next move resulting in draw game. For this the game continuously checks the board conditions after every chance for a win game condition or a draw condition and gets the set of available moves.



```
def check_win_game(board):
    def check_draw_game():
        return sum(board)%10 == 9

    def check_horizontal(player):
        for i in [0, 3, 6]:
            if sum(board[i:i+3]) == 3 * player:
                return player

    def check_vertical(player):
        for i in range(3):
            if sum(board[i::3]) == 3 * player:
                return player

    def check_diagonals(player):
        if (sum(board[0::4]) == 3 * player) or (sum(board[2:7:2]) == 3 * player):
            return player

    for player in [PLAYER_X, PLAYER_O]:
        if any([check_horizontal(player), check_vertical(player), check_diagonals(player)]):
            return player

    return DRAW_GAME if check_draw_game() else CONT_GAME
```

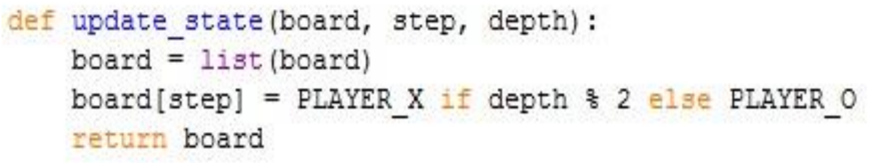


```
def get_available_step(board):
    return [i for i in range(9) if board[i] == BLANK]
```

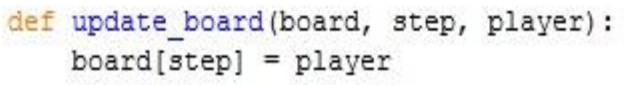


```
def minmax(board, depth):
    global choice
    result = check_win_game(board)
```

Update Board After Moves: There is constant need to update the board after every chance so for this purpose 2 function are made which update board and state of game after every chance enabling user to experience a very responsive and fast gaming experience.



```
def update_state(board, step, depth):
    board = list(board)
    board[step] = PLAYER_X if depth % 2 else PLAYER_O
    return board
```



```
def update_board(board, step, player):
    board[step] = player
```

Main Function: All the function listed above along with some more function are called in main function and when the game is run we get an AI tic tac toe which is difficult wo win over and fast at the same time .For reseting all the board values and then start the new game, there is a button provided named ***"New Game".***

CHAPTER 4 → **PACMAN**

Pacman in Python with PyGame

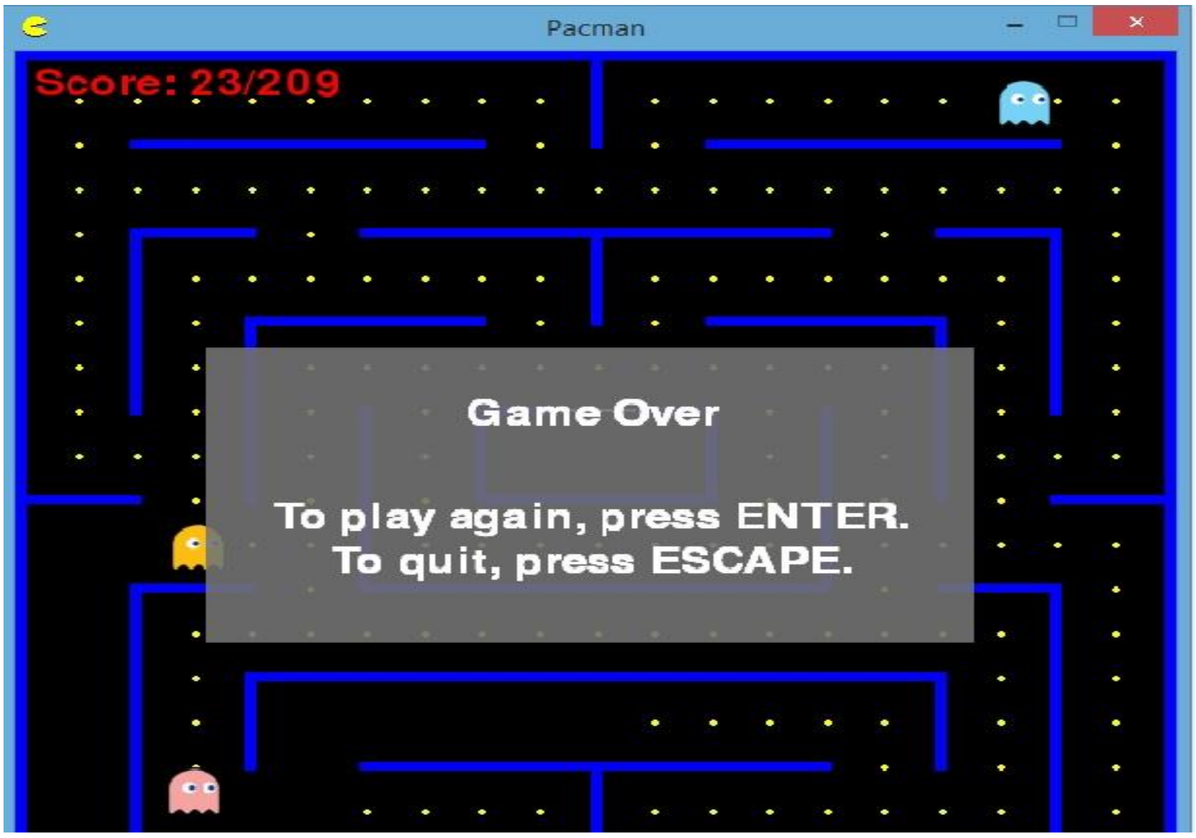
This is a very minimal implementation of the Pacman game, having only one level and with ghosts strategy (the routes are programmed).

Rules of the game:

- The user needs to move pacman using arrow keys and eat up all the dots total 209.
- The user needs to save its pacman from the ghost roaming in the game.
- The game ends when all the dots are eaten by the user or the user's pacman dies by a ghost.

Implementation

The game involves a playing area having some walls and a pacman(controlled by user) along with four ghost. Whenever the game ends the game ask the user weather he wants to continue or wants to exit by pressing enter or escape repectively.

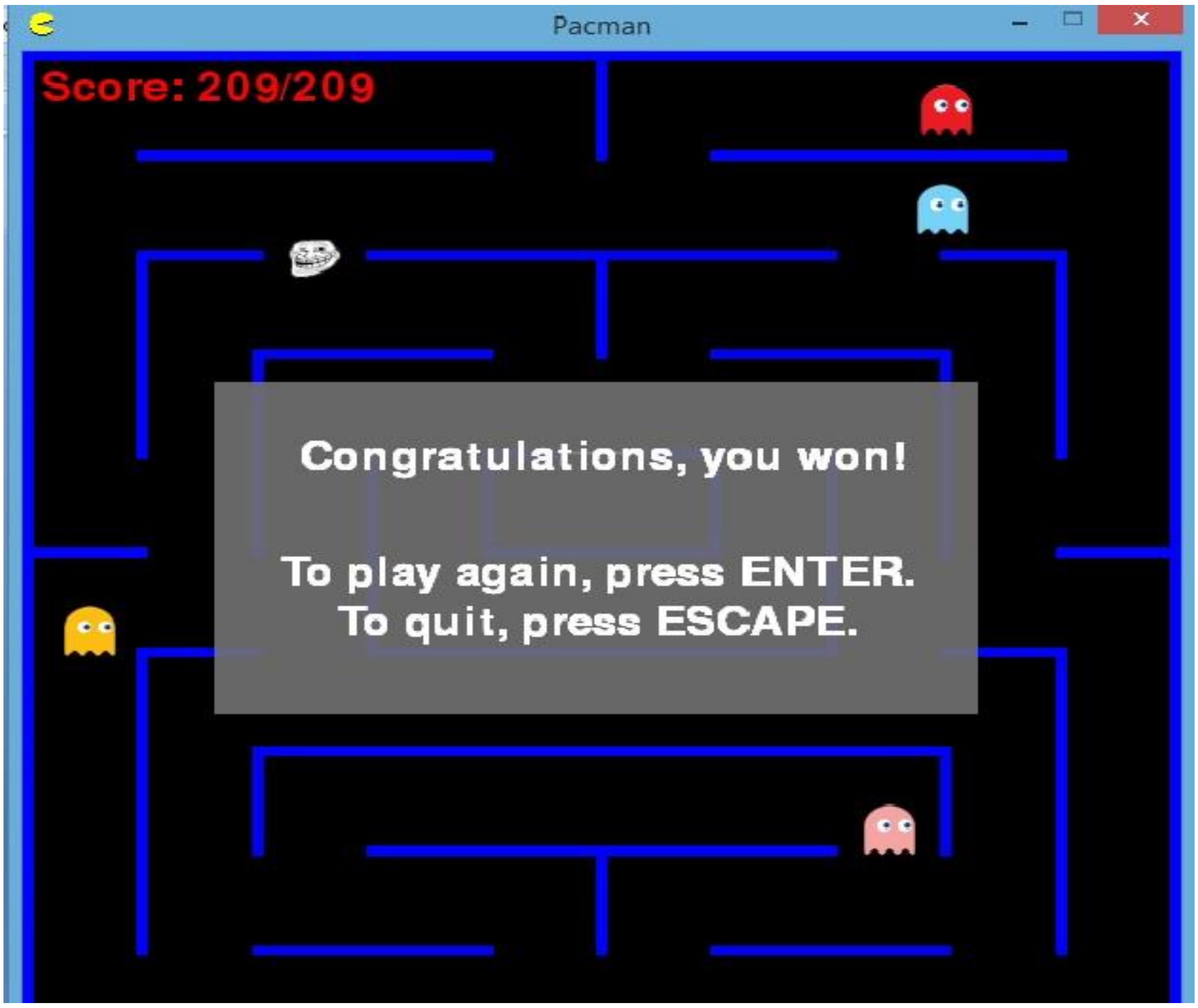


Score: 23/209

Game Over

To play again, press ENTER.
To quit, press ESCAPE.

As in the above screenshot the user lost as it is killed by a ghost.



Congratulations, you won!

To play again, press ENTER.
To quit, press ESCAPE.

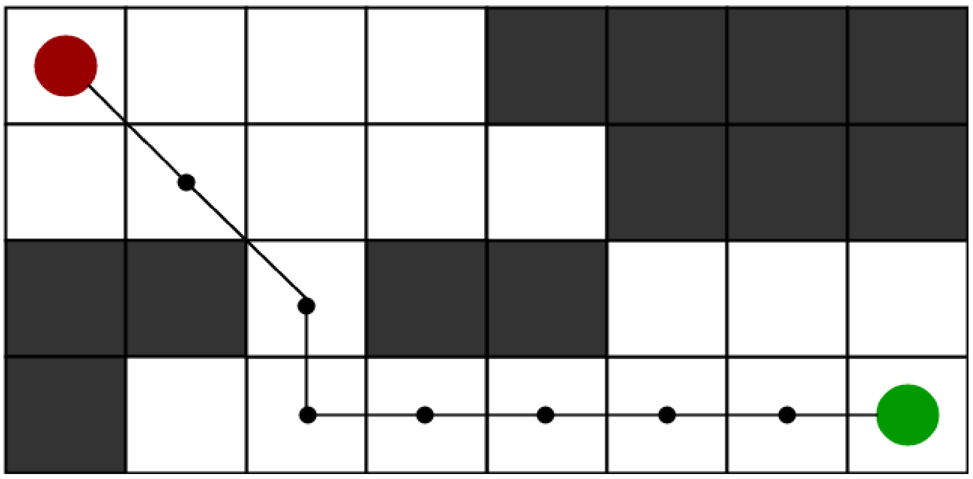
Result displayed after player won the match.

Algorithms: Some of the main algorithms used in the game are:

- A*.
- BFS.
- DFS.
- Ghost Algorithm.

A* Algorithm:

- **Motivation**
  To approximate the shortest path in real-life situations, like- in maps, games where there can be many hindrances.
- We can consider a 2D Grid having several obstacles and we start from a source cell (coloured red below) to reach towards a goal cell (coloured green below



What is A algorithm?*

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Why A algorithm?*

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections. And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.

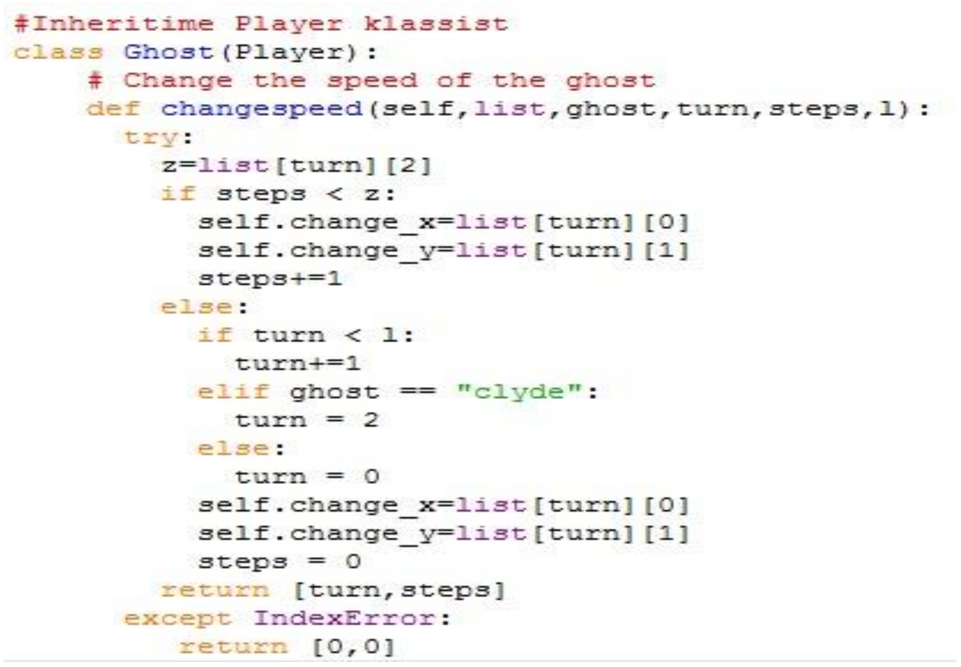
What A* Search Algorithm does is that at each step it picks the node according to a value-‘**f**’ which is a parameter equal to the sum of two other parameters – ‘**g**’ and ‘**h**’. At each step it picks the node/cell having the lowest ‘**f**’, and process that node/cell.

We define ‘**g**’ and ‘**h**’ as simply as possible below
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this ‘h’ which are discussed in the later sections.

Algorithm:

We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

```
// A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
    put the starting node on the open
    list (you can leave its **f** at zero)

3.  while the open list is not empty
    a) find the node with the least **f** on
       the open list, call it "q"

    b) pop q off the open list

    c) generate q's 8 successors and set their
       parents to q

    d) for each successor
        i) if successor is the goal, stop search
          successor.**g** = q.**g** + distance between
                              successor and q
          successor.**h** = distance from goal to
          successor (This can be done using many
          ways, we will discuss three heuristics-
          Manhattan, Diagonal and Euclidean
          Heuristics)

          successor.**f** = successor.**g** + successor.**h**

        ii) if a node with the same position as
            successor is in the OPEN list which has a
           lower **f** than successor, skip this successor

        iii) if a node with the same position as
            successor  is in the CLOSED list which has
            a lower **f** than successor, skip this successor
            otherwise, add  the node to the open list
     end (for loop)

    e) push q on the closed list
    end (while loop)
```



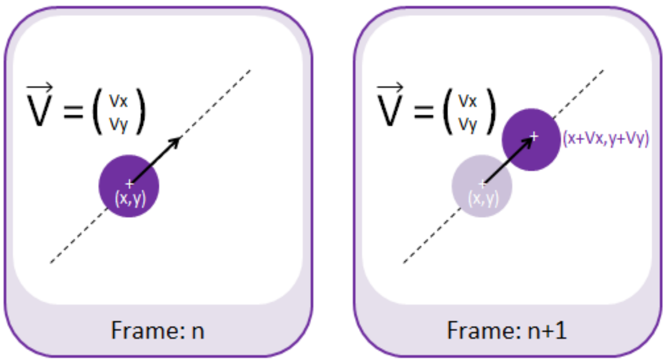
```
#Inheritime Player klassist
class Ghost(Player):
    # Change the speed of the ghost
    def changespeed(self,list,ghost,turn,steps,l):
      try:
        z=list[turn][2]
        if steps < z:
          self.change_x=list[turn][0]
          self.change_y=list[turn][1]
          steps+=1
        else:
          if turn < l:
            turn+=1
          elif ghost == "clyde":
            turn = 2
          else:
            turn = 0
          self.change_x=list[turn][0]
          self.change_y=list[turn][1]
          steps = 0
        return [turn,steps]
      except IndexError:
         return [0,0]
```

Pacman Ghost Algorithm:

In a game of Pacman a specific algorithm is used to control the movement of the ghosts who are chasing (running towards) Pacman.

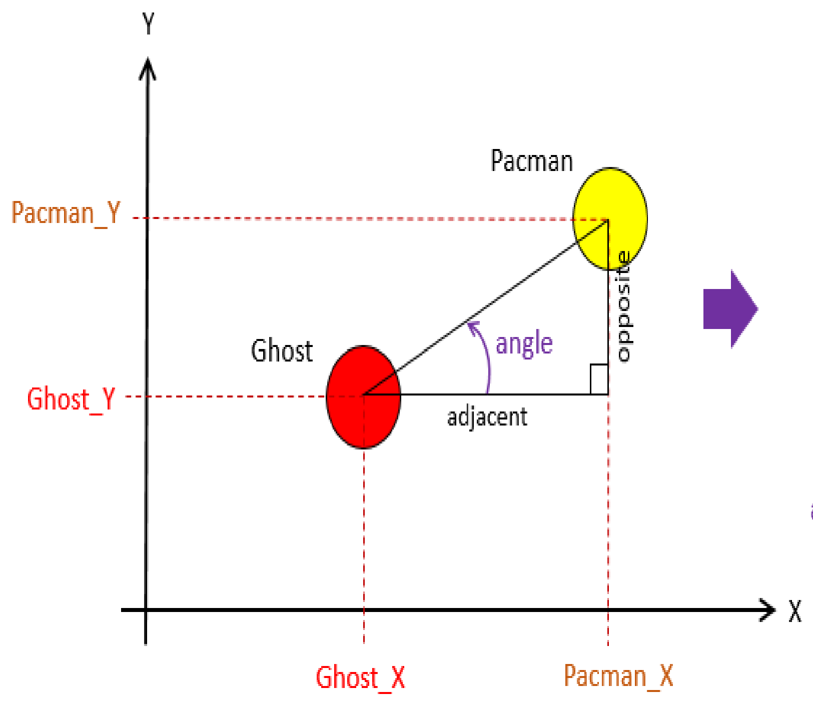
For this challenge we will assume that ghosts can walk through walls (as ghosts do!). So we will implement an algorithm that is slightly different to the algorithm used in the real game of Pacman where ghosts can only run alongside the corridors of the maze.

Our algorithm will be used in a frame based game where the sprites (e.g. Pacman, Ghosts) are positionned using **(x,y) coordinates**. The Pacman movement will be based on the position of the mouse cursor whereas the Ghosts will use a **velocity vector (vx,vy)** to move/translate between two frames.



Trigonometric Ratios: SOH-CAH-TOA:

Our algorithm will use the trigonometric ration to find the angle the Ghost needs to head towards to chase Pacman.

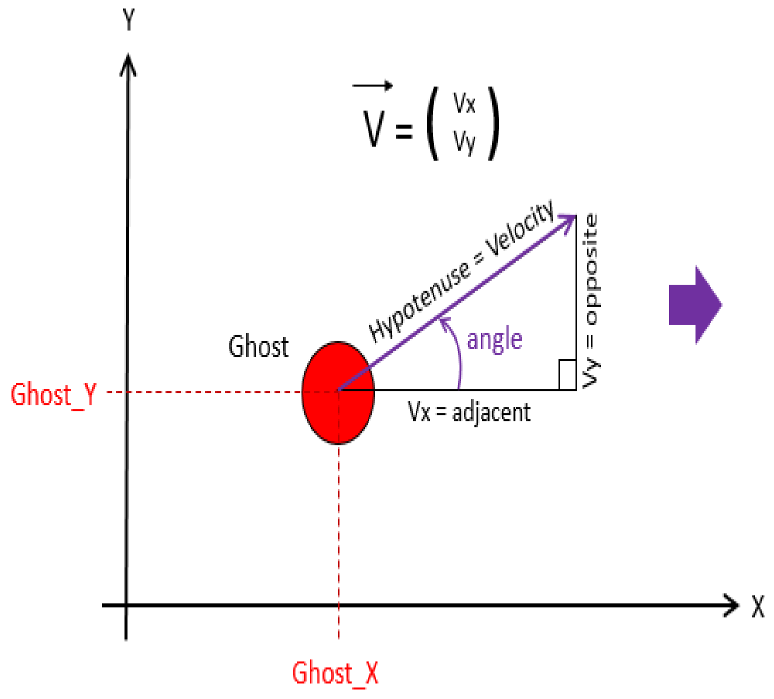


$$\tan(\text{angle}) = \frac{\text{opposite}}{\text{adjacent}}$$

$$\text{angle} = \tan^{-1}\left(\frac{\text{opposite}}{\text{adjacent}}\right)$$

$$\text{angle} = \tan^{-1}\left(\frac{\text{Pacman\_Y} - \text{Ghost\_Y}}{\text{Pacman\_X} - \text{Ghost\_X}}\right)$$

The next step of our algorithm will use this angle to calculate the velocity vector (Vx,Vy) of the ghost:



$$\cos(\text{angle}) = \frac{\text{adjacent}}{\text{hypotenuse}} = \frac{Vx}{\text{Velocity}}$$

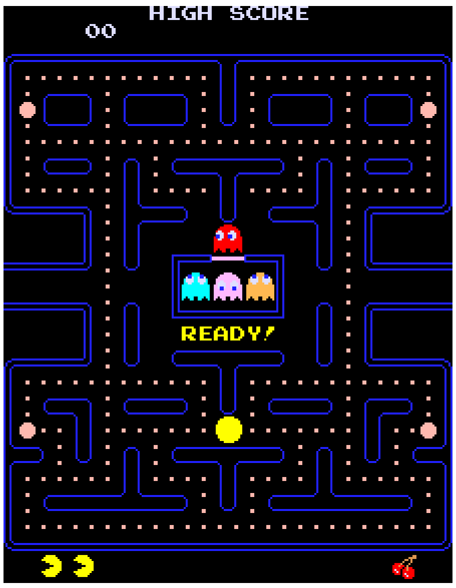
$$Vx = \text{Velocity} \times \cos(\text{angle})$$

$$\sin(\text{angle}) = \frac{\text{opposite}}{\text{hypotenuse}} = \frac{Vy}{\text{Velocity}}$$

$$Vy = \text{Velocity} \times \sin(\text{angle})$$

Understanding Pac-Man Ghost Behavior

Pac-Man is one of the most iconic video games of all time, and most people (even non-gamers) have at least a passing familiarity with it. The purpose of the game is very simple — the player is placed in a maze filled with food (depicted as pellets or dots) and needs to eat all of it to advance to the next level. This task is made difficult by four ghosts that pursue Pac-Man through the maze. If Pac-Man makes contact with any of the ghosts, the player loses a life and the positions of Pac-Man and the ghosts are reset back to their starting locations, though any dots that were eaten remain so. Other than simply avoiding them, Pac-Man's only defense against the ghosts are the four larger "energizer" pellets located at the corners of the maze. Eating one causes the ghosts to become frightened and retreat for a short time, and in the early levels of the game Pac-Man can even eat the ghosts for bonus points during this period. An eaten ghost is not completely eliminated, but is returned to its starting position before resuming its pursuit. Other than eating dots and ghosts, the only other source of points are the two pieces of fruit which appear during each level near the middle of the maze. The first fruit appears when Pac-Man has eaten 70 of the dots in the maze, and the second when 170 have been eaten.



Every level of Pac-Man uses the same maze layout, containing 240 regular "food" dots and 4 energizers. The tunnels that lead off of the left and right edges of the screen act as shortcuts to the opposite side of the screen, and are usable by both Pac-Man and the ghosts, though the ghosts' speed is greatly reduced while they are in the tunnel. Even though the layout is always the same, the levels become increasingly difficult due to modifications to Pac-Man's speed, as well as changes to both the speed and behavior of the ghosts. After reaching level 21, no further changes to the game's mechanics are made, and every level from 21 onwards is effectively identical.

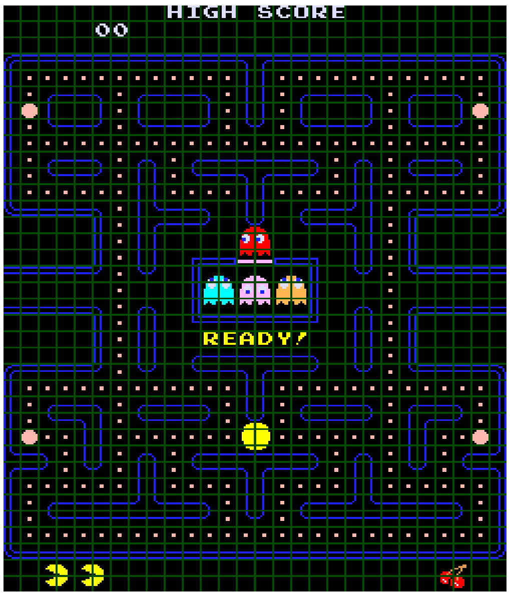
Common Element of Ghost Behaviour

The Ghost House

When a player begins a game of Pac-Man, they are not immediately attacked by all four of the ghosts. As shown on the diagram of the initial game position, only one ghost begins in the actual maze, while the others are inside a small area in the middle of the maze, often referred to as the “ghost house”. Other than at the beginning of a level, the ghosts will only return to this area if they are eaten by an energized Pac-Man, or as a result of their positions being reset when Pac-Man dies. The ghost house is otherwise inaccessible, and is not a valid area for Pac-Man or the ghosts to move into. Ghosts always move to the left as soon as they leave the ghost house, but they may reverse direction almost immediately due to an effect that will be described later.

The conditions that determine when the three ghosts that start inside the ghost house are able to leave it are actually fairly complex. Because of this, I’m going to consider them outside the scope of this article, especially since they become much less relevant after completing the first few levels. If you’re interested in reading about these rules (and an interesting exploit of them), the Pac-Man Dossier covers them in-depth, as always.

Target Tiles



Much of Pac-Man’s design and mechanics revolve around the idea of the board being split into tiles. “Tile” in this context refers to an 8 x 8 pixel square on the screen. Pac-Man’s screen resolution is 224 x 288, so this gives us a total board size of 28 x 36 tiles, though most of these are not accessible to Pac-Man or the ghosts. As an example of the impact of tiles, a ghost is considered to have caught Pac-Man when it occupies the same tile as him. In addition, every pellet in the maze is in the center of its own tile. It should be noted that since the sprites for Pac-Man and the ghosts are larger than one tile in size, they are never completely contained in a single tile. Due to this, for the game’s purposes, the character is considered to occupy whichever tile contains its center point. This is important knowledge when avoiding ghosts, since Pac-Man will only be caught if a ghost manages to move its center point into the same tile as Pac-Man’s.

The key to understanding ghost behavior is the concept of a target tile. The large majority of the time, each ghost has a specific tile that it is trying to reach, and its behavior revolves around trying to get to that tile from its current one. All of the ghosts use identical methods to travel towards their targets, but the different ghost personalities come about due to the individual way each ghost has of selecting its target tile. Note that there are no restrictions that a target tile must actually be possible to reach, they can (and often are) located on an inaccessible tile, and many of the common ghost behaviors are a direct result of this possibility. Target tiles will be discussed in more detail in upcoming sections, but for now just keep in mind that the ghosts are almost always motivated by trying to reach a particular tile.

Ghost Movement Modes

The ghosts are always in one of three possible modes: Chase, Scatter, or Frightened. The "normal" mode with the ghosts pursuing Pac-Man is Chase, and this is the one that they spend most of their time in. While in Chase mode, all of the ghosts use Pac-Man's position as a factor in selecting their target tile, though it is more significant to some ghosts than others. In Scatter mode, each ghost has a fixed target tile, each of which is located just outside a different corner of the maze. This causes the four ghosts to disperse to the corners whenever they are in this mode. Frightened mode is unique because the ghosts do not have a specific target tile while in this mode. Instead, they pseudorandomly decide which turns to make at every intersection. A ghost in Frightened mode also turns dark blue, moves much more slowly and can be eaten by Pac-Man. However, the duration of Frightened mode is shortened as the player progresses through the levels, and is completely eliminated from level 19 onwards.

Changes between Chase and Scatter modes occur on a fixed timer, which causes the "wave" effect described by Iwatani. This timer is reset at the beginning of each level and whenever a life is lost. The timer is also paused while the ghosts are in Frightened mode, which occurs whenever Pac-Man eats an energizer. When Frightened mode ends, the ghosts return to their previous mode, and the timer resumes where it left off. The ghosts start out in Scatter mode, and there are four waves of Scatter/Chase alternation defined, after which the ghosts will remain in Chase mode indefinitely (until the timer is reset). For the first level, the durations of these phases are:

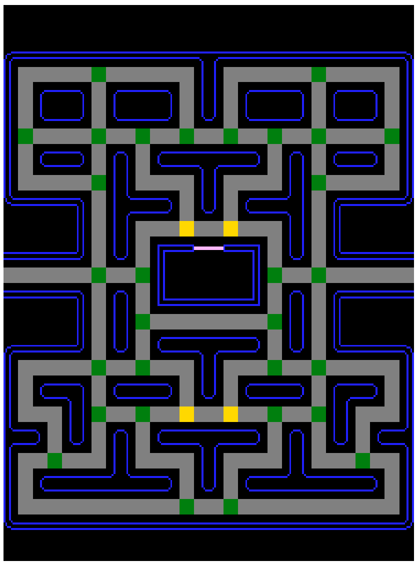
1. Scatter for 7 seconds, then Chase for 20 seconds.
2. Scatter for 7 seconds, then Chase for 20 seconds.
3. Scatter for 5 seconds, then Chase for 20 seconds.
4. Scatter for 5 seconds, then switch to Chase mode permanently.

The durations of these phases are changed somewhat when the player reaches level 2, and once again when they reach level 5. Starting on level 2, the third Chase mode lengthens considerably, to 1033 seconds (17 minutes and 13 seconds), and the following Scatter mode lasts just 1/60 of a second before the ghosts proceed to their permanent Chase mode. The level 5 changes build on top of this, additionally reducing the first two Scatter lengths to 5 seconds, and adding the 4 seconds gained here to the third Chase mode, lengthening it to 1037 seconds (17:17). Regarding the 1/60-of-a-second Scatter mode on every level except the first, even though it may seem that switching modes for such an insignificant amount of time is pointless, there is a reason behind it, which shall be revealed shortly.

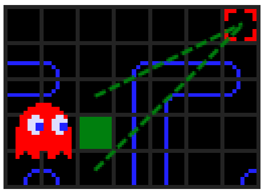
Basic Ghost Movement Rules

The next step is understanding exactly *how* the ghosts attempt to reach their target tiles. The ghosts' AI is very simple and short-sighted, which makes the complex behavior of the ghosts even more impressive. Ghosts only ever plan one step into the future as they move about the maze. Whenever a ghost enters a new tile, it looks ahead to the next tile that it will reach, and makes a decision about which direction it will turn when it gets there. These decisions have one very important restriction, which is that ghosts may never choose to reverse their direction of travel. That is, a ghost cannot enter a tile from the left side and then decide to reverse direction and move back to the left. The implication of this restriction is that whenever a ghost enters a tile with only two exits, it will always continue in the same direction.

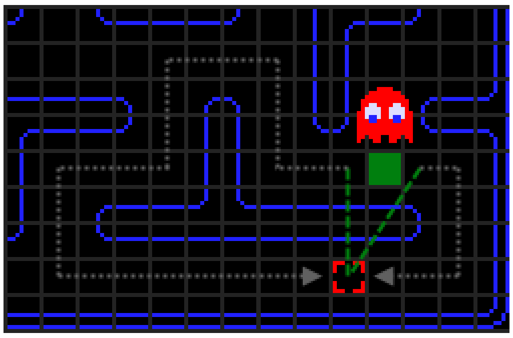
However, there is one exception to this rule, which is that whenever ghosts change from Chase or Scatter to any other mode, they are *forced* to reverse direction as soon as they enter the next tile. This forced instruction will overwrite whatever decision the ghosts had previously made about the direction to move when they reach that tile. This effectively acts as a notifier to the player that the ghosts have changed modes, since it is the only time a ghost can possibly reverse direction. Note that when the ghosts leave Frightened mode they do not change direction, but this particular switch is already obvious due to the ghosts reverting to their regular colors from the dark blue of Frightened. So then, the 1/60-of-a-second Scatter mode on every level after the first will cause all the ghosts to reverse their direction of travel, even though their target effectively remains the same. This forced direction-reversal instruction is also applied to any ghosts still inside the ghost house, so a ghost that hasn't yet entered the maze by the time the first mode switch occurs will exit the ghost house with a "reverse direction as soon as you can" instruction already pending. This causes them to move left as usual for a very short time, but they will almost immediately reverse direction and go to the right instead.



The diagram above shows a simplified representation of the maze layout. Decisions are only necessary at all when approaching "intersection" tiles, which are indicated in green on the diagram.



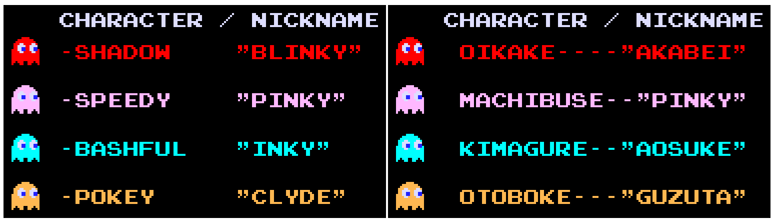
When a decision about which direction to turn is necessary, the choice is made based on which tile adjoining the intersection will put the ghost nearest to its target tile, measured in a straight line. The distance from every possibility to the target tile is measured, and whichever tile is closest to the target will be selected. In the diagram to the left, the ghost will turn upwards at the intersection. If two or more potential choices are an equal distance from the target, the decision between them is made in the order of up > left > down. A decision to exit right can never be made in a situation where two tiles are equidistant to the target, since any other option has a higher priority.



Since the only consideration is which tile will immediately place the ghost closer to its target, this can result in the ghosts selecting the "wrong" turn when the initial choice places them closer, but the overall path is longer. An example is shown to the right, where straight-line measurement makes exiting left appear to be a better choice. However, this will result in an overall path length of 26 tiles to reach the target, when exiting right would have had a path only 8 tiles long.

One final special case to be aware of are the four intersections that were colored yellow on the simplified maze diagram. These specific intersections have an extra restriction — ghosts can not choose to turn upwards from these tiles. If entering them from the right or left side they will always proceed out the opposite side (excepting a forced direction-reversal). Note that this restriction does not apply to Frightened mode, and Frightened ghosts may turn upwards here if that decision occurs randomly. A ghost entering these tiles from the top can also reverse direction back out the top if a mode switch occurs as they are entering the tile, the restriction is only applied during "regular" decision-making. If Pac-Man is being pursued closely by ghosts, he can gain some ground on them by making an upwards turn in one of these intersections, since they will be forced to take a longer route around.

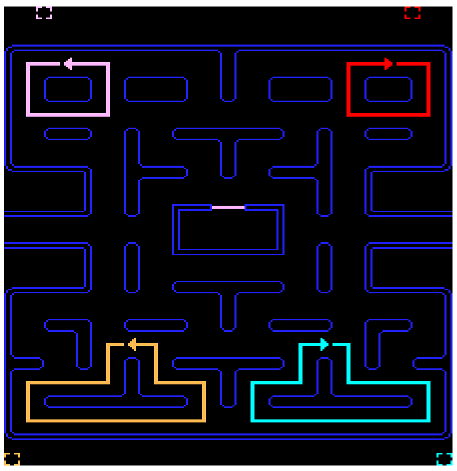
Individual Ghost Personalities



CHARACTER /	NICKNAME	CHARACTER /	NICKNAME
-SHADOW	"BLINKY"	OIKAKE---	"AKABEI"
-SPEEDY	"PINKY"	MACHIBUSE--	"PINKY"
-BASHFUL	"INKY"	KIMAGURE--	"AOSUKE"
-POKEY	"CLYDE"	OTOBOKE---	"GUZUTA"

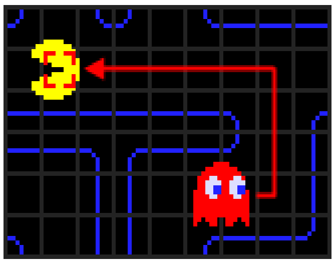
As has been previously mentioned, the only differences between the ghosts are their methods of selecting target tiles in Chase and Scatter modes. The only official description of each ghost's personality comes from the one-word "character" description shown in the game's attract mode. We'll first take a look at how the ghosts behave in Scatter mode, since it's extremely straightforward, and then look at each ghost's approach to targeting in Chase mode.

Scatter Mode



Each ghost has a pre-defined, fixed target tile while in this mode, located just outside the corners of the maze. When Scatter mode begins, each ghost will head towards their "home" corner using their regular path-finding methods. However, since the actual target tiles are inaccessible and the ghosts cannot stop moving or reverse direction, they are forced to continue past the target, but will turn back towards it as soon as possible. This results in each ghost's path eventually becoming a fixed loop in their corner. If left in Scatter mode, each ghost would remain in its loop indefinitely. In practice, the duration of Scatter mode is always quite short, so the ghosts often do not have time to even reach their corner or complete a circuit of their loop before reverting back to Chase mode. The diagram shows each ghost's target tile and eventual looping path, color-coded to match their own color.

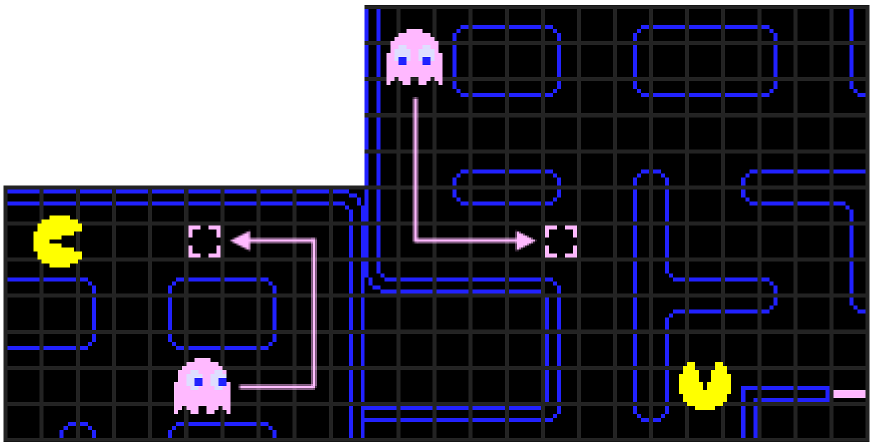
The Red Ghost



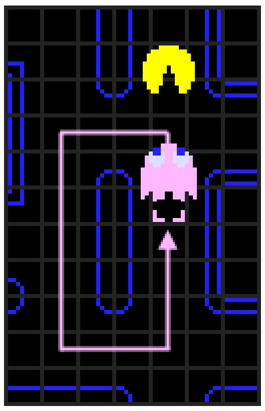
The red ghost starts outside of the ghost house, and is usually the first one to be seen as a threat, since he makes a beeline for Pac-Man almost immediately. He is referred to as "Blinky", and the game describes his personality as *shadow*. In Japanese, his personality is referred to as 追いかけ, *oikake*, which translates as "pursuer" or "chaser". Both languages' descriptions are accurate, since Blinky's target tile in Chase mode is defined as Pac-Man's current tile. This ensures that Blinky almost always follows directly behind Pac-Man, unless the short-sighted decision-making causes him to take an inefficient path.

Even though Blinky's targeting method is very simple, he does have one idiosyncrasy that the other ghosts do not; at two defined points in each level (based on the number of dots remaining), his speed increases by 5% and his behavior in Scatter mode changes. The timing of the speed change varies based on the level, with the change occurring earlier and earlier as the player progresses. The change to Scatter targeting is perhaps more significant than the speed increases, since it causes Blinky's target tile to remain as Pac-Man's position even while in Scatter mode, instead of his regular fixed tile in the upper-right corner. This effectively keeps Blinky in Chase mode permanently, though he will still be forced to reverse direction as a result of a mode switch. When in this enhanced state, Blinky is generally referred to as "Cruise Elroy", though the origin of this term seems to be unknown. Not even the almighty Pac-Man Dossier has an answer here. If Pac-Man dies while Blinky is in Cruise Elroy mode, he reverts back to normal behavior temporarily, but returns to Elroy mode as soon as all other ghosts have exited the ghost house.

The Pink Ghost

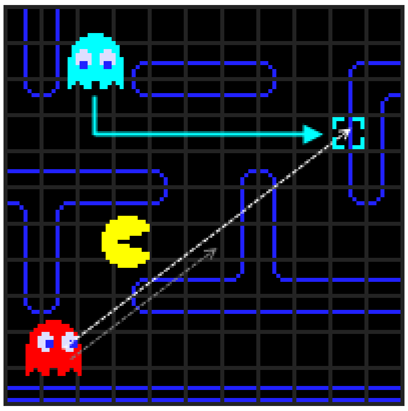


The pink ghost starts inside the ghost house, but always exits immediately, even in the first level. His nickname is “Pinky”, and his personality is described as *speedy*. This is a considerable departure from his Japanese personality description, which is *machibuse*, which translates as “ambusher”. The Japanese version is much more appropriate, since Pinky does not move faster than any of the other ghosts (and slower than Blinky in Cruise Elroy mode), but his targeting scheme attempts to move him to the place where Pac-Man is going, instead of where he currently is. Pinky’s target tile in Chase mode is determined by looking at Pac-Man’s current position and orientation, and selecting the location four tiles straight ahead of Pac-Man. At least, this was the intention, and it works when Pac-Man is facing to the left, down, or right, but when Pac-Man is facing upwards, an overflow error in the game’s code causes Pinky’s target tile to actually be set as four tiles ahead of Pac-Man *and* four tiles to the left of him. I don’t wish to frighten off non-programmers, but if you’re interested in the technical details behind this bug, including the actual assembly code for Pinky’s targeting, as well as a fixed version.



One important implication of Pinky’s targeting method is that Pac-Man can often win a game of “chicken” with him. Since his target tile is set four tiles in front of Pac-Man, if Pac-Man heads directly towards him, Pinky’s target tile will actually be *behind* himself once they are less than four tiles apart. This will cause Pinky to choose to take any available turn-off in order to loop back around to his target. Because of this, it is a common strategy to momentarily “fake” back towards Pinky if he starts following closely. This will often send him off in an entirely different direction.

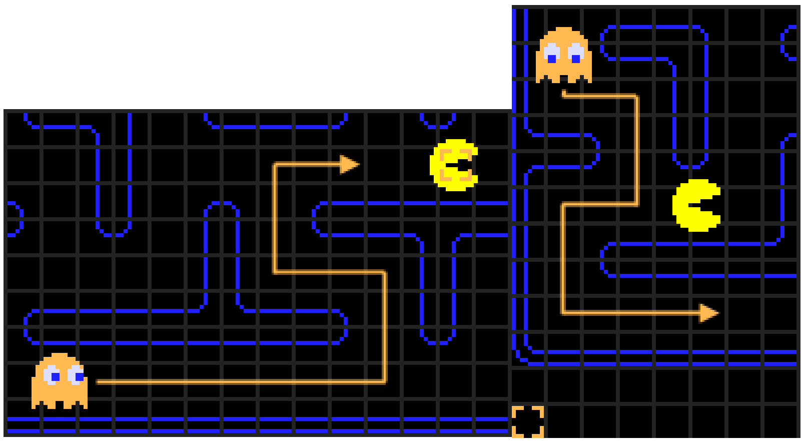
The Blue Ghost



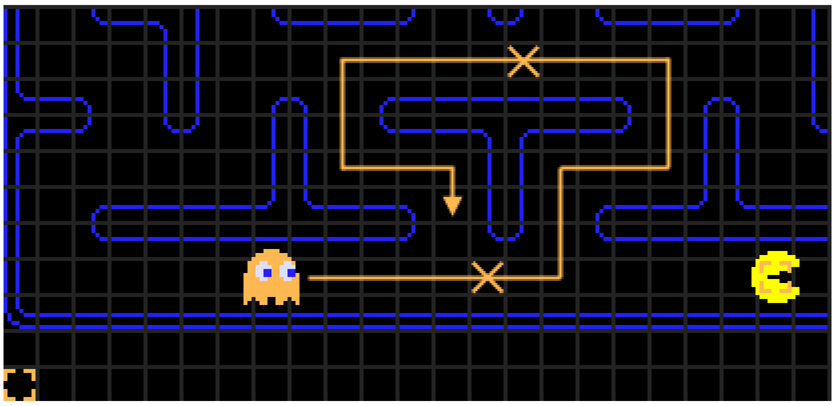
The blue ghost is nicknamed Inky, and remains inside the ghost house for a short time on the first level, not joining the chase until Pac-Man has managed to consume at least 30 of the dots. His English personality description is *bashful*, while in Japanese he is referred to as 気紛れ, *kimagure*, or “whimsical”. Inky is difficult to predict, because he is the only one of the ghosts that uses a factor other than Pac-Man’s position/orientation when determining his target tile. Inky actually uses both Pac-Man’s position/facing as well as Blinky’s (the red ghost’s) position in his calculation. To locate Inky’s target, we first start by selecting the position two tiles in front of Pac-Man in his current direction of travel, similar to Pinky’s targeting method. From there, imagine drawing a vector from Blinky’s position to this tile, and then doubling the length of the vector. The tile that this new, extended vector ends on will be Inky’s actual target.

As a result, Inky’s target can vary wildly when Blinky is not near Pac-Man, but if Blinky is in close pursuit, Inky generally will be as well. Note that Inky’s “two tiles in front of Pac-Man” calculation suffers from exactly the same overflow error as Pinky’s four-tile equivalent, so if Pac-Man is heading upwards, the endpoint of the initial vector from Blinky (before doubling) will actually be two tiles up and two tiles left of Pac-Man.

The Orange Ghost



The orange ghost, “Clyde”, is the last to leave the ghost house, and does not exit at all in the first level until over a third of the dots have been eaten. Clyde’s English personality description is *pokey*, whereas the Japanese description is お惚け, *otoboke* or “feigning ignorance”. As is typical, the Japanese version is more accurate, since Clyde’s targeting method can give the impression that he is just “doing his own thing”, without concerning himself with Pac-Man at all. The unique feature of Clyde’s targeting is that it has two separate modes which he constantly switches back and forth between, based on his proximity to Pac-Man. Whenever Clyde needs to determine his target tile, he first calculates his distance from Pac-Man. If he is *farther* than eight tiles away, his targeting is identical to Blinky’s, using Pac-Man’s current tile as his target. However, as soon as his distance to Pac-Man becomes *less* than eight tiles, Clyde’s target is set to the same tile as his fixed one in Scatter mode, just outside the bottom-left corner of the maze.



The combination of these two methods has the overall effect of Clyde alternating between coming directly towards Pac-Man, and then changing his mind and heading back to his corner whenever he gets too close. On the diagram above, the X marks on the path represent the points where Clyde’s mode switches. If Pac-Man somehow managed to remain stationary in that position, Clyde would indefinitely loop around that T-shaped area. As long as the player is not in the lower-left corner of the maze, Clyde can be avoided completely by simply ensuring that you do not block his “escape route” back to his corner. While Pac-Man is within eight tiles of the lower-left corner, Clyde’s path will end up in exactly the same loop as he would eventually maintain in Scatter mode.

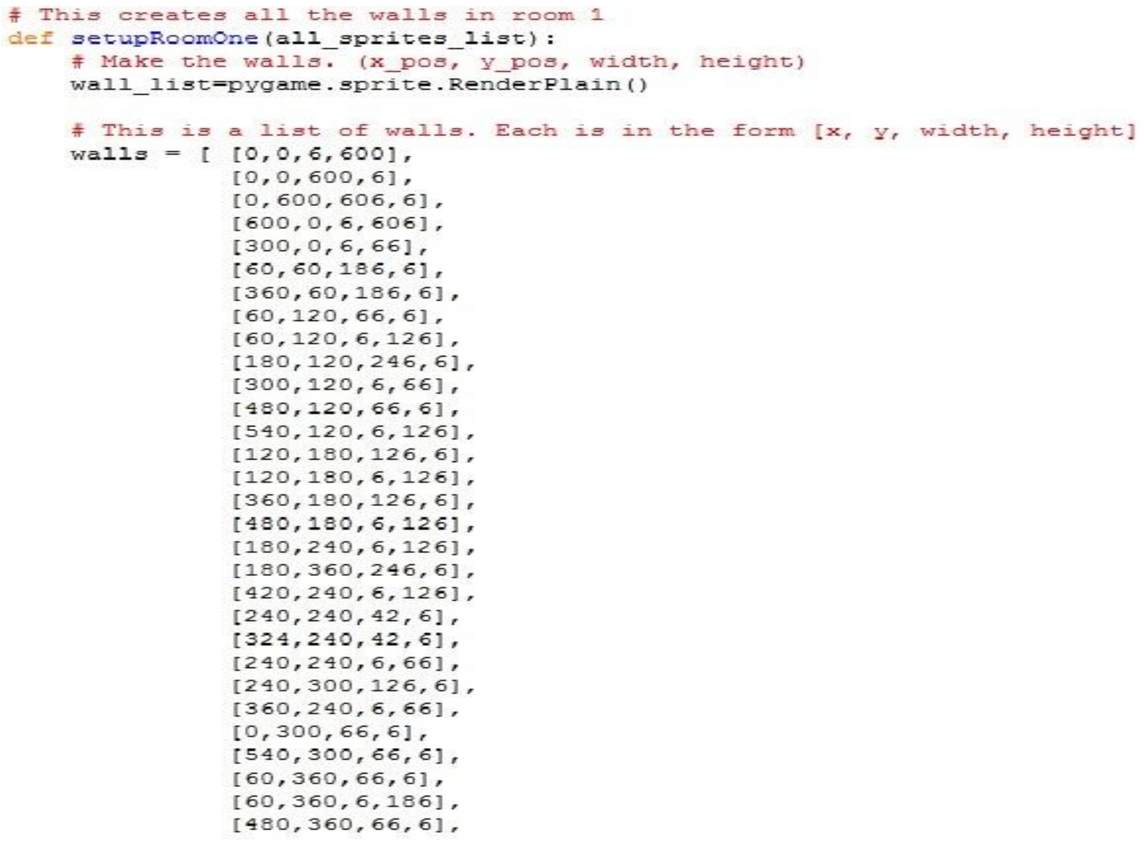
Wrapping Up

If you’ve made it this far, you should now have a fairly complete understanding of the logic behind Pac-Man’s ghost movement. Understanding the ghosts’ behavior is probably the single most important step towards becoming a skilled Pac-Man player, and even a general idea of where they are likely to move next should greatly improve your abilities. I’ve never been good at Pac-Man, but while I was researching this article and testing a few things, I found that I was able to avoid the ghosts much more easily than before. Even small things make a huge difference, such as recognizing a switch to Scatter mode and knowing that you have a few seconds where the ghosts won’t (deliberately) try to kill you.

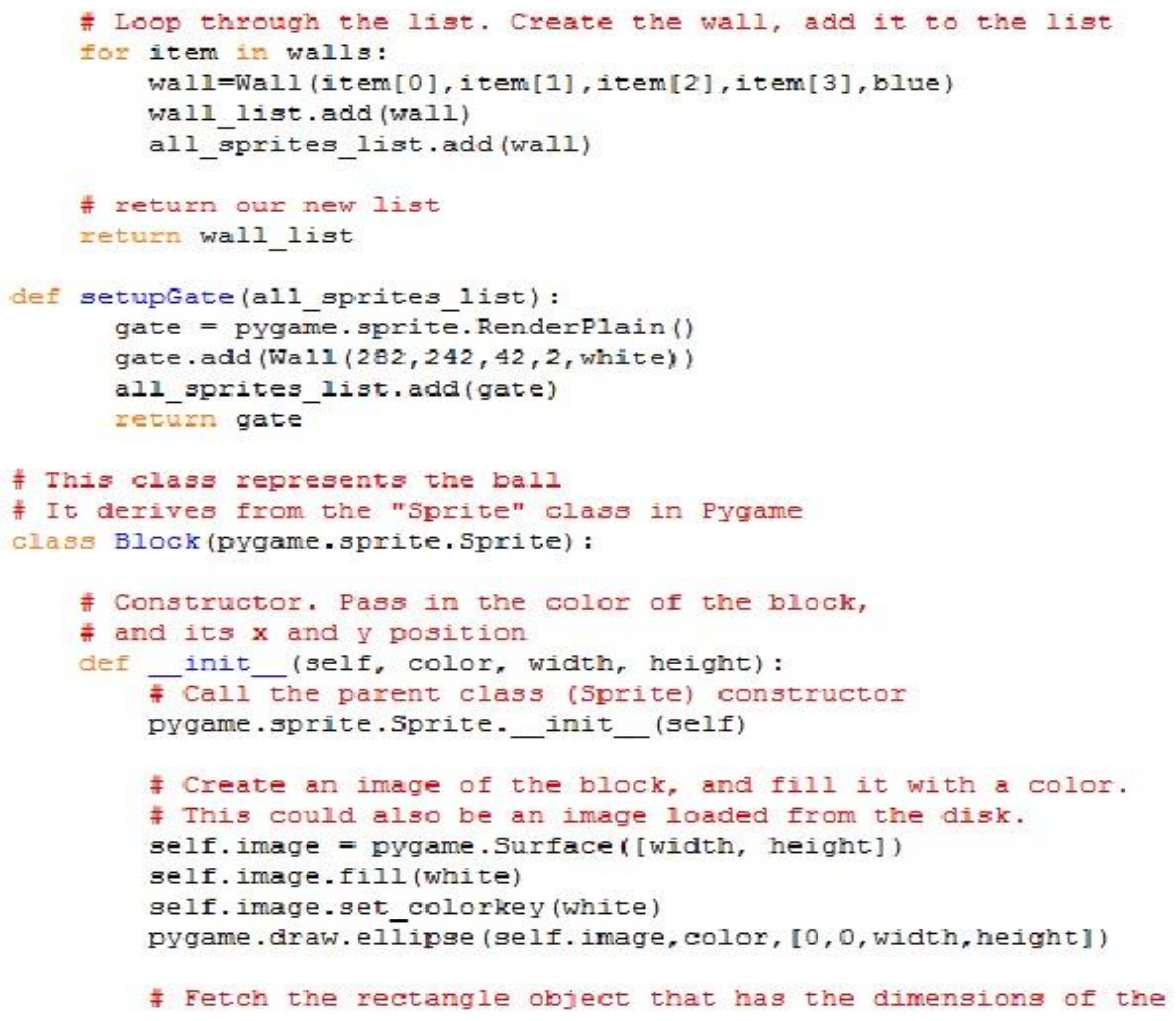
Pac-Man is an amazing example of seemingly-complex behavior arising from only a few cleverly-designed rules, with the result being a deep and challenging game that players still strive to master, 30 years after its release.

Source code snaps:

Draw room : The room for playing the game along with walls for obstruction can be implemented by defining function having list of coordinates of walls and gate from where the ghosts will come out.

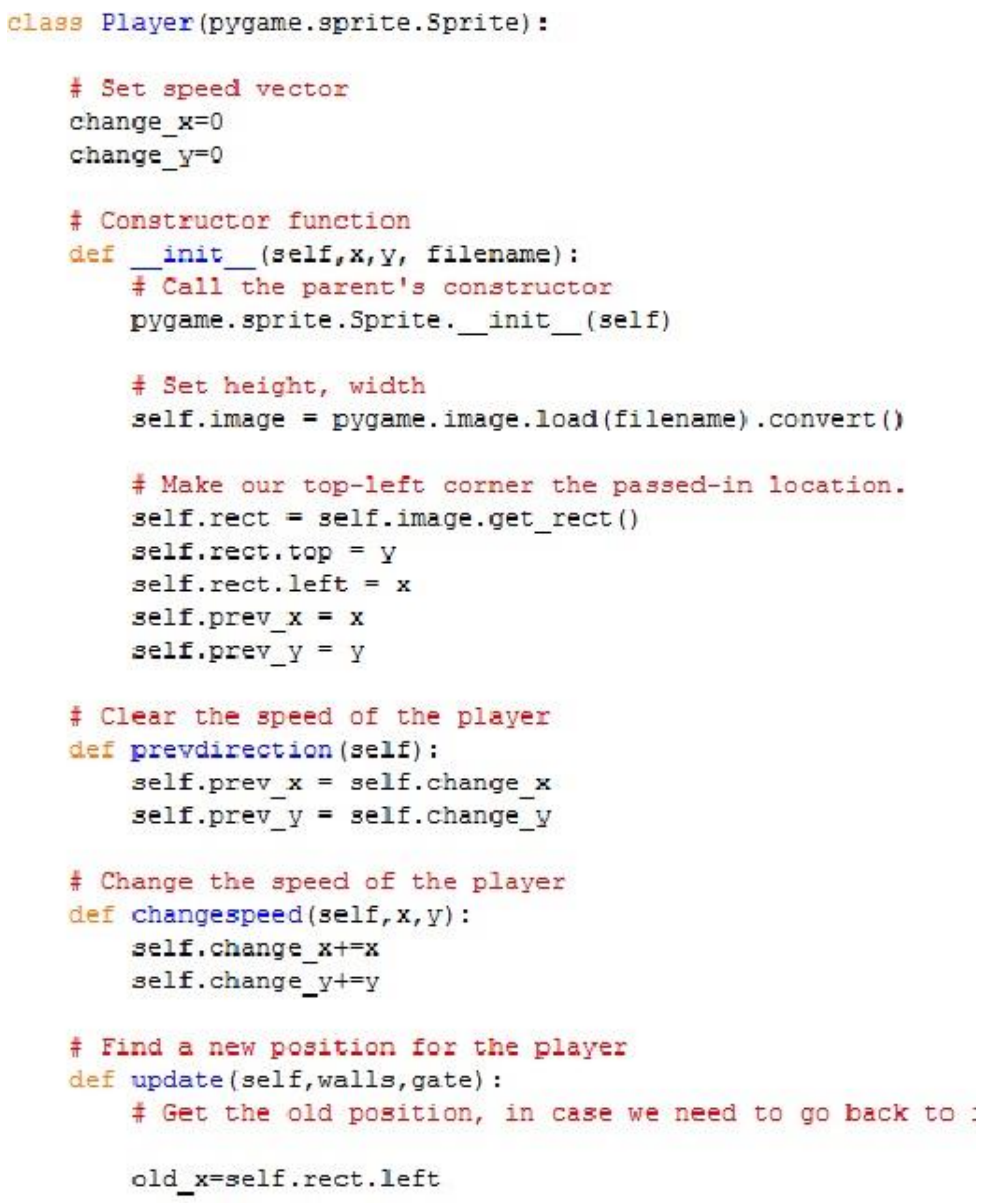


```
# This creates all the walls in room 1
def setupRoomOne(all_sprites_list):
    # Make the walls. (x_pos, y_pos, width, height)
    wall_list=pygame.sprite.RenderPlain()

    # This is a list of walls. Each is in the form [x, y, width, height]
    walls = [ [0,0,6,600],
              [0,0,600,6],
              [0,600,606,6],
              [600,0,6,606],
              [300,0,6,66],
              [60,60,186,6],
              [360,60,186,6],
              [60,120,66,6],
              [60,120,6,126],
              [180,120,246,6],
              [300,120,6,66],
              [480,120,66,6],
              [540,120,6,126],
              [120,180,126,6],
              [120,180,6,126],
              [360,180,126,6],
              [480,180,6,126],
              [180,240,6,126],
              [180,360,246,6],
              [420,240,6,126],
              [240,240,42,6],
              [324,240,42,6],
              [240,240,6,66],
              [240,300,126,6],
              [360,240,6,66],
              [0,300,66,6],
              [540,300,66,6],
              [60,360,66,6],
              [60,360,6,186],
              [480,360,66,6],
```



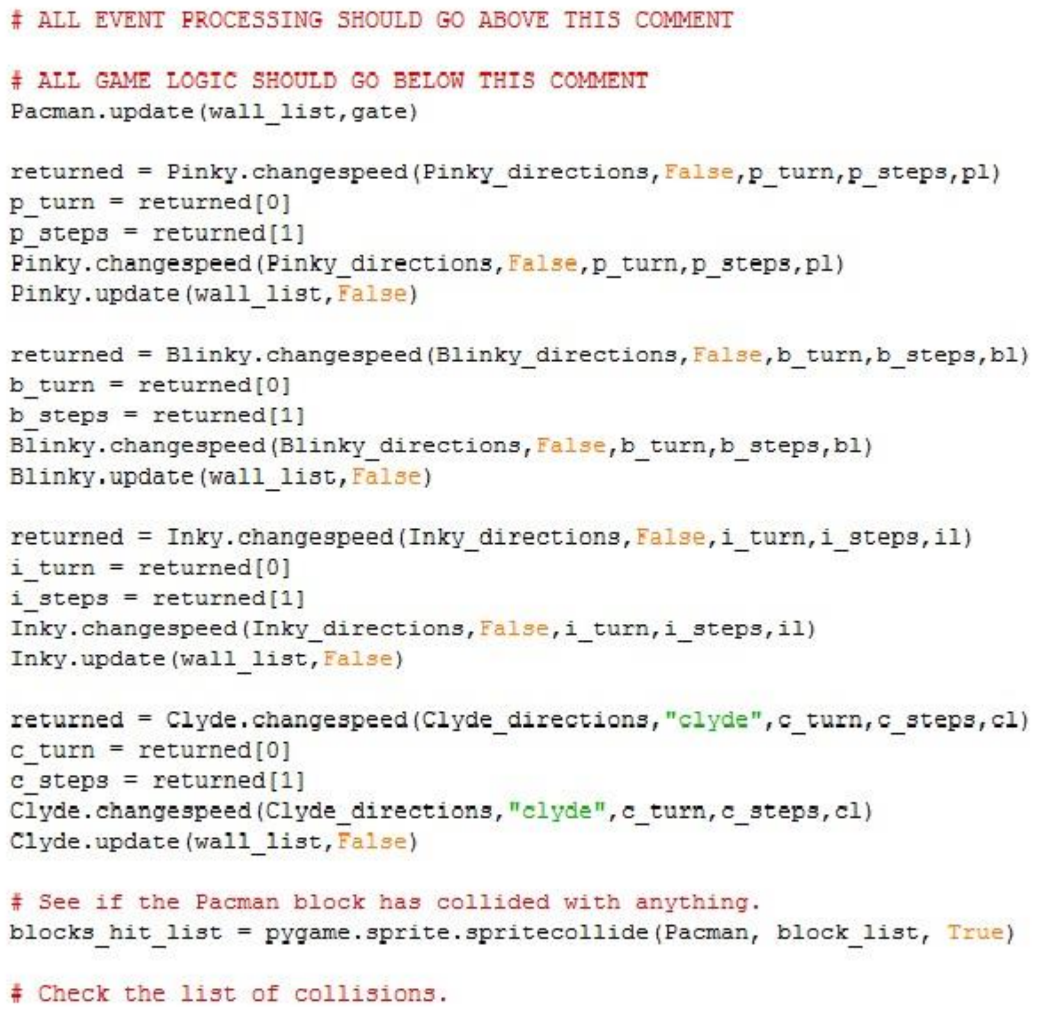
```
    # Loop through the list. Create the wall, add it to the list
    for item in walls:
        wall=Wall(item[0],item[1],item[2],item[3],blue)
        wall_list.add(wall)
        all_sprites_list.add(wall)

    # return our new list
    return wall_list

def setupGate(all_sprites_list):
      gate = pygame.sprite.RenderPlain()
      gate.add(Wall(282,242,42,2,white))
      all_sprites_list.add(gate)
      return gate

# This class represents the ball
# It derives from the "Sprite" class in Pygame
class Block(pygame.sprite.Sprite):

    # Constructor. Pass in the color of the block,
    # and its x and y position
    def __init__(self, color, width, height):
        # Call the parent class (Sprite) constructor
        pygame.sprite.Sprite.__init__(self)

        # Create an image of the block, and fill it with a color.
        # This could also be an image loaded from the disk.
        self.image = pygame.Surface([width, height])
        self.image.fill(white)
        self.image.set_colorkey(white)
        pygame.draw.ellipse(self.image,color,[0,0,width,height])

        # Fetch the rectangle object that has the dimensions of the
```

Player Setup: To setup the initial co ordinates ,direction, speed of the pacman a function is defined in the class player. The main purpose of this class is to have the entire attribute related to pacman and also move it and update it with respect to directions givenby user using arrow key. The function first gets the current coordinates of the pacman and then updates it with new one by adding the values to the previous coordinates. The python code for the above mentioned task is given below:



```
class Player(pygame.sprite.Sprite):

    # Set speed vector
    change_x=0
    change_y=0

    # Constructor function
    def __init__(self,x,y, filename):
        # Call the parent's constructor
        pygame.sprite.Sprite.__init__(self)

        # Set height, width
        self.image = pygame.image.load(filename).convert()

        # Make our top-left corner the passed-in location.
        self.rect = self.image.get_rect()
        self.rect.top = y
        self.rect.left = x
        self.prev_x = x
        self.prev_y = y

    # Clear the speed of the player
    def prevdirection(self):
        self.prev_x = self.change_x
        self.prev_y = self.change_y

    # Change the speed of the player
    def changespeed(self,x,y):
        self.change_x+=x
        self.change_y+=y

    # Find a new position for the player
    def update(self,walls,gate):
        # Get the old position, in case we need to go back to :

        old_x=self.rect.left
```

Game Logic: The game logic is a part of game where processing related to current position of game happens.It basically computes the ghost’s movement,pacman collision, win and loss condition.



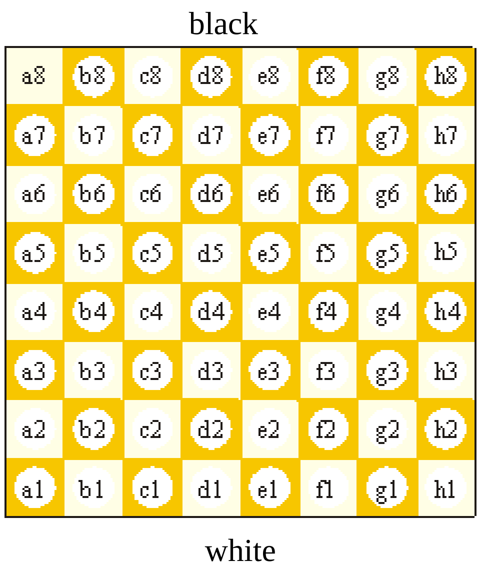
```
# ALL EVENT PROCESSING SHOULD GO ABOVE THIS COMMENT

# ALL GAME LOGIC SHOULD GO BELOW THIS COMMENT
Pacman.update(wall_list,gate)

returned = Pinky.changespeed(Pinky_directions,False,p_turn,p_steps,pl)
p_turn = returned[0]
p_steps = returned[1]
Pinky.changespeed(Pinky_directions,False,p_turn,p_steps,pl)
Pinky.update(wall_list,False)

returned = Blinky.changespeed(Blinky_directions,False,b_turn,b_steps,bl)
b_turn = returned[0]
b_steps = returned[1]
Blinky.changespeed(Blinky_directions,False,b_turn,b_steps,bl)
Blinky.update(wall_list,False)

returned = Inky.changespeed(Inky_directions,False,i_turn,i_steps,il)
i_turn = returned[0]
i_steps = returned[1]
Inky.changespeed(Inky_directions,False,i_turn,i_steps,il)
Inky.update(wall_list,False)

returned = Clyde.changespeed(Clyde_directions,"clyde",c_turn,c_steps,cl)
c_turn = returned[0]
c_steps = returned[1]
Clyde.changespeed(Clyde_directions,"clyde",c_turn,c_steps,cl)
Clyde.update(wall_list,False)

# See if the Pacman block has collided with anything.
blocks_hit_list = pygame.sprite.spritecollide(Pacman, block_list, True)

# Check the list of collisions.
```

CHAPTER 5 → **MAKING OF CHESS**

<u>**The Rules of Chess**</u>**:**

Chess is a game, played by two players. One player plays with the white pieces, and the other player plays with the black pieces. Each player has sixteen pieces in the beginning of the game: one king, one queen, two rooks, two bishops, two knights, and eight pawns.

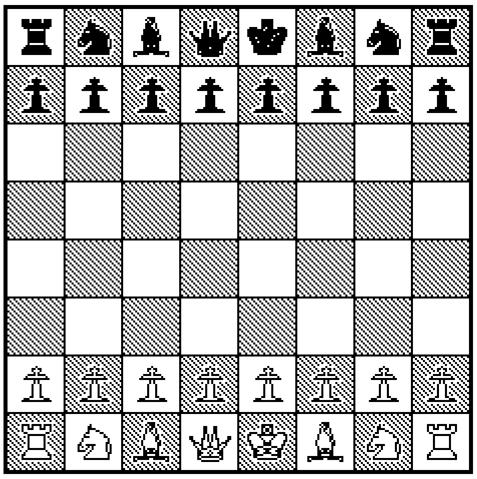
The game is played on a chessboard, consisting of 64 squares: eight rows and eight columns. The squares are alternately light (white) and dark colored. The board must be laid down such that there is a black square in the lower-left corner. To facilitate notation of moves, all squares are given a name. From the view of the white player, the rows are numbered 1, 2, 3, 4, 5, 6, 7, 8; the lowest row has number 1, and the upper row has number 8. The columns are named, from left to right, a, b, c, d, e, f, g, h. A square gets a name, consisting of the combination of its column-letter and row-number, e.g., the square in the lower left corner (for white) is a1.



Alternately, the players make a move, starting with the white player (the player that plays with the white pieces.) A move consists of moving one of the pieces of the player to a different square, following the rules of movement for that piece - there is one special exception, named **castling**, where players move two pieces simultaneously.

A player can **take** a piece of the opponent by moving one of his own pieces to the square that contains a piece of the opponent. The opponents piece then is removed from the board, and out of play for the rest of the game. (Taking is not compulsory.)

At the start of the game, the position of the pieces is as follows.

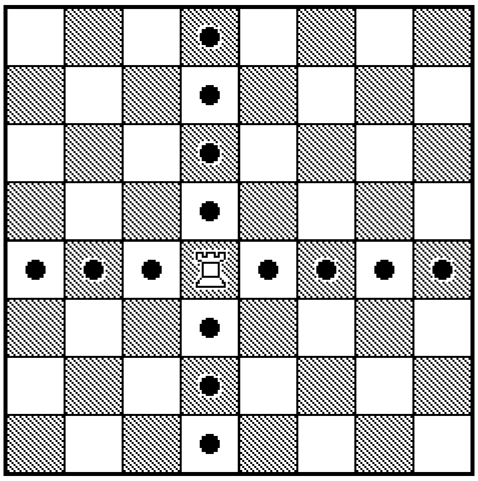


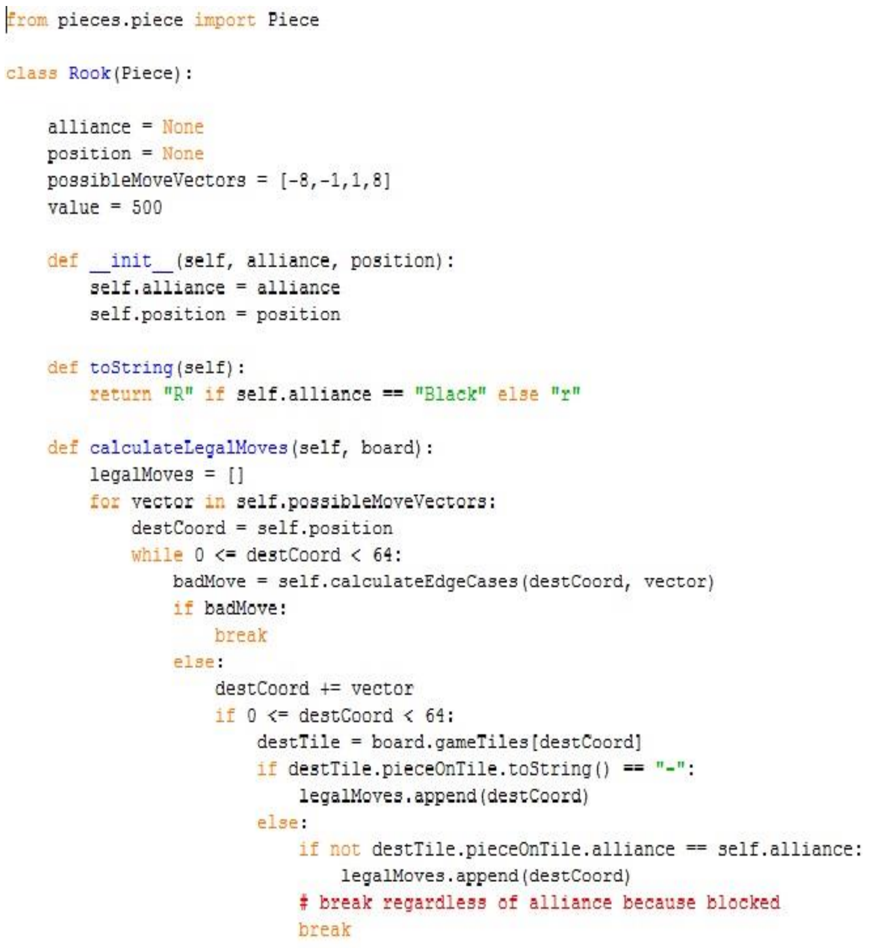
Thus, at the second row, there are eight white pawns, at the seventh row, there are eight black pawns. At the first row, from left to right, we have a: rook, knight, bishop, queen, king, bishop, knight, and rook. Note that the queens start of squares of their own color, with a dark square in each players left hand corner.

Movement of the pieces:

Rook

The rook moves in a straight line, horizontally or vertically. The rook may not jump over other pieces, that is: all squares between the square where the rook starts its move and where the rook ends its move must be empty. (As for all pieces, when the square where the rook ends his move contains a piece of the opponent, then this piece is taken. The square where the rook ends his move may not contain a piece of the player owning this rook.)

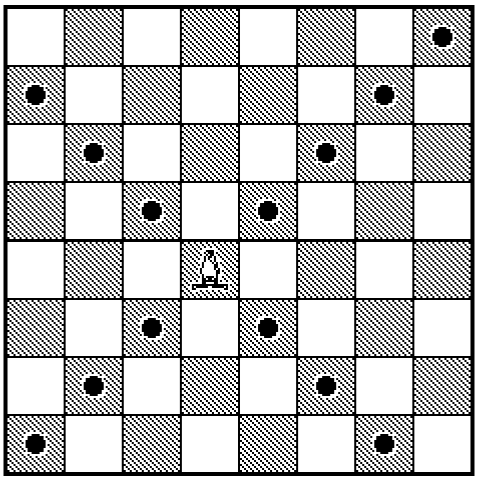


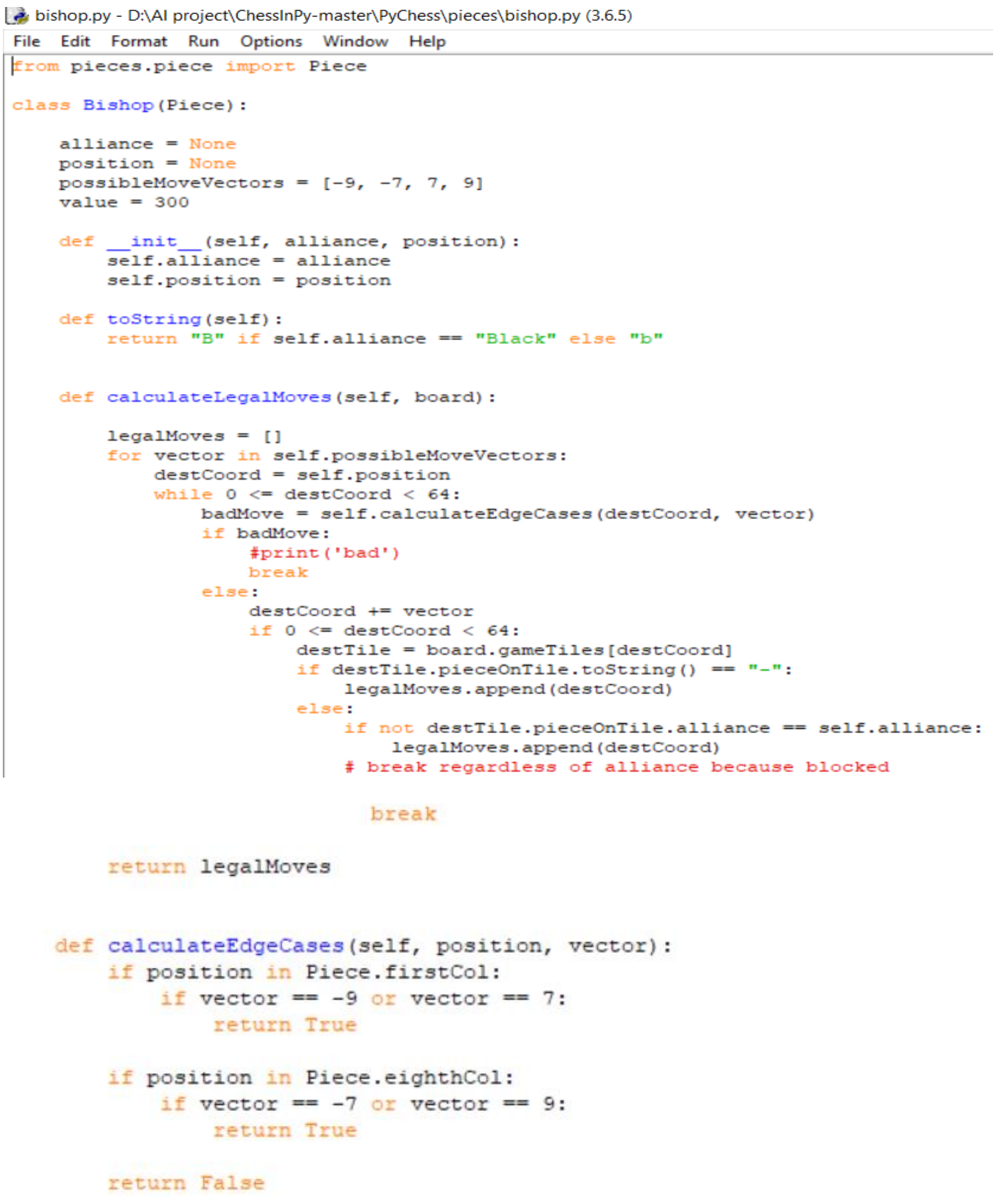


```
from pieces.piece import Piece

class Rook(Piece):

    alliance = None
    position = None
    possibleMoveVectors = [-8,-1,1,8]
    value = 500

    def __init__(self, alliance, position):
        self.alliance = alliance
        self.position = position

    def toString(self):
        return "R" if self.alliance == "Black" else "r"

    def calculateLegalMoves(self, board):
        legalMoves = []
        for vector in self.possibleMoveVectors:
            destCoord = self.position
            while 0 <= destCoord < 64:
                badMove = self.calculateEdgeCases(destCoord, vector)
                if badMove:
                    break
                else:
                    destCoord += vector
                    if 0 <= destCoord < 64:
                        destTile = board.gameTiles[destCoord]
                        if destTile.pieceOnTile.toString() == "-":
                            legalMoves.append(destCoord)
                        else:
                            if not destTile.pieceOnTile.alliance == self.alliance:
                                legalMoves.append(destCoord)
                            # break regardless of alliance because blocked
                            break
```

Bishop

The bishop moves in a straight diagonal line. The bishop may also not jump over other pieces.

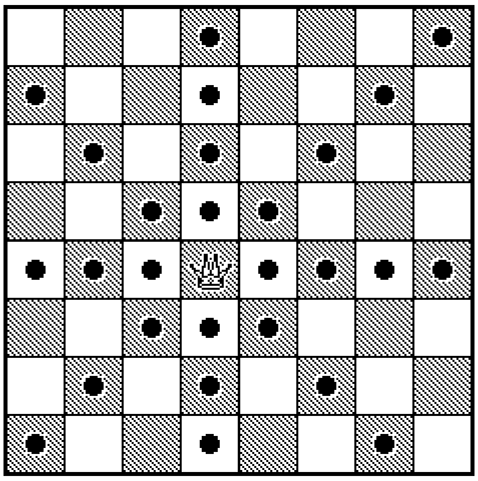


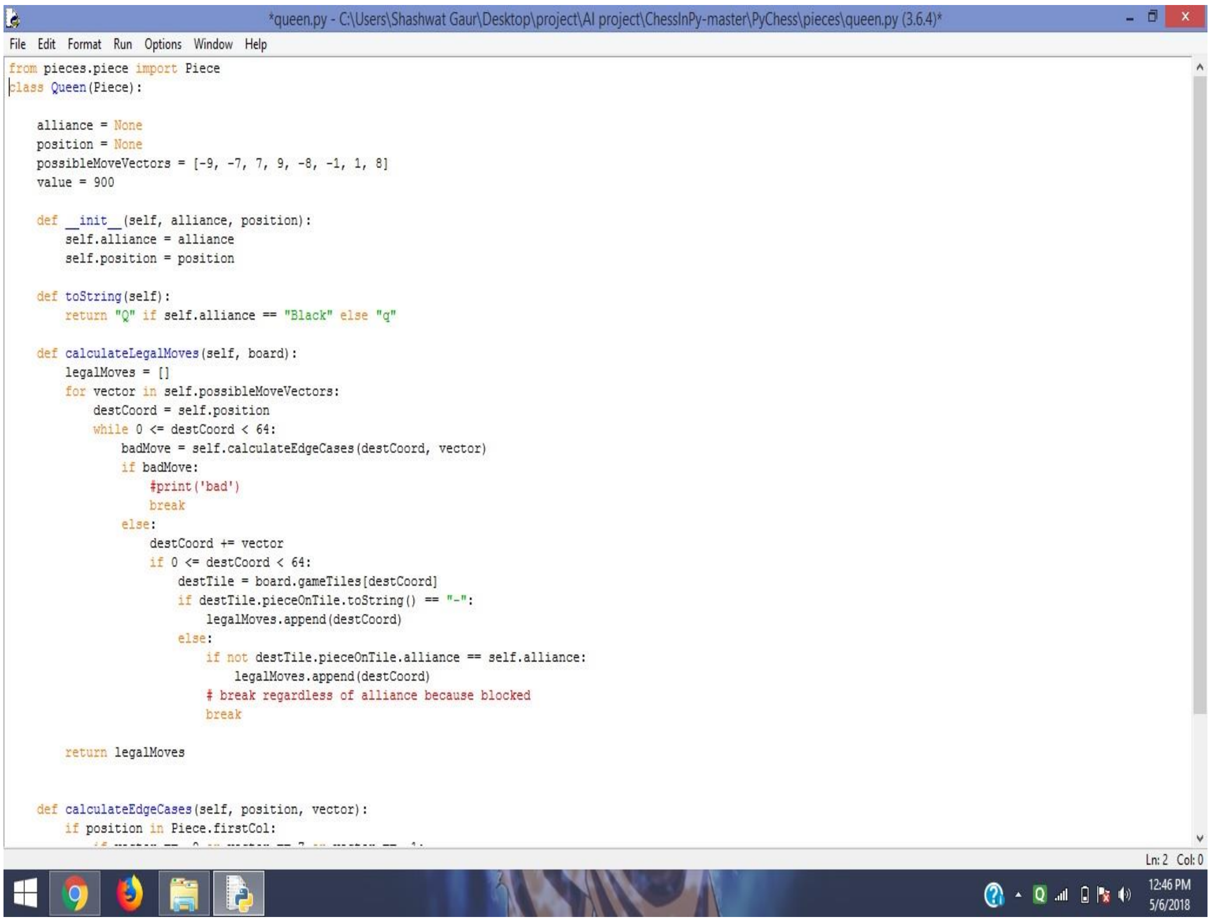

bishop.py - D:\AI project\ChessInPy-master\PyChess\pieces\bishop.py (3.6.5)
File Edit Format Run Options Window Help

```
from pieces.piece import Piece

class Bishop(Piece):

    alliance = None
    position = None
    possibleMoveVectors = [-9, -7, 7, 9]
    value = 300

    def __init__(self, alliance, position):
        self.alliance = alliance
        self.position = position

    def toString(self):
        return "B" if self.alliance == "Black" else "b"


    def calculateLegalMoves(self, board):

        legalMoves = []
        for vector in self.possibleMoveVectors:
            destCoord = self.position
            while 0 <= destCoord < 64:
                badMove = self.calculateEdgeCases(destCoord, vector)
                if badMove:
                    #print('bad')
                    break
                else:
                    destCoord += vector
                    if 0 <= destCoord < 64:
                        destTile = board.gameTiles[destCoord]
                        if destTile.pieceOnTile.toString() == "-":
                            legalMoves.append(destCoord)
                        else:
                            if not destTile.pieceOnTile.alliance == self.alliance:
                                legalMoves.append(destCoord)
                            # break regardless of alliance because blocked

                            break

        return legalMoves


    def calculateEdgeCases(self, position, vector):
        if position in Piece.firstCol:
            if vector == -9 or vector == 7:
                return True

        if position in Piece.eighthCol:
            if vector == -7 or vector == 9:
                return True

        return False
```

Queen

The queen has the *combined* moves of the rook and the bishop, i.e., the queen may move in any straight line, horizontal, vertical, or diagonal.

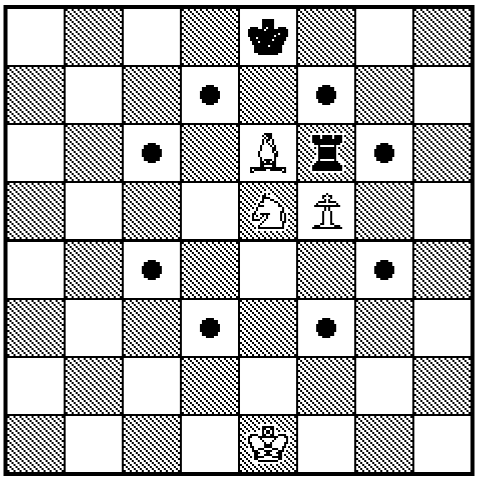


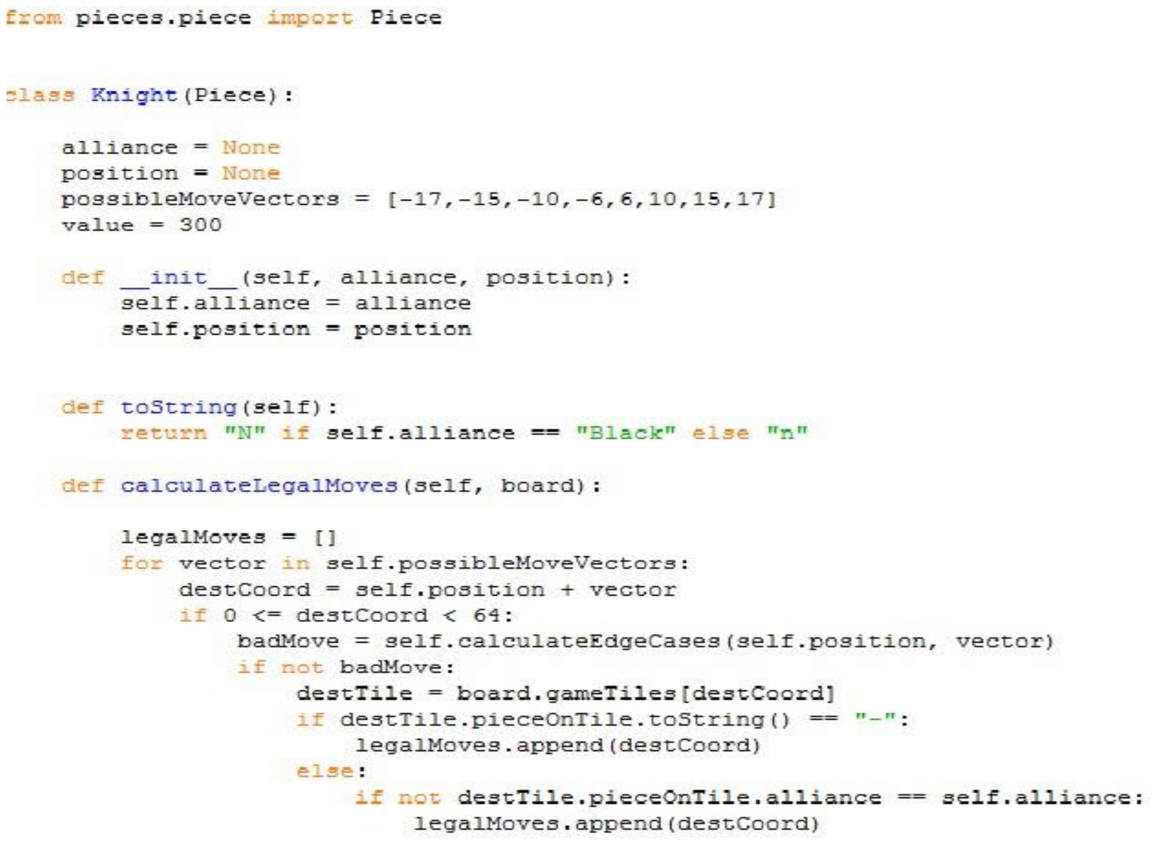

queen.py - C:\Users\Shashwat Gaur\Desktop\project\AI project\ChessInPy-master\PyChess\pieces\queen.py (3.6.4)
File Edit Format Run Options Window Help
Ln: 2 Col: 0
12:46 PM
5/6/2018

```
from pieces.piece import Piece
class Queen(Piece):

    alliance = None
    position = None
    possibleMoveVectors = [-9, -7, 7, 9, -8, -1, 1, 8]
    value = 900

    def __init__(self, alliance, position):
        self.alliance = alliance
        self.position = position

    def toString(self):
        return "Q" if self.alliance == "Black" else "q"

    def calculateLegalMoves(self, board):
        legalMoves = []
        for vector in self.possibleMoveVectors:
            destCoord = self.position
            while 0 <= destCoord < 64:
                badMove = self.calculateEdgeCases(destCoord, vector)
                if badMove:
                    #print('bad')
                    break
                else:
                    destCoord += vector
                    if 0 <= destCoord < 64:
                        destTile = board.gameTiles[destCoord]
                        if destTile.pieceOnTile.toString() == "-":
                            legalMoves.append(destCoord)
                        else:
                            if not destTile.pieceOnTile.alliance == self.alliance:
                                legalMoves.append(destCoord)
                            # break regardless of alliance because blocked
                            break

        return legalMoves


    def calculateEdgeCases(self, position, vector):
        if position in Piece.firstCol:
```

Knight

The knight makes a move that consists of first one step in a horizontal or vertical direction, and then one step diagonally in an outward direction. The knight *jumps*: it is allowed that the first square that the knight passes over is occupied by an arbitrary piece. For instance, white can start the game by moving his knight from b1 to c3. The piece that is jumped over is further not affected by the knight: as usual, a knight takes a piece of the opponent by moving to the square that contains that piece.



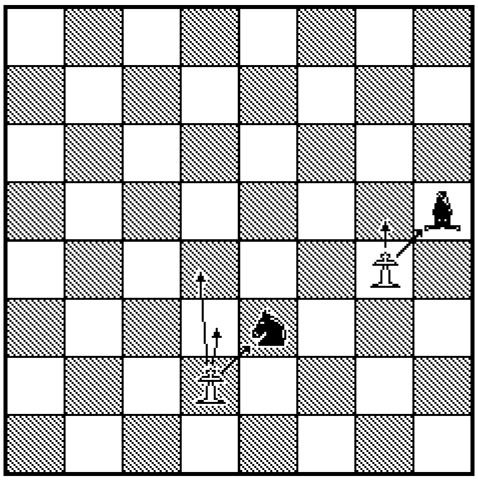


```
from pieces.piece import Piece


class Knight(Piece):

    alliance = None
    position = None
    possibleMoveVectors = [-17,-15,-10,-6,6,10,15,17]
    value = 300

    def __init__(self, alliance, position):
        self.alliance = alliance
        self.position = position


    def toString(self):
        return "N" if self.alliance == "Black" else "n"

    def calculateLegalMoves(self, board):

        legalMoves = []
        for vector in self.possibleMoveVectors:
            destCoord = self.position + vector
            if 0 <= destCoord < 64:
                badMove = self.calculateEdgeCases(self.position, vector)
                if not badMove:
                    destTile = board.gameTiles[destCoord]
                    if destTile.pieceOnTile.toString() == "-":
                        legalMoves.append(destCoord)
                    else:
                        if not destTile.pieceOnTile.alliance == self.alliance:
                            legalMoves.append(destCoord)
        return legalMoves


    def calculateEdgeCases(self, position, vector):
        if position in Piece.firstCol:
            if vector == -17 or vector == -10 or vector == 6 or vector == 15:
                return True

        if position in Piece.secondCol:
            if vector == -10 or vector == 6:
                return True

        if position in Piece.seventhCol:
            if vector == -6 or vector == 10:
                return True

        if position in Piece.eighthCol:
            if vector == -15 or vector == -6 or vector == 10 or vector == 17:
                return True

        return False
```

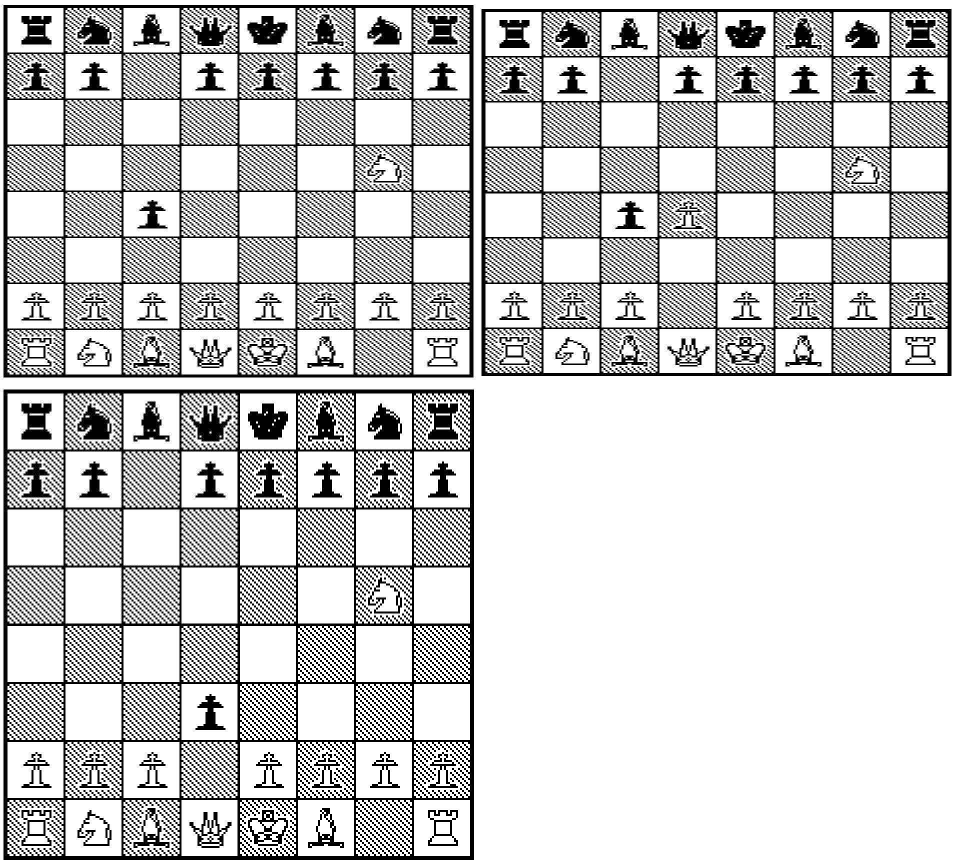
Pawn

The pawn moves differently regarding whether it moves to an empty square or whether it takes a piece of the opponent. When a pawn does not take, it moves one square straight forward. When this pawn has not moved at all, i.e., the pawn is still at the second row (from the owning players view), the pawn may make a double step straight forward. For instance, a white pawn on d2 can be moved to d4.

When taking, the pawn goes one square diagonally forward.

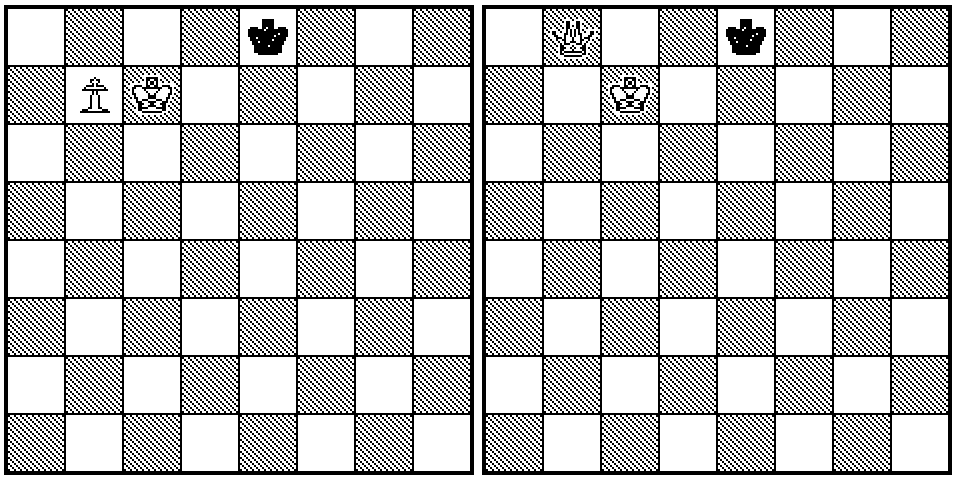


There is one special rule, called *taking en-passant*. When a pawn makes a double step from the second row to the fourth row, and there is an enemy pawn on an adjacent square on the fourth row, then this enemy pawn inthe next move may move diagonally to the square that was passed over by the double-stepping pawn, which is on the third row. In this same move, the double-stepping pawn is taken. This taking en-passant must be done directly: if the player who could take en-passant does not do this in the first move after the double step, this pawn cannot be taken anymore by an en-passant move.



A double pawn step, and a following en-passant capture

Pawns that reach the last row of the board *promote*. When a player moves a pawn to the last row of the board, he replaces the pawn by a queen, rook, knight, or bishop (of the same color). Usually, players will promote the pawn to a queen, but the other types of pieces are also allowed. (It is not required that the pawn is promoted to a piece taken. Thus, it is for instance possible that a player has at a certain moment two queens.)



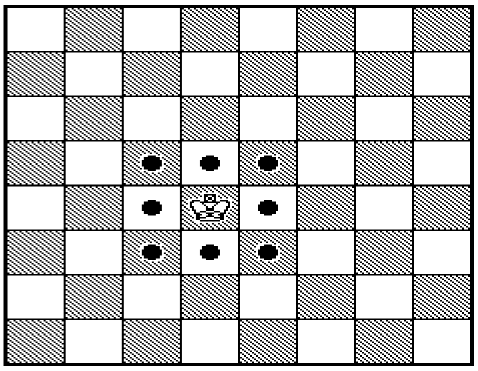
Before and after a promotion

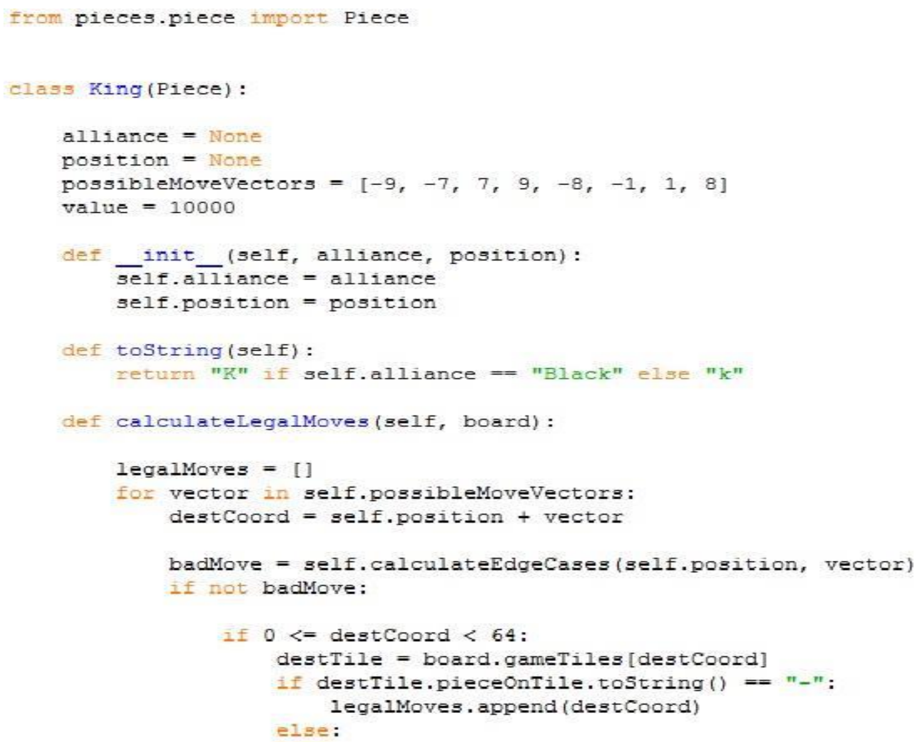
```
from pieces.piece import Piece


class Pawn(Piece):

    alliance = None
    position = None
    possibleMoveVectors = [7,9,8,16]
    allianceMultiple = None
    firstMove = True
    value = 100

    def __init__(self, alliance, position):
        self.alliance = alliance
        self.position = position
        if self.alliance == "Black":
            self.allianceMultiple = 1
        else:
            self.allianceMultiple = -1

    def toString(self):
        return "P" if self.alliance == "Black" else "p"

    def calculateLegalMoves(self, board):

        legalMoves = []
        #print(self.possibleMoveVectors)
        #print(self.allianceMultiple)

        for vector in self.possibleMoveVectors:

            destCoord = self.position + (vector*self.allianceMultiple)
```

King

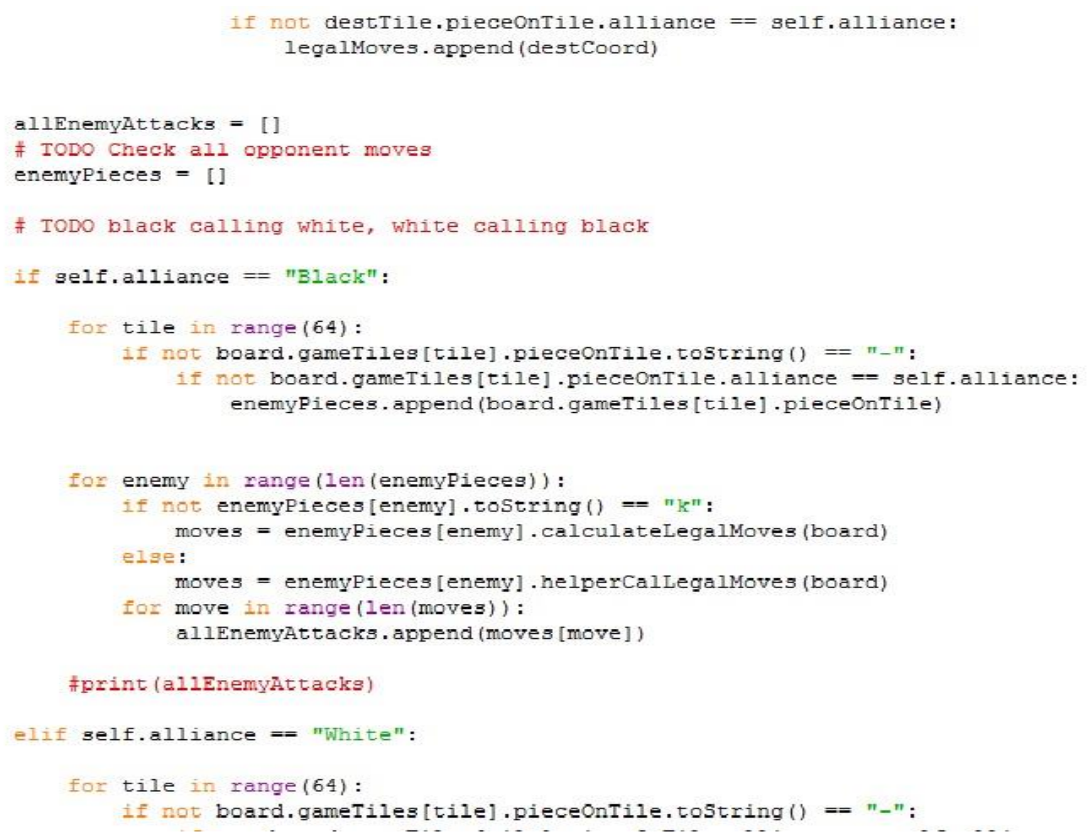
The king moves one square in any direction, horizontally, vertically, or diagonally. There is one special type of move, made by a king and rook simultaneously, called *castling*: see below.

The king is the most important piece of the game, and moves must be made in such a way that the king is never in check: see below.

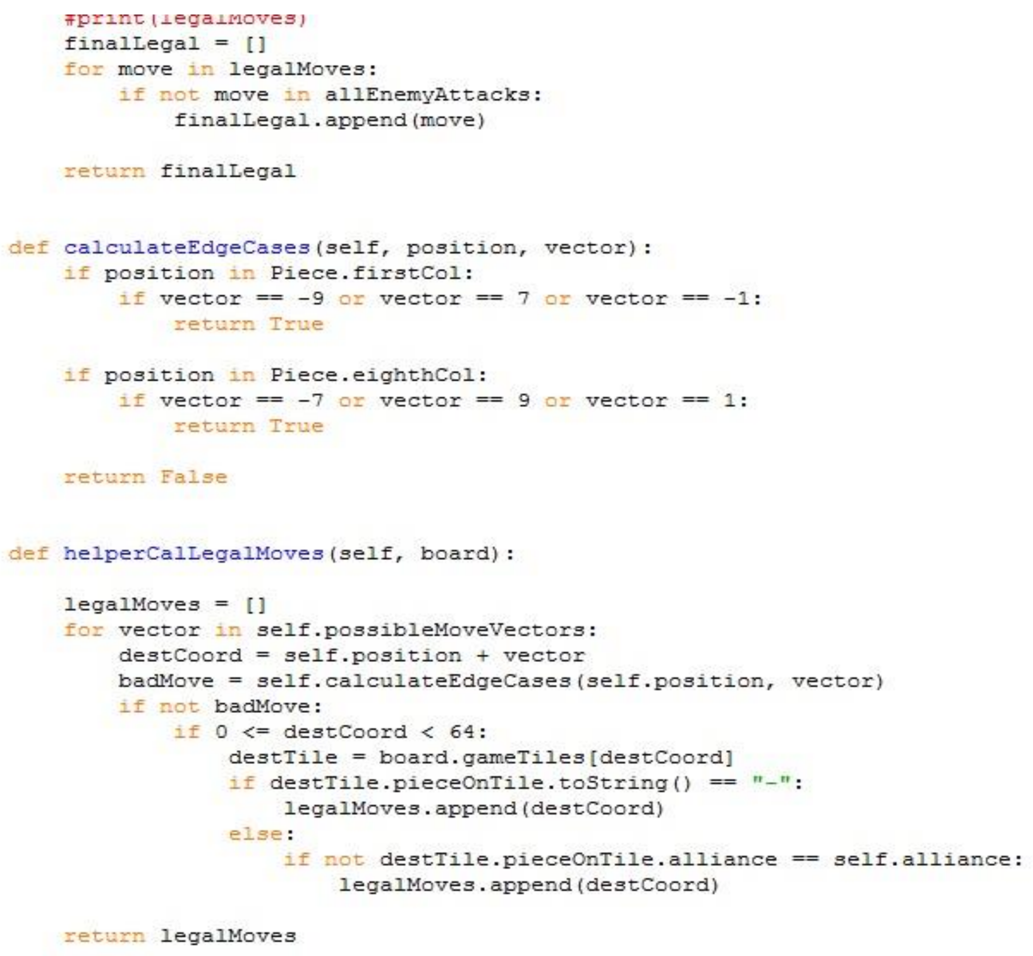




```
from pieces.piece import Piece


class King(Piece):

    alliance = None
    position = None
    possibleMoveVectors = [-9, -7, 7, 9, -8, -1, 1, 8]
    value = 10000

    def __init__(self, alliance, position):
        self.alliance = alliance
        self.position = position

    def toString(self):
        return "K" if self.alliance == "Black" else "k"

    def calculateLegalMoves(self, board):

        legalMoves = []
        for vector in self.possibleMoveVectors:
            destCoord = self.position + vector

            badMove = self.calculateEdgeCases(self.position, vector)
            if not badMove:

                if 0 <= destCoord < 64:
                    destTile = board.gameTiles[destCoord]
                    if destTile.pieceOnTile.toString() == "-":
                        legalMoves.append(destCoord)
                    else:
```



```
                if not destTile.pieceOnTile.alliance == self.alliance:
                    legalMoves.append(destCoord)


allEnemyAttacks = []
# TODO Check all opponent moves
enemyPieces = []

# TODO black calling white, white calling black

if self.alliance == "Black":

    for tile in range(64):
        if not board.gameTiles[tile].pieceOnTile.toString() == "-":
            if not board.gameTiles[tile].pieceOnTile.alliance == self.alliance:
                enemyPieces.append(board.gameTiles[tile].pieceOnTile)


    for enemy in range(len(enemyPieces)):
        if not enemyPieces[enemy].toString() == "k":
            moves = enemyPieces[enemy].calculateLegalMoves(board)
        else:
            moves = enemyPieces[enemy].helperCalLegalMoves(board)
        for move in range(len(moves)):
            allEnemyAttacks.append(moves[move])

    #print(allEnemyAttacks)

elif self.alliance == "White":

    for tile in range(64):
        if not board.gameTiles[tile].pieceOnTile.toString() == "-":
            [illegible]
```



```
    #print(legalMoves)
    finalLegal = []
    for move in legalMoves:
        if not move in allEnemyAttacks:
            finalLegal.append(move)

    return finalLegal


def calculateEdgeCases(self, position, vector):
    if position in Piece.firstCol:
        if vector == -9 or vector == 7 or vector == -1:
            return True

    if position in Piece.eighthCol:
        if vector == -7 or vector == 9 or vector == 1:
            return True

    return False


def helperCalLegalMoves(self, board):

    legalMoves = []
    for vector in self.possibleMoveVectors:
        destCoord = self.position + vector
        badMove = self.calculateEdgeCases(self.position, vector)
        if not badMove:
            if 0 <= destCoord < 64:
                destTile = board.gameTiles[destCoord]
                if destTile.pieceOnTile.toString() == "-":
                    legalMoves.append(destCoord)
                else:
                    if not destTile.pieceOnTile.alliance == self.alliance:
                        legalMoves.append(destCoord)

    return legalMoves
```

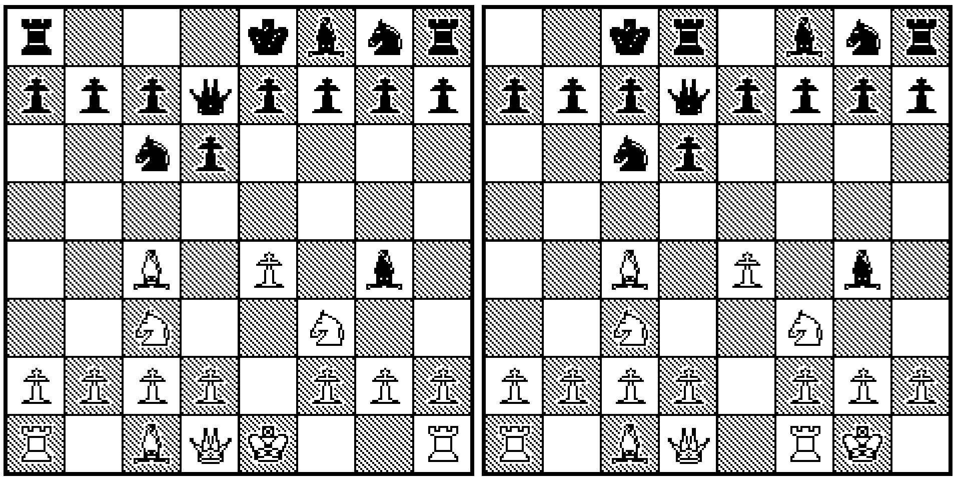
Castling

Under certain, special rules, a king and rook can move simultaneously in a *castling* move.

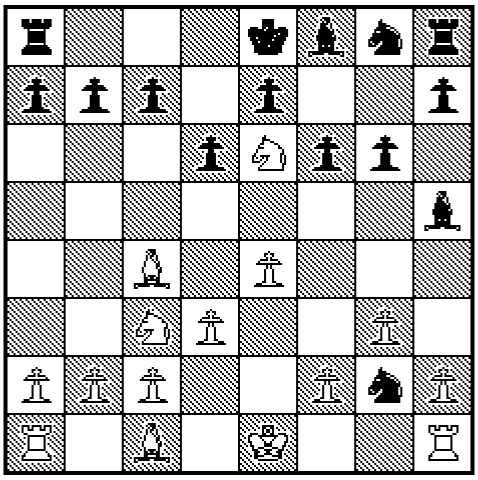
The following conditions must be met:

- The king that makes the castling move has not yet moved in the game.
- The rook that makes the castling move has not yet moved in the game.
- The king is not in check.
- The king does not move over a square that is attacked by an enemy piece during the castling move, i.e., when castling, there may not be an enemy piece that can move (in case of pawns: by diagonal movement) to a square that is moved over by the king.
- The king does not move to a square that is attacked by an enemy piece during the castling move, i.e., you may not castle and end the move with the king in check.
- All squares between the rook and king before the castling move are empty.
- The King and rook must occupy the same rank (or row).

When castling, the king moves two squares towards the rook, and the rook moves over the king to the next square, i.e., black's king on e8 and rook on a8 move to: king c8, rook d8 (*long castling*), white's king on e1 and rook on h1 move to: king g1, rook f1 (*short castling*).



position before and after castling: white short, and black long



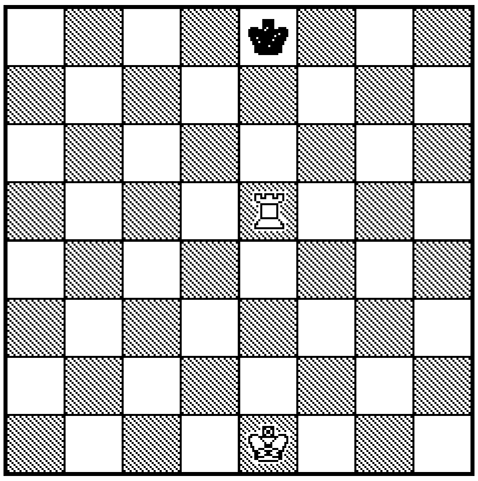
Neither white nor black may castle: white is in check, and the black king may not move over d8

Check, mate, and stalemate

Check

When the king of a player can be taken by a piece of the opponent, one says that the king is *in check*. For instance, the white player moves his rook to a position such that it attacks the black king, i.e., if black doesn't do anything about it, the rook could take the black king in the next move: we say that the white rook *gives check.* It is considered good manners to say *check* when one checks ones opponent.

It is not allowed to make a move, such that ones king is in check after the move. If a player accidently tries to make such a move, he must take the move back and make another move (following the rules that one must move with the piece one has touched, see below.)

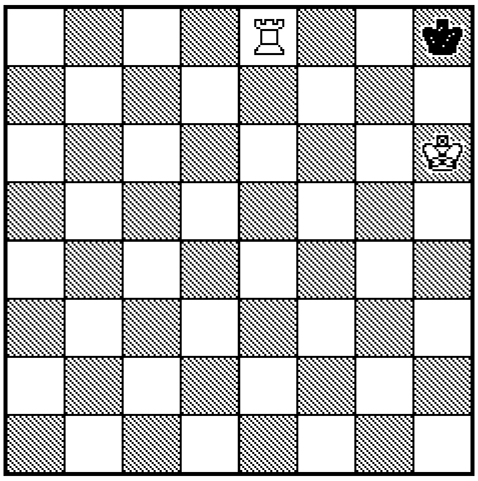


Mate

When a player is in check, and he cannot make a move such that after the move, the king is not in check, then he is *mated*. The player that is mated lost the game, and the player that mated him won the game.

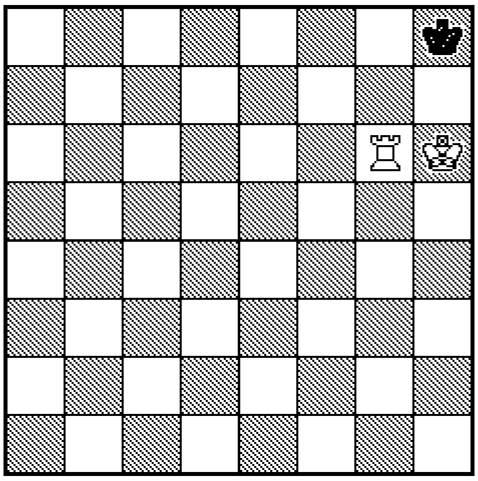
Note that there are three different possible ways to remove a check:

1. Move the king away to a square where he is not in check.
2. Take the piece that gives the check.
3. (In case of a check, given by a rook, bishop or queen:) move a piece between the checking piece and the king.



Stalemate

When a player cannot make any legal move, but he is not in check, then the player is said to be *stalemated*. In a case of a stalemate, the game is a draw.



When black must move, the game is a stalemate

Other rules

Resign and draw proposals

A player can resign the game, which means that he has lost and his opponent has won.

After making a move, a player can propose a draw: his opponent can accept the proposal (in which case the game ends and is a draw) or refuse the proposal (in which case the game continues).

Repetition of moves

If the same position with the same player to move is repeated three times in the game, the player to move can claim a draw. (When the right to make a certain castling move is lost by one of the players between positions, then the positions are considered to be different. For the fine points of this rule.

One case where the repetition of move occurs is when a player continues to give check forever.

50 moves rules

If there are have been 50 consecutive moves of white and of black without

- any piece taken
- any pawn move

then a player can claim a draw.

Touching pieces

When a player touches one of his own pieces, he must, if possible, make a legal move with this piece. When a player touches a piece of the opponent, he must, if possible, take this piece.

Chess clocks and time

Often, players play the game with chess clocks. These clocks count the time that each player separately takes for making his own moves. Additional rules are then used, saying how many (possibly all) moves must be made before a player has used a certain time for his moves

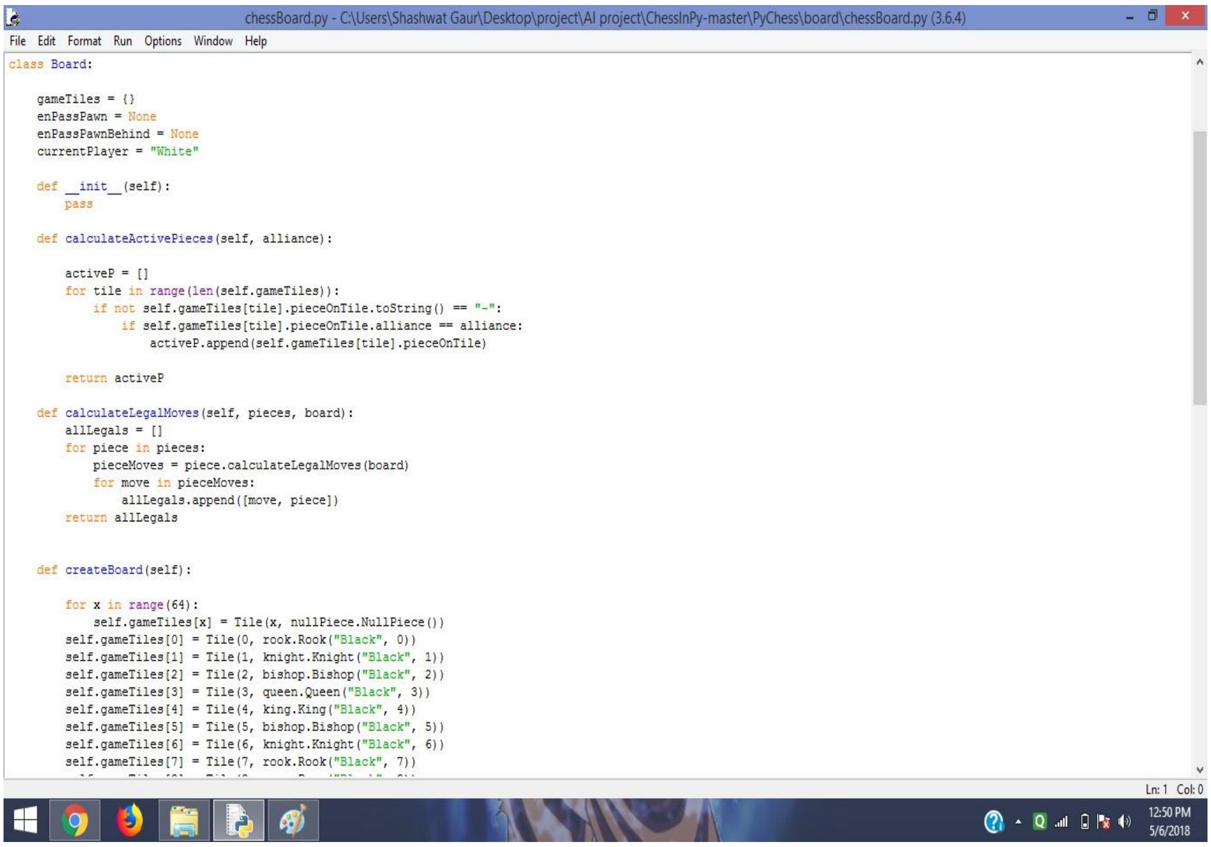
Other rules

There are other rules, telling what should happen in special occasions, like that players started the game with a wrong setup, etc. These are not so important for friendly games; for details, again see the official rules of chess.

Chessboard

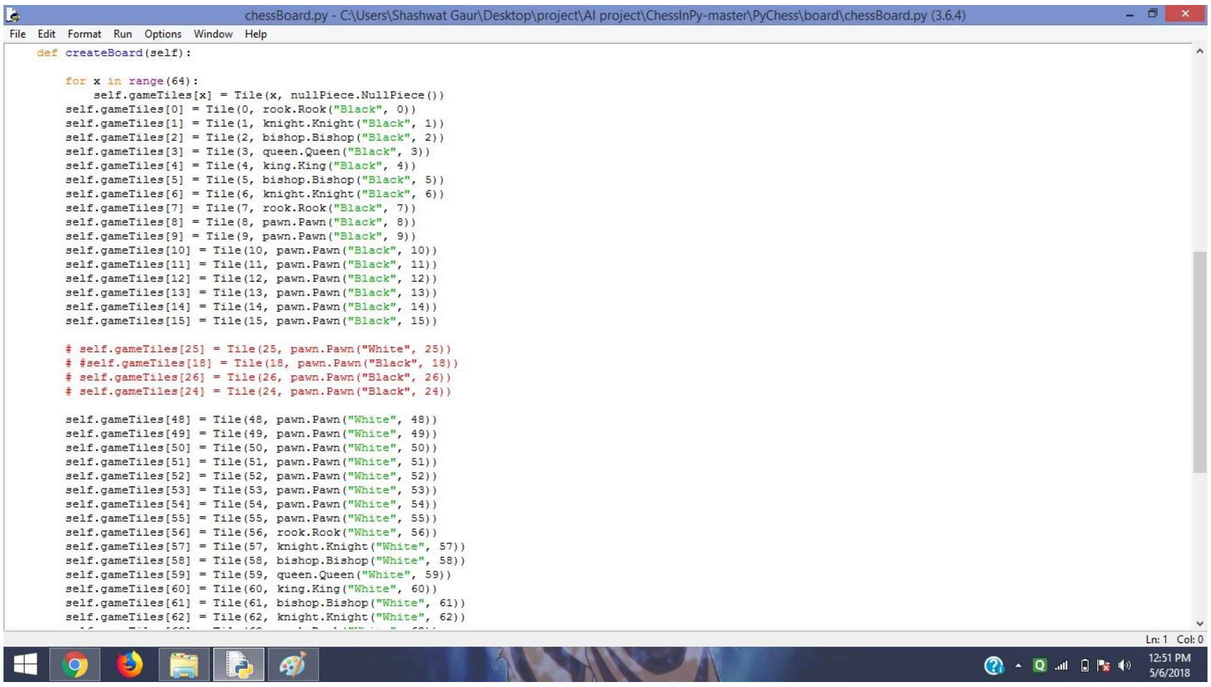
A **chessboard** is the type of checkerboard used in the board game chess, consisting of 64 squares (eight rows and eight columns). The squares are arranged in two alternating colors (light and dark). Wooden boards may use naturally light and dark brown woods, while plastic and vinyl boards often use brown or green for the dark squares and shades such as buff or cream for the light squares. Materials vary widely; while wooden boards are generally used in high-level games; vinyl, plastic, and cardboard are common for low-level and informal play. Decorative glass and marble boards are available but rarely accepted for games rated by national or international chess federations.

Chessboard Snap code:


chessBoard.py - C:\Users\Shashwat Gaur\Desktop\project\AI project\ChessInPy-master\PyChess\board\chessBoard.py (3.6.4)

```python
class Board:

    gameTiles = {}
    enPassPawn = None
    enPassPawnBehind = None
    currentPlayer = "White"

    def __init__(self):
        pass

    def calculateActivePieces(self, alliance):

        activeP = []
        for tile in range(len(self.gameTiles)):
            if not self.gameTiles[tile].pieceOnTile.toString() == "-":
                if self.gameTiles[tile].pieceOnTile.alliance == alliance:
                    activeP.append(self.gameTiles[tile].pieceOnTile)

        return activeP

    def calculateLegalMoves(self, pieces, board):
        allLegals = []
        for piece in pieces:
            pieceMoves = piece.calculateLegalMoves(board)
            for move in pieceMoves:
                allLegals.append([move, piece])
        return allLegals


    def createBoard(self):

        for x in range(64):
            self.gameTiles[x] = Tile(x, nullPiece.NullPiece())
        self.gameTiles[0] = Tile(0, rook.Rook("Black", 0))
        self.gameTiles[1] = Tile(1, knight.Knight("Black", 1))
        self.gameTiles[2] = Tile(2, bishop.Bishop("Black", 2))
        self.gameTiles[3] = Tile(3, queen.Queen("Black", 3))
        self.gameTiles[4] = Tile(4, king.King("Black", 4))
        self.gameTiles[5] = Tile(5, bishop.Bishop("Black", 5))
        self.gameTiles[6] = Tile(6, knight.Knight("Black", 6))
        self.gameTiles[7] = Tile(7, rook.Rook("Black", 7))
```

Ln: 1 Col: 0


chessBoard.py - C:\Users\Shashwat Gaur\Desktop\project\AI project\ChessInPy-master\PyChess\board\chessBoard.py (3.6.4)

```python
    def createBoard(self):

        for x in range(64):
            self.gameTiles[x] = Tile(x, nullPiece.NullPiece())
        self.gameTiles[0] = Tile(0, rook.Rook("Black", 0))
        self.gameTiles[1] = Tile(1, knight.Knight("Black", 1))
        self.gameTiles[2] = Tile(2, bishop.Bishop("Black", 2))
        self.gameTiles[3] = Tile(3, queen.Queen("Black", 3))
        self.gameTiles[4] = Tile(4, king.King("Black", 4))
        self.gameTiles[5] = Tile(5, bishop.Bishop("Black", 5))
        self.gameTiles[6] = Tile(6, knight.Knight("Black", 6))
        self.gameTiles[7] = Tile(7, rook.Rook("Black", 7))
        self.gameTiles[8] = Tile(8, pawn.Pawn("Black", 8))
        self.gameTiles[9] = Tile(9, pawn.Pawn("Black", 9))
        self.gameTiles[10] = Tile(10, pawn.Pawn("Black", 10))
        self.gameTiles[11] = Tile(11, pawn.Pawn("Black", 11))
        self.gameTiles[12] = Tile(12, pawn.Pawn("Black", 12))
        self.gameTiles[13] = Tile(13, pawn.Pawn("Black", 13))
        self.gameTiles[14] = Tile(14, pawn.Pawn("Black", 14))
        self.gameTiles[15] = Tile(15, pawn.Pawn("Black", 15))

        # self.gameTiles[25] = Tile(25, pawn.Pawn("White", 25))
        # #self.gameTiles[18] = Tile(18, pawn.Pawn("Black", 18))
        # self.gameTiles[26] = Tile(26, pawn.Pawn("Black", 26))
        # self.gameTiles[24] = Tile(24, pawn.Pawn("Black", 24))

        self.gameTiles[48] = Tile(48, pawn.Pawn("White", 48))
        self.gameTiles[49] = Tile(49, pawn.Pawn("White", 49))
        self.gameTiles[50] = Tile(50, pawn.Pawn("White", 50))
        self.gameTiles[51] = Tile(51, pawn.Pawn("White", 51))
        self.gameTiles[52] = Tile(52, pawn.Pawn("White", 52))
        self.gameTiles[53] = Tile(53, pawn.Pawn("White", 53))
        self.gameTiles[54] = Tile(54, pawn.Pawn("White", 54))
        self.gameTiles[55] = Tile(55, pawn.Pawn("White", 55))
        self.gameTiles[56] = Tile(56, rook.Rook("White", 56))
        self.gameTiles[57] = Tile(57, knight.Knight("White", 57))
        self.gameTiles[58] = Tile(58, bishop.Bishop("White", 58))
        self.gameTiles[59] = Tile(59, queen.Queen("White", 59))
        self.gameTiles[60] = Tile(60, king.King("White", 60))
        self.gameTiles[61] = Tile(61, bishop.Bishop("White", 61))
        self.gameTiles[62] = Tile(62, knight.Knight("White", 62))
```

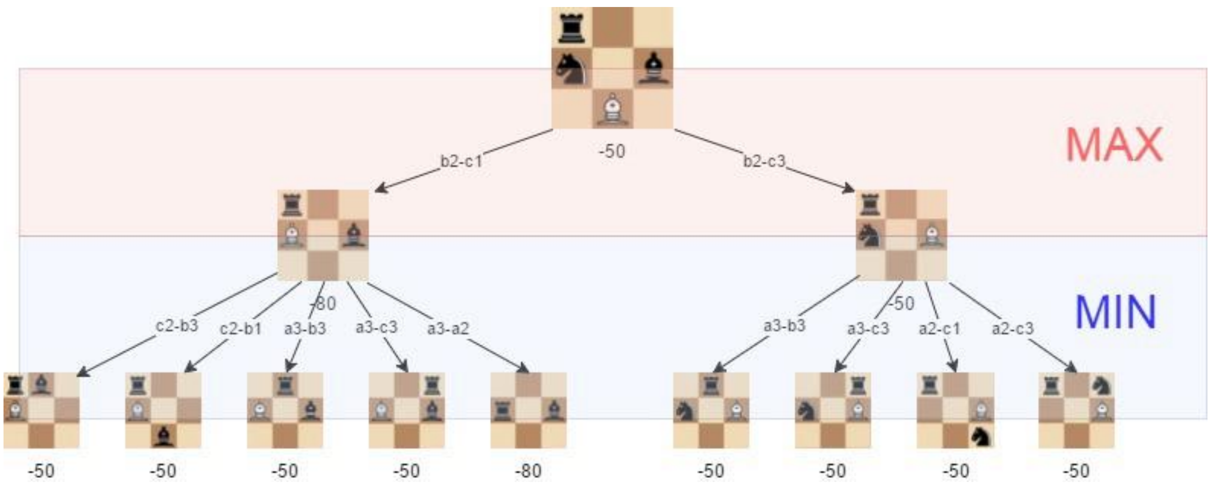
Ln: 1 Col: 0

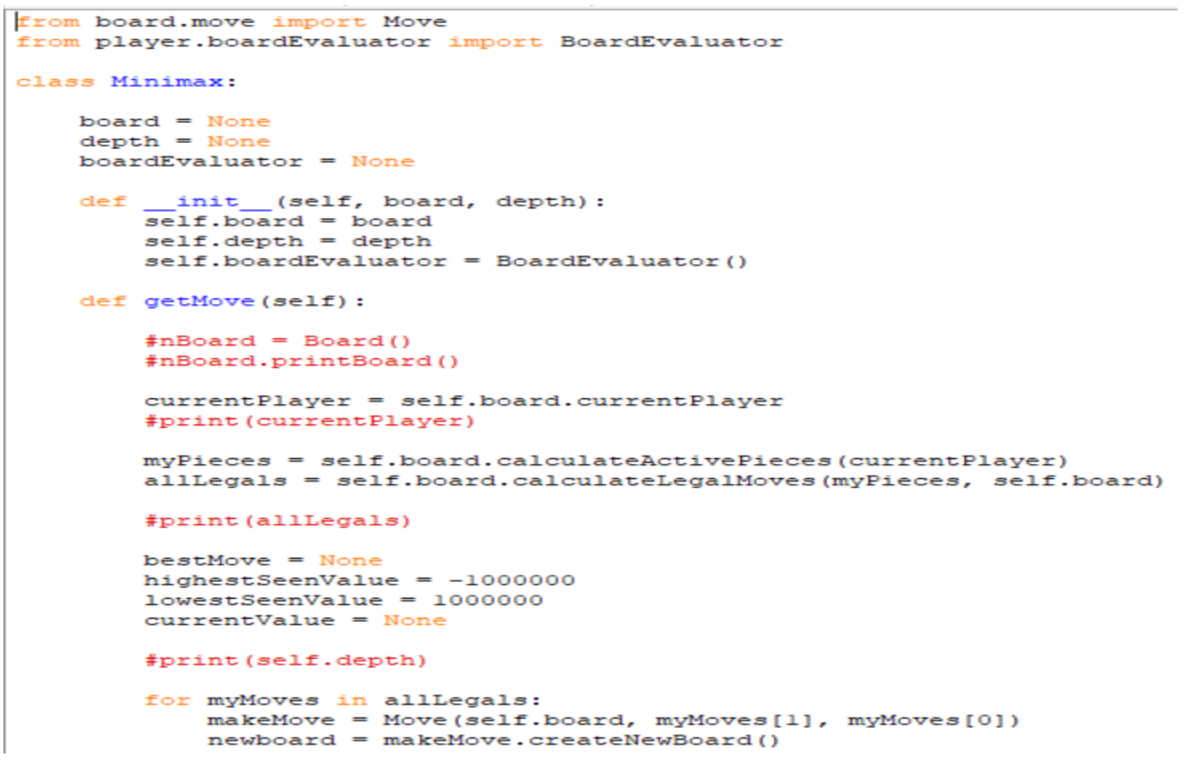
Implementation of MinMax Algorithm in Chess:

Search tree using Minimax

Next we’re going to create a search tree from which the algorithm can chose the best move. This is done by using the Minimax algorithm.In this algorithm, the recursive tree of all possible moves is explored to a given depth, and the position is evaluated at the ending “leaves” of the tree.

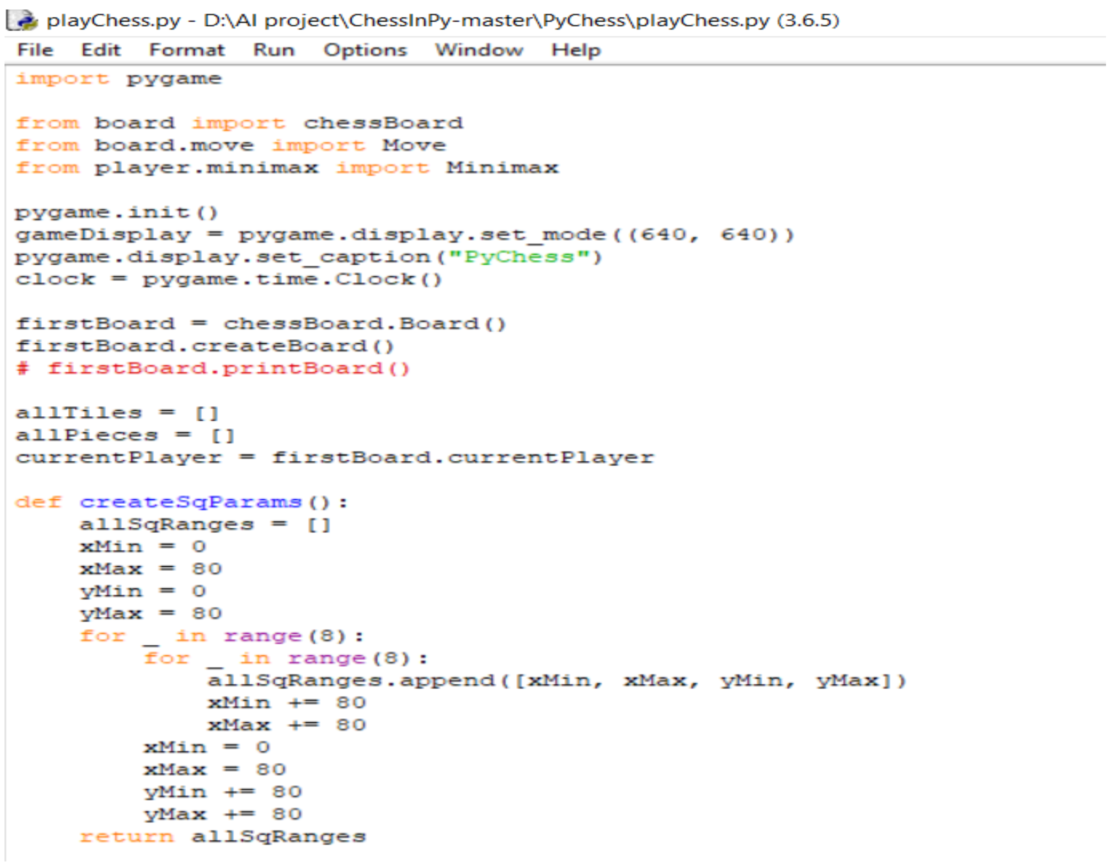
After that, we return either the smallest or the largest value of the child to the parent node, depending on whether it’s a white or black to move. (That is, we try to either minimize or maximize the outcome at each level.)





```
from board.move import Move
from player.boardEvaluator import BoardEvaluator

class Minimax:

    board = None
    depth = None
    boardEvaluator = None

    def __init__(self, board, depth):
        self.board = board
        self.depth = depth
        self.boardEvaluator = BoardEvaluator()

    def getMove(self):

        #nBoard = Board()
        #nBoard.printBoard()

        currentPlayer = self.board.currentPlayer
        #print(currentPlayer)

        myPieces = self.board.calculateActivePieces(currentPlayer)
        allLegals = self.board.calculateLegalMoves(myPieces, self.board)

        #print(allLegals)

        bestMove = None
        highestSeenValue = -1000000
        lowestSeenValue = 1000000
        currentValue = None

        #print(self.depth)

        for myMoves in allLegals:
            makeMove = Move(self.board, myMoves[1], myMoves[0])
            newboard = makeMove.createNewBoard()
            if not newboard == False:
                if currentPlayer == "White":
                    #print("Work w")
                    currentValue = self.min(newboard, self.depth)
                    #print(currentValue)
                else:
                    #print("Work b")
                    currentValue = self.max(newboard, self.depth)
                    #print(currentValue)

                if currentPlayer == "White" and currentValue > highestSeenValue:
                    highestSeenValue = currentValue
                    bestMove = newboard

                if currentPlayer == "Black" and currentValue < lowestSeenValue:
                    lowestSeenValue = currentValue
                    bestMove = newboard

        return bestMove


    def max(self, board, depth):

        #TODO checkmate/stalemate
        if depth == 0:
            #print("Hello")
            #return 99
            return self.boardEvaluator.evaluate(board, depth)
```

Snap of main code:



playChess.py - D:\AI project\ChessInPy-master\PyChess\playChess.py (3.6.5)

```
import pygame

from board import chessBoard
from board.move import Move
from player.minimax import Minimax

pygame.init()
gameDisplay = pygame.display.set_mode((640, 640))
pygame.display.set_caption("PyChess")
clock = pygame.time.Clock()

firstBoard = chessBoard.Board()
firstBoard.createBoard()
# firstBoard.printBoard()

allTiles = []
allPieces = []
currentPlayer = firstBoard.currentPlayer

def createSqParams():
    allSqRanges = []
    xMin = 0
    xMax = 80
    yMin = 0
    yMax = 80
    for _ in range(8):
        for _ in range(8):
            allSqRanges.append([xMin, xMax, yMin, yMax])
            xMin += 80
            xMax += 80
        xMin = 0
        xMax = 80
        yMin += 80
        yMax += 80
    return allSqRanges
```

Conclusion

Our aim of the project was to implement the different algorithms of Artificial Intelligence and in the process of development of minor project we achieved that too ,We develop three popular 2 player games named TIC-TAC-TOE ,Pacman and Chess and in this process we used different gaming library such as turtle and pygames for the creation of different layout and to make it user friendly . We used algorithms like Alpha -Beta Pruning ,MinMax algorithm ,BFS ,DFS in the execution of our code. We also used Board evaluation in development of chess .The outcome of the project is that we achieved success in implementation of AI algorithms ,development of GUI based games, and gain experience of developing project in a professional manner.

References

- Artificial Intelligence-A modern Day Approach (Stuart Russel and Peter Norvig)
- INTRODUCTION TO ARTIFICIAL INTELLIGENCE (Gunjan Goswami)
- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/
- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/
- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/
- https://docs.python.org/2/library/turtle.html
- https://en.wikipedia.org/wiki/Pygame
- The Pac-Man Dossier
- https://my.xfinity.com/~jpittman2/pacman/pacmandossier.html

Personal Details:

Rachit Shrivastava (151361)

Contact:7697373866

Email id: rachitshrivastava 90@gmail.com

CGPA : 6.2

Shashwat Gaur (151386)

Contact :8989468071

Email id: shashwatgaur23@gmail.com

CGPA:6.7

Shashwat Singh (151387)

Contact: 7240915591

Email id: shaswat17it@gmail.com

CGPA :6.3