X’s and O’s

Creating a Program to Play Tic-Tac-Toe, with Artificial Opponents

Daniel Avalos
Austin Community College

August 2018

This Report details the construction of a Tic-Tac-Toe game designed in the programming language Python 3, with playable artificial opponents. Contents include discussion of the method of board game analysis used, the theory behind the game, in depth analysis of code used, and concludes with a summary and future roadmap for the code.

Figures and Charts

Introduction

Tic Tac Toe is a children's game played on a 3x3 grid between two players. Players alternate placing tokens 'X' and 'O' on the board with the final goal of setting three of their tokens in a row horizontally, vertically, or diagonally. If the game board fills without a victory state, the game ties with no winner.

Tic-Tac-Toe is an elegantly simple game and lends itself well to computer translation, thanks to its grid layout and concrete rules. Moves are set with no removals or nebulous placement, turns are definite, and victories or tied games (terminal states) are easily identified.

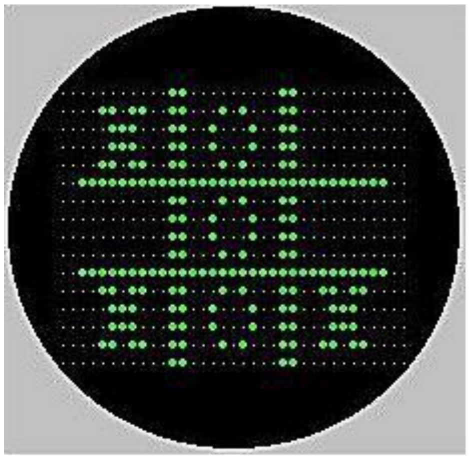


Figure 0.1: The Original Tic-Tac-Toe Computer Game (Cohen, 2018)

Developing software solutions to play Tic-Tac-Toe is not a novel idea; one of the first computer programs with an artificial opponent ever developed was the game OXO, by Alexander Douglas, CBE, at Oxford University in 1952 (Cohen, 2018). The game was a digital rendition of Tic Tac Toe, played against an artificial opponent on the tape-fed *Electronic Delay Storage Automatic Calculator (EDSAC)* and displayed on a cathode-ray display, a historic achievement of programming and implementation of artificial intelligence. Numerous books and articles have been created studying the original code, possible enhancements, and deeper algorithms involved in creating a digital version of this game. Today, creating a Tic Tac Toe program with artificial players is seen as an exercise in programming, data processing, game theory, and software development. This report will focus on the creation of a program with a heavy focus on modularity, where programming elements are built separate from each other and operate reliably, using the versatile programming language Python 3. This language was chosen due to its readability, compactness, and simplicity in implementation.

The body of this report is divided into six sections. The first focuses on the theory and analysis of the game elements and will be used to establish the processing method used to parse the board and handle the game. The second section is dedicated to the Table, the centralized storage for game elements. This explains the construction of the Board and the tools it uses. The third section describes the Player modules, standardized sections of code uses to populate the game with human and artificial players. The fourth section details the working of the Referee, gamemaster of the code. As the orchestrator, the Referee contains the most crucial code; all other elements of code must be constructed before it can operate. The fifth section shows the code in action and how the reader can recreate code execution for their own analysis. The final section summarizes the code, lessons learned, and plans for future iterations of the game.

Above all, this report seeks to maintain readability and clarity of language, without reliance on obfuscating language. To paraphrase famed computer scientist Alan Turing, if one can clearly explain how a calculation is to be done, in plain English, then it is always possible to create a program to do that calculation (Turing, 1953)

I. Defining Game Elements

Processing Methods

To translate the game to code requires concrete rules established for how the computer will process the game board and moves placed. There are two methods of processing available for the code:

(1) Raw processing, where every square is unique, and moves are calculated for each position

(2) Symmetrical processing, where patterns are recognized, and board states are treated as similar layouts.

These methods treat the board with different levels of uniqueness, a valid consideration for processing the game. If two games played with different moves create the same final board, are they functionally identical? Does it matter if 'X' played corner 1 or 3 if both ended in a tie game? Should we acknowledge the board's symmetry, to save effort? *Figure 1.1* maps the optimal strategies, with board symmetry accounted for. Shouldn't the code implement an advanced system that recognizes patterns into our AI?

Defining Our Calculations

If taking a raw processing approach there are 255,168 unique games of Tic Tac Toe to consider. Were we to use a symmetrical processing, that number reduces to 26,830 games. (*Schaefer, 2002*). While instinct may say the smaller number is the better goal, creating and implementing such a recognition system exceeds the scope of this project, and wastes more effort than resources saved. To a computer, the difference 26,000 and 250,000 is a matter of a few seconds. For this small-scale project, the code will implement a raw processing method.

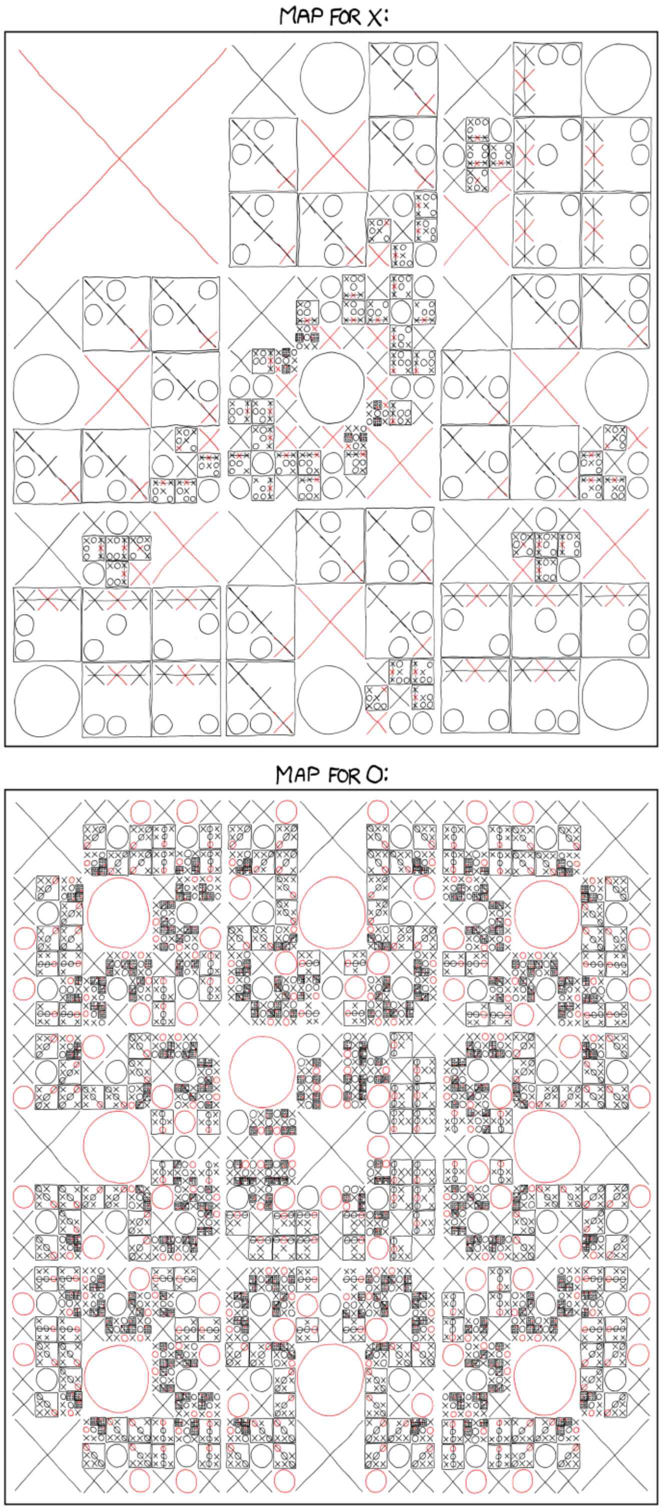


Figure 1.1: Parallel Moves (Munroe, 2010)

Game States

To build artificial players that can accurately analyze and play the game requires a substantial understanding of the game's theory. Mapping terminal states and understanding a player's options is essential in any game play analysis.

Counting States

Figure 1.2 is a short table compiled from Dr. Schaefer's 2002 article *Tic-Tac-Toe*, in which he discussed the number of possible games on the 3x3 board. On move 1, Player 1 has 9 possible moves, creating 9 possible board states. Player 2's 8 possible responses multiply the number of board states to 72, and so on through the game. Terminal conditions only become possible by move 5, the earliest possible Player 1 victory. Knowing the soonest terminal state will be crucial for later move analysis by our artificial players.

Move	Total states	Terminal States
1	9	-
2	72	-
3	504	-
4	3,024	-
5	15,120	1,440
6	54,720	5,328
7	148,176	47,952
8	200,448	72,576
9	127,872	127,872
Total	-	**255,168**

Figure 1.2: Number of Tic Tac Toe Board Configurations
Data Source: Schaefer 2002

Figure 1.3 gives a better visualization of the total number of terminal states as the game progresses, if losing some of the detail above. It becomes apparejt just how quickly the game grows in complexity after the 5th move, and how essential proper midgame ability is. Complexity grows and peaks by move 8 immediately before the 9th move terminates all possible states. The sharp decline in possible states compared to the previous move suggests greater significance in move 8. While both sets of data detail game states, a deeper analysis of individual states is required to fully understand late game states.

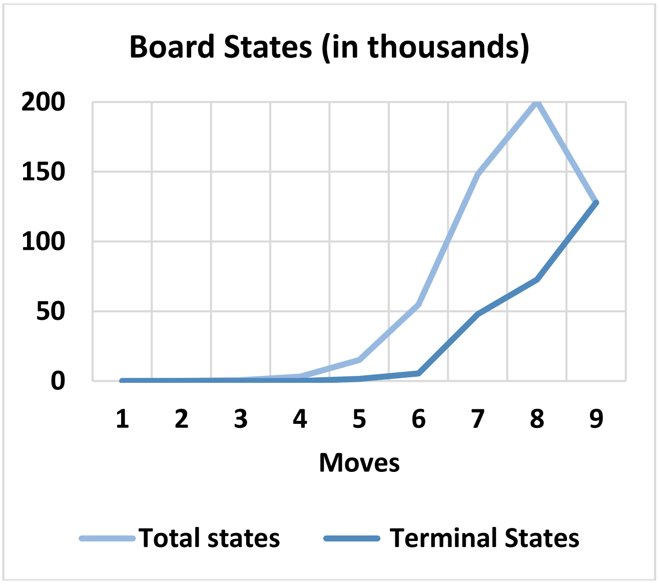


Figure 1.3: States of the Board Through Game Progression
Data Source: Schaefer 2002

Breaking Down the States

Dr Jesper Juul's document *All Games of Tic Tac Toe* provided a printout of all possible terminal states. Schaefer's breakdown is essential for tracking terminal states overall, but Juul's construction of all 255,168 thousand terminal Tic-Tac-Toe boards allows for the deeper analysis required. From the start an easy overview is possible as all terminal states can be counted.

1) Player 1 wins 131,184 games

2) Player 2 wins 77,904 games

3) 46,080 games end in a tie

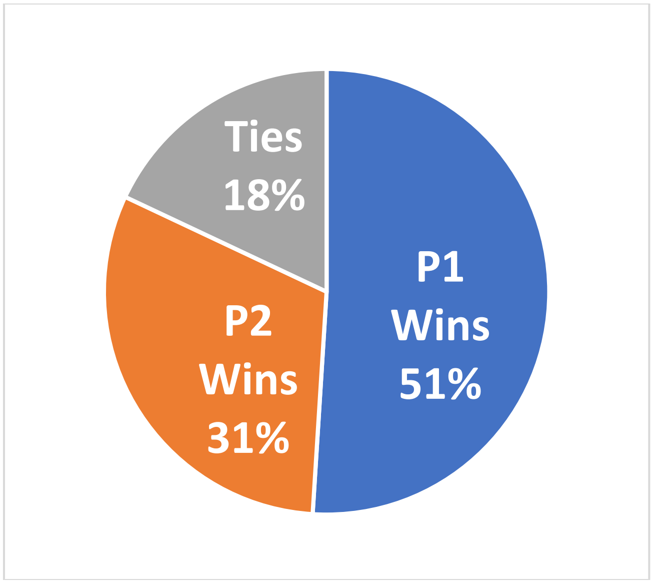
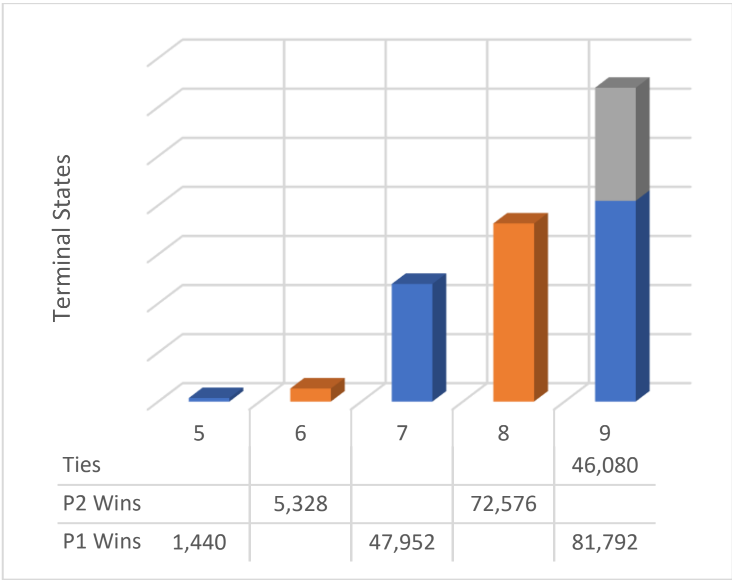


Figure 1.4: Breakdown of Total Terminal States (Data: Juul, 2003)

Solid indications of the advantages granted to Player 1, but additional parsing of all 255,168 games per move further emphasizes both the imbalanced advantage and significance of late game play.

Running a data crunching script on all quarter million games allowed sorting terminal states into separate categories based on the number of moves, after which a clear trend emerges. Player 2's greatest chance of victory is not until Move 8, while Player 1 has substantial play advantages. This information is crucial to developing the playstyle of the artificial player profiles. Early play must attempt to establish a solid foothold to account for the severe imbalance between players.



	5	6	7	8	9
Ties					46,080
P2 Wins		5,328		72,576	
P1 Wins	1,440		47,952		81,792

Figure 1.5: Detailed Terminal States by Move (Data: Juul, 2003)

Efficient play demands recognizing the significant imbalance between players. Player 1 is given significant power from possessing a greater number of moves, opening and potentially closing the game at the 9th move. Player 2's options are limited, their victories accounting for nearly a third of all possible terminal states, and even then the majority of those are only available by the 8th move. Such a disadvantage means Player 2 must be prepared to spend substantial effort in blocking the many wins available to Player 1. Ultimately when a player faces an equally clever opponent, draw games are the most likely outcome as both players will recognize potential wins and seek to block them.

II. The Table

The Table is the portion of code storing common resources and tools. Each element on the Table is accessible by all players, with external management handled by the Referee (section IV). Comprised of five separate parts, the Table acts as the central point of the game and our code. The table's centralized design was inspired by physical game tables, where players refer to the elements in play on the table as they mentally consider moves. A centralized data structure means players only require easily set pointers on where to find the relevant data and tools, rather than having to hand the board to each player. Centralization creates more efficient and easily managed code.

- **The Board**
- **Valid moves**
- **Player tokens**
- **Draw Board**
- **Is Won**

The Board

The game's 3x3 grid board requires some translation into machine language; numbering the grid and create a list for the squares easily accomplishes this requirement. Using the number-pad for our game board offers a familiar point of references and ultimately creates readable code while still easily processed by the computer.

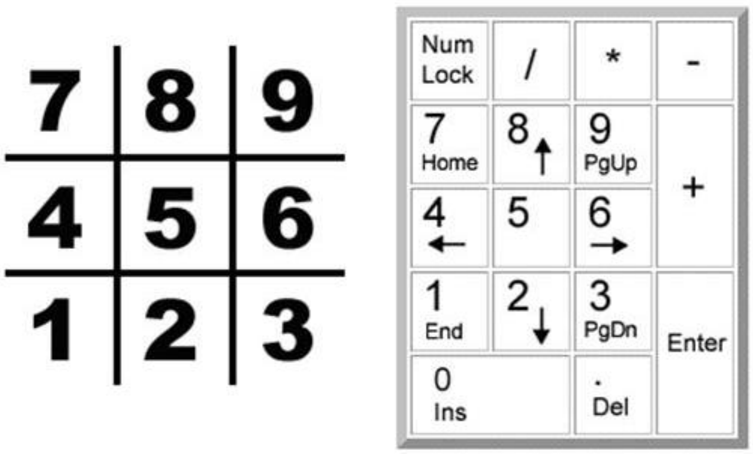
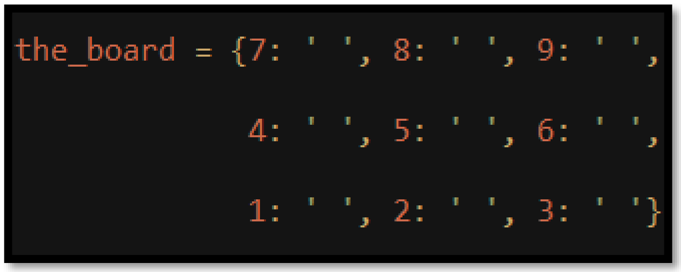


Figure 2.1: The Board and Number-pad (Sweigart, 2017)

Dictionaries

A list of items is easily managed by the computer, but the code still requires a way to track and reference player tokens in play, a goal easily accomplished with a programming dictionary. Dictionaries are special kind of list that store a list entry with a distinct and modifiable bit of information, in this case our player tokens. The Table will use a dictionary with entries 1-9, allowing each square to be written as moves are made. *Figure 2.2* is a visualization of the board code. The `' '` next to each entry indicaes an empty square. The appripriate token is written as played, so if X plays center square, entry `5: ' '` becomes `5: 'X'` (In the final code the board is shown as `{}`, as the numbers and corresponding entries are created later by the Referee)



```
the_board = {7: ' ', 8: ' ', 9: ' ',
             4: ' ', 5: ' ', 6: ' ',
             1: ' ', 2: ' ', 3: ' '}
```

Figure 2.2: Visualization of code for the Board

Valid Moves

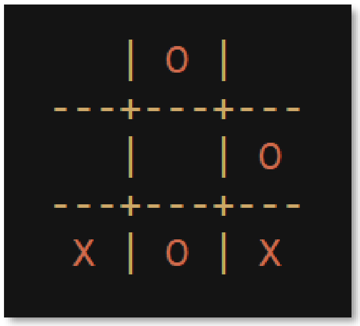
A crucial part of our game analysis is an understanding of available moves on the board. To save our artificial players from having to parse the board and discover blank squares, a list is stored next to the board containing the grid numbers of blank squares. This list is created and managed externally, and as moves are made the corresponding entries are removed. Creating a separate list simplifies the code for players, as otherwise the necessary parsing tools would add unnecessary complexity, while our simple list is easily read by player code and accomplishes the same.

Player Tokens

To ensure moves are never confused or players improperly called, a two-item list was created with the entries 'X' and 'O'. This list operates on a simple principle: the first list entry is the next token to be played, and after each move the list was reversed by the Referee. Storing tokens in a list enables easy reference for the Referee and allows players to easily identify their own tokens; a player with an active turn simply checks the first and last tokens in the list to respectively identify itself and its opponent, instead of having to know or be explicitly told token identities.

Draw Board

A simple tool purely for human benefit, as lines of code are far less readable and entertaining than a visual board. With the Draw Board tool, the code for the board can be visualized in a simple, readable format. *Figure 2.3* shows a drawn board.



```
   | O |
---+---+---
   |   | O
---+---+---
 X | O | X
```

Figure 2.3: Drawn board

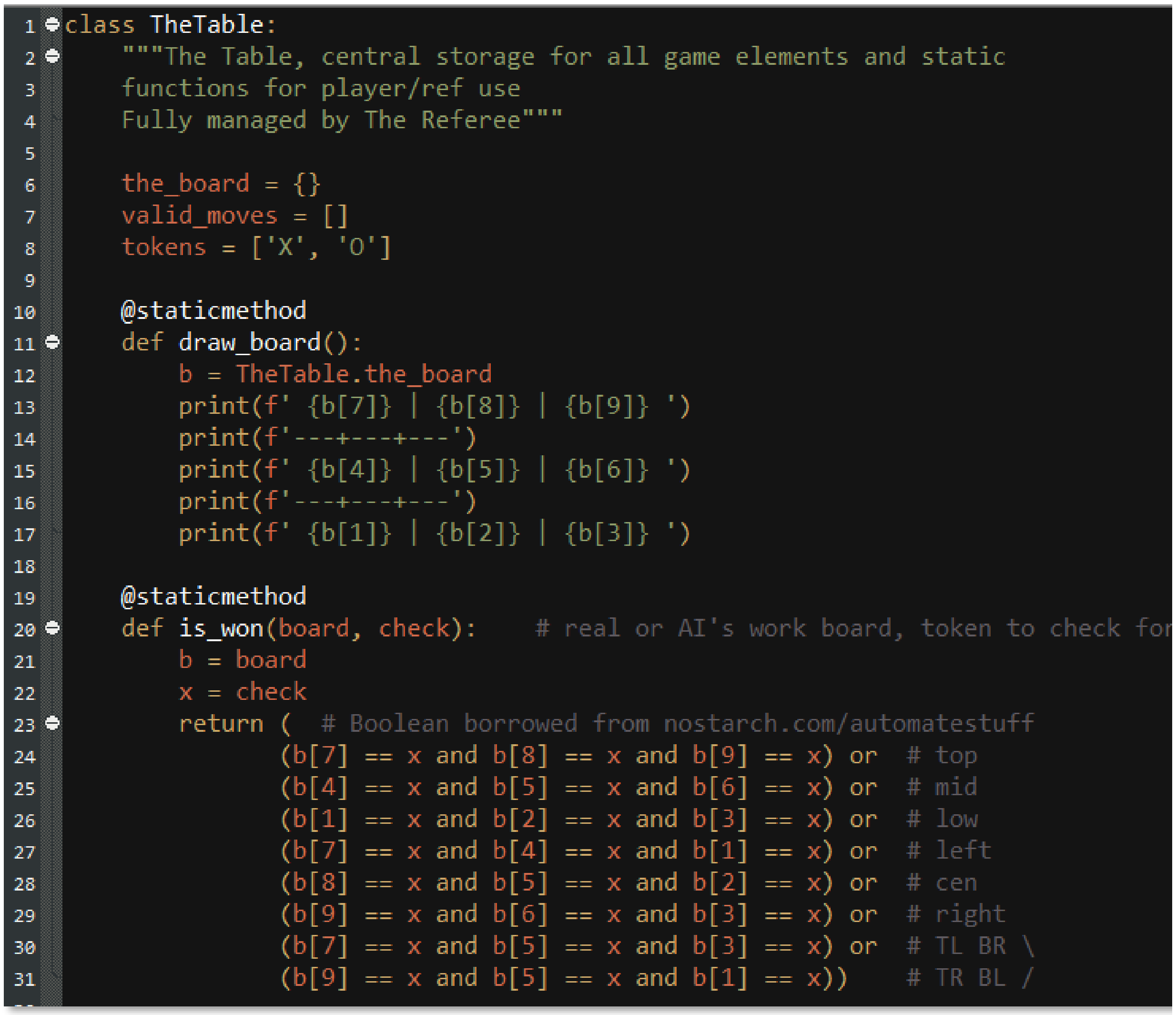
Is Won

Games without winners are games without purpose. This section of code is designed to parse the board and identify any winning positions (horizontals, verticals, or diagonals). This tool is designed to allow win checks against any board with any token, a crucial component for players seeking to determine if future moves could win the game for either player.
The name 'Is Won' was selected for readability in code ('If game is won, do a thing' is an easily understood and functional statement)

The Code

The Table's centralized designed was modeled after physical game tables, where players refer to game elements in playas they mentally consider moves.

In earlier builds the game placed heavy reliance on handoffs, where game elements were handed to players each turn and players relied on their own validity parsing and condition checks. Scalability was greatly hampered, as any modification required explicit restructuring on how the game elements were handed to players and how player modules received them.



```python
class TheTable:
    """The Table, central storage for all game elements and static
    functions for player/ref use
    Fully managed by The Referee"""

    the_board = {}
    valid_moves = []
    tokens = ['X', 'O']

    @staticmethod
    def draw_board():
        b = TheTable.the_board
        print(f' {b[7]} | {b[8]} | {b[9]} ')
        print(f'---+---+---')
        print(f' {b[4]} | {b[5]} | {b[6]} ')
        print(f'---+---+---')
        print(f' {b[1]} | {b[2]} | {b[3]} ')

    @staticmethod
    def is_won(board, check):    # real or AI's work board, token to check for
        b = board
        x = check
        return (  # Boolean borrowed from nostarch.com/automatestuff
                (b[7] == x and b[8] == x and b[9] == x) or  # top
                (b[4] == x and b[5] == x and b[6] == x) or  # mid
                (b[1] == x and b[2] == x and b[3] == x) or  # low
                (b[7] == x and b[4] == x and b[1] == x) or  # left
                (b[8] == x and b[5] == x and b[2] == x) or  # cen
                (b[9] == x and b[6] == x and b[3] == x) or  # right
                (b[7] == x and b[5] == x and b[3] == x) or  # TL BR \
                (b[9] == x and b[5] == x and b[1] == x))    # TR BL /
```

Figure 2.4: Code for The Table

III. The Players

With the Table built, dedicated player modules must be developed to fill the metaphorical seats. Our code contains four unique player modules: one human input module, and three artificial player modules with varied levels of intelligence.

- **Human Prompt**
- **AI Rando**
- **AI Levi**
- **AI Ralph**

Module Template

To standardize implementation, a template was designed for the player modules. Standardizing the input and outputs of the player modules allows for modular construction, ensuring that player modules built to specifications will function the same when later utilized.
The standardized template requires two common elements for modules:

1) a Main function, called by the Referee. The Main function is built to run each module's toolset.

2) the Final Move variable, decided by the module's toolset and returned to the Referee. The Some modules may revise their move selection before completion; the prefix Final accommodates this possibility for such instances.

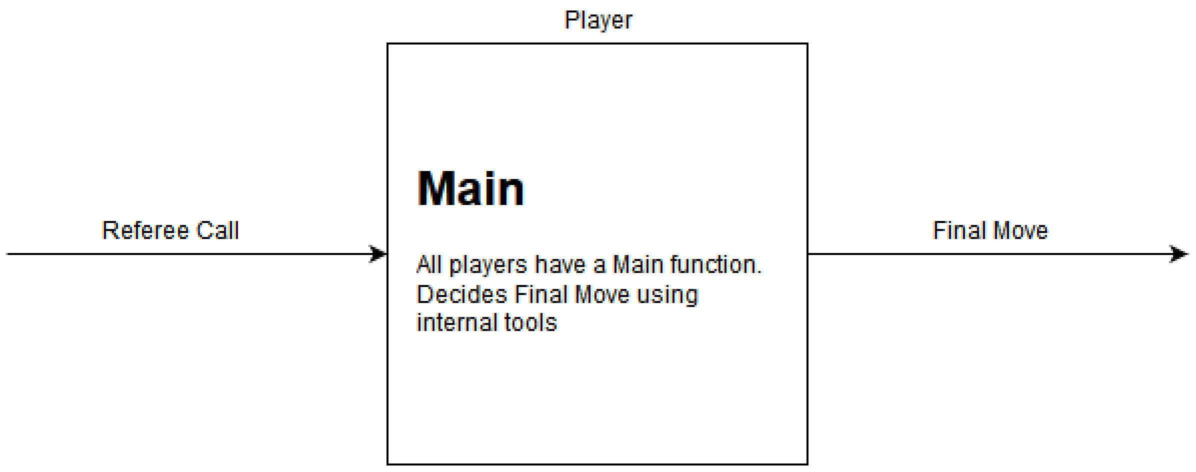
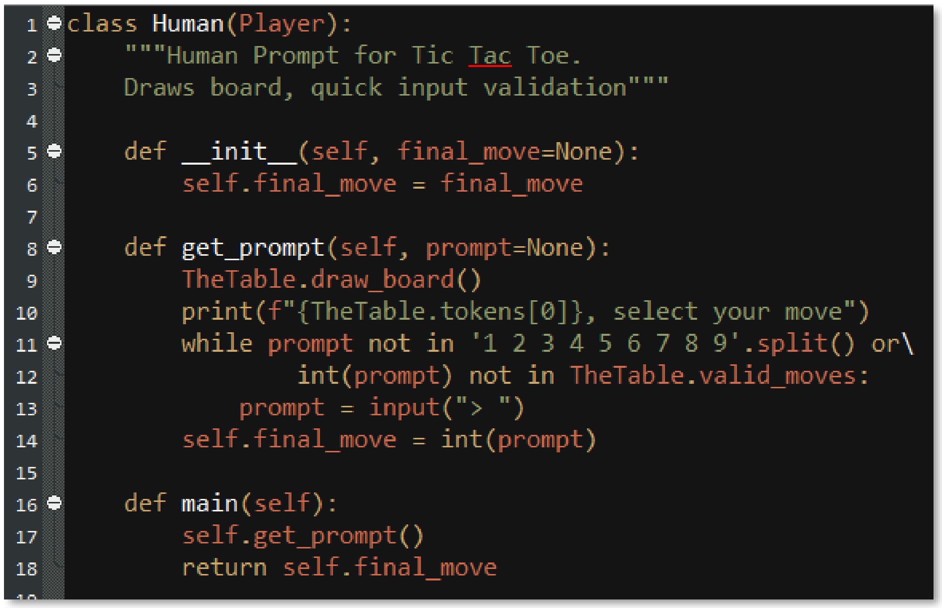

Player
Main
All players have a Main function. Decides Final Move using internal tools
Referee Call
Final Move

Figure 3.1: Standardized Player Template

Human Prompt

The Human Prompt module is built for human interaction. The module displays the board in a viewable format with the Draw Board tool and asks for the next move.

The code also contains a quick validation against the Valid Moves list to prevent illegal move submissions. The code will not accept input until a valid entry is submitted.

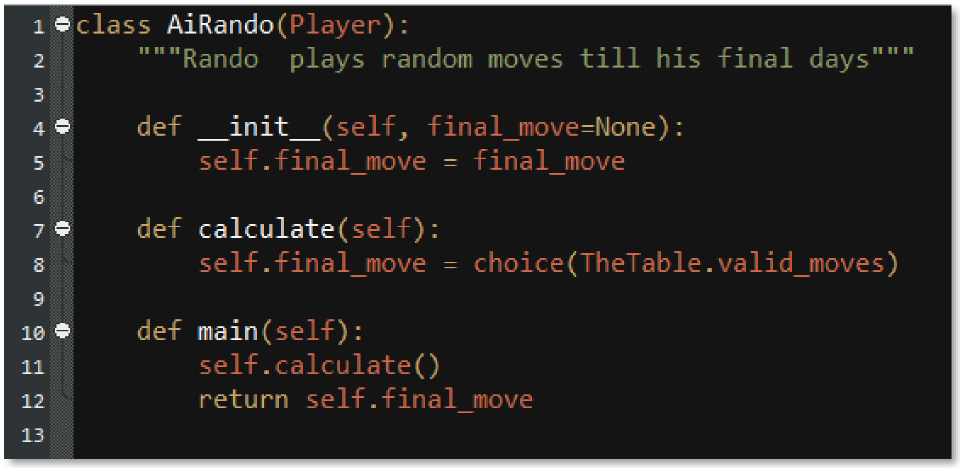


```python
class Human(Player):
    """Human Prompt for Tic Tac Toe.
    Draws board, quick input validation"""

    def __init__(self, final_move=None):
        self.final_move = final_move

    def get_prompt(self, prompt=None):
        TheTable.draw_board()
        print(f"{TheTable.tokens[0]}, select your move")
        while prompt not in '1 2 3 4 5 6 7 8 9'.split() or\
                int(prompt) not in TheTable.valid_moves:
            prompt = input("> ")
        self.final_move = int(prompt)

    def main(self):
        self.get_prompt()
        return self.final_move
```

Figure 3.2: Human Prompt

AI Rando

The simplest of our artificial players, Rando's Main function selects a random entry from the Table's list of valid moves. Working with the valid list ensures Rando will always return a valid, if illogical, move.



```python
class AiRando(Player):
    """Rando  plays random moves till his final days"""

    def __init__(self, final_move=None):
        self.final_move = final_move

    def calculate(self):
        self.final_move = choice(TheTable.valid_moves)

    def main(self):
        self.calculate()
        return self.final_move

```

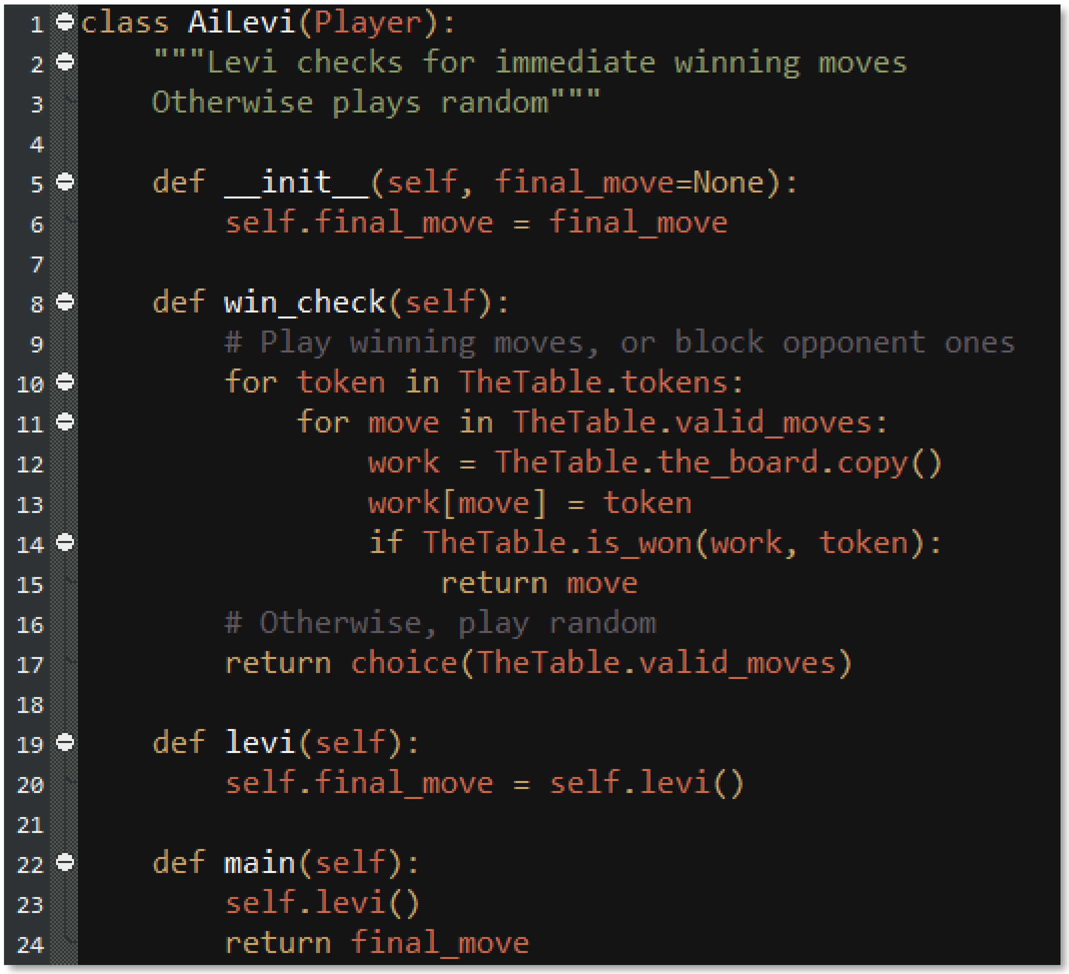
Figure 3.3: Rando's code

AI Levi

Levi is built on a simple conditional check: compare all valid moves for a winning move, first for itself and then for its opponent. This conditional check creates an opponent that actively tries to win while attempting to prevent its opponent from doing the same. If no immediate winning conditions are met, Levi falls back to simpler routines and plays a random space.

Levi's Evolution

An earlier version of Levi was built out of a spanning list of if/else statements based on the Model of Expert Performance (Crowley and Siegler, 1993). While an effective player, this level 1 opponent relied on far too many lines of code and was notoriously unaccommodating to adjustments; implementation of new routines regularly broke Levi. The routines were pruned and refined to the simpler win check. With a robust code, Levi fell comfortably in the middle as our 'moderate' opponent.



```python
class AiLevi(Player):
    """Levi checks for immediate winning moves
    Otherwise plays random"""

    def __init__(self, final_move=None):
        self.final_move = final_move

    def win_check(self):
        # Play winning moves, or block opponent ones
        for token in TheTable.tokens:
            for move in TheTable.valid_moves:
                work = TheTable.the_board.copy()
                work[move] = token
                if TheTable.is_won(work, token):
                    return move
        # Otherwise, play random
        return choice(TheTable.valid_moves)

    def levi(self):
        self.final_move = self.levi()

    def main(self):
        self.levi()
        return final_move
```

Figure 3.4: Levi's code

AI Ralph

Ralph is the most ambitious of our player modules, aiming to work all quarter million possible games of Tic-Tac-Toe and scoring every terminal state.

Ralph first clones the board and table into its memory, allowing greater mutability without risk of modifying the actual game board. Additionally, a clone of the list of payers is created so that Ralph can safely track moves as it iterates down the board. With clones in hand a scoresheet is created, the key element in tracking possible wins and losses.

Recursive Analysis

Ralph's core component is a recursive function, worked on the clone board built in the preparation stage. The function works by figuring each all possible paths from each valid move available to itself and the opponent and scoring the eventual outcomes.

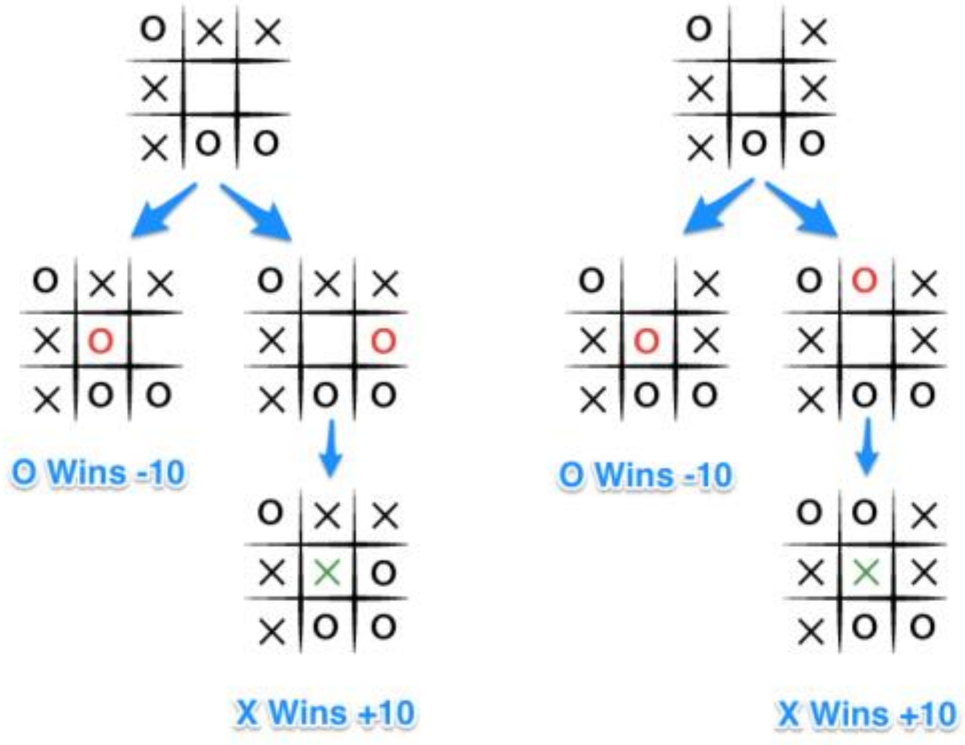


Figure 3.5: Simple Recursion and Scoring (Fox, 2013)

Depth Tracking

A depth system was also implemented, to balance immediate wins over later ones. Points are subtracted from the score depending on how many moves ahead the terminal state is.

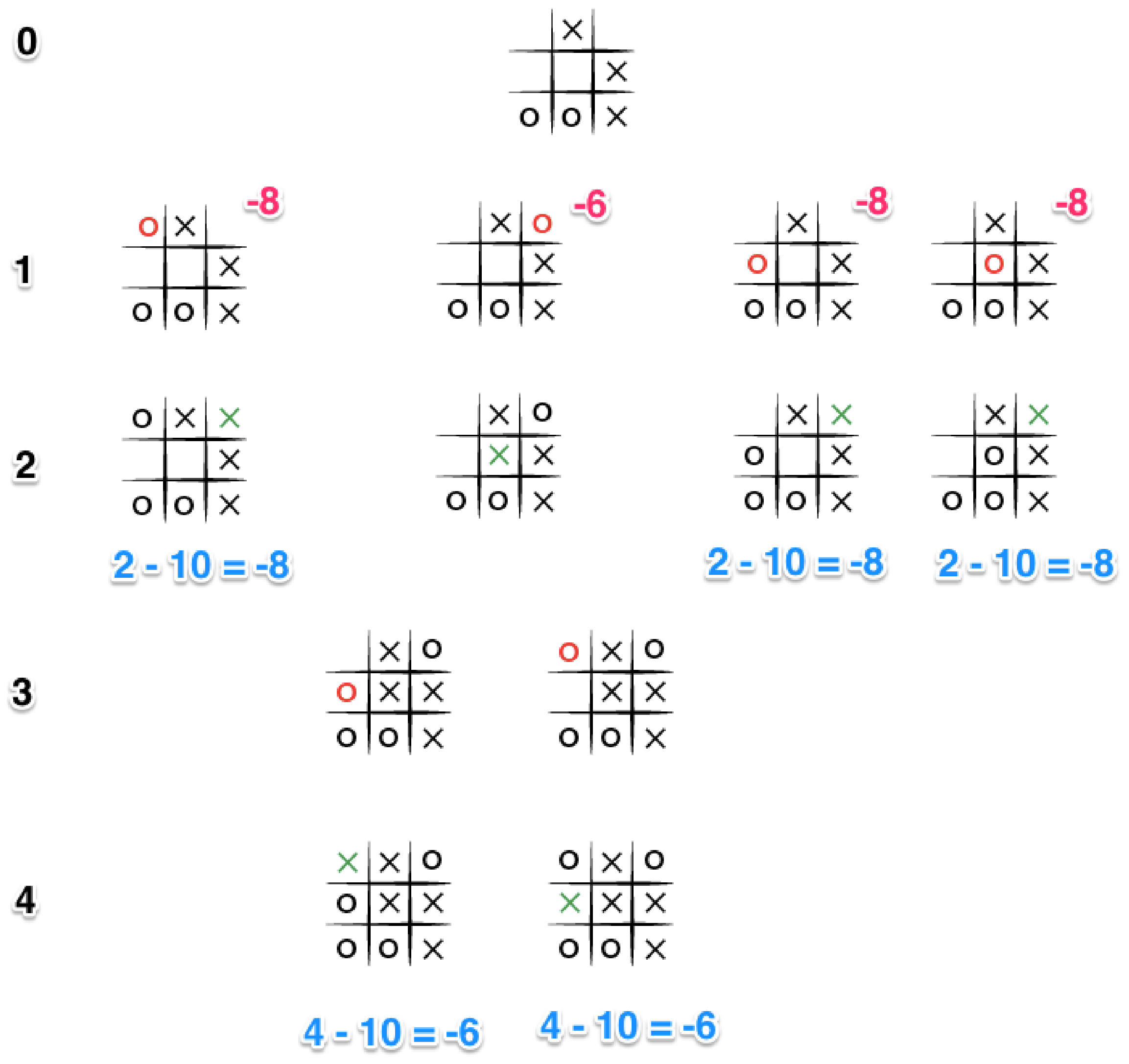


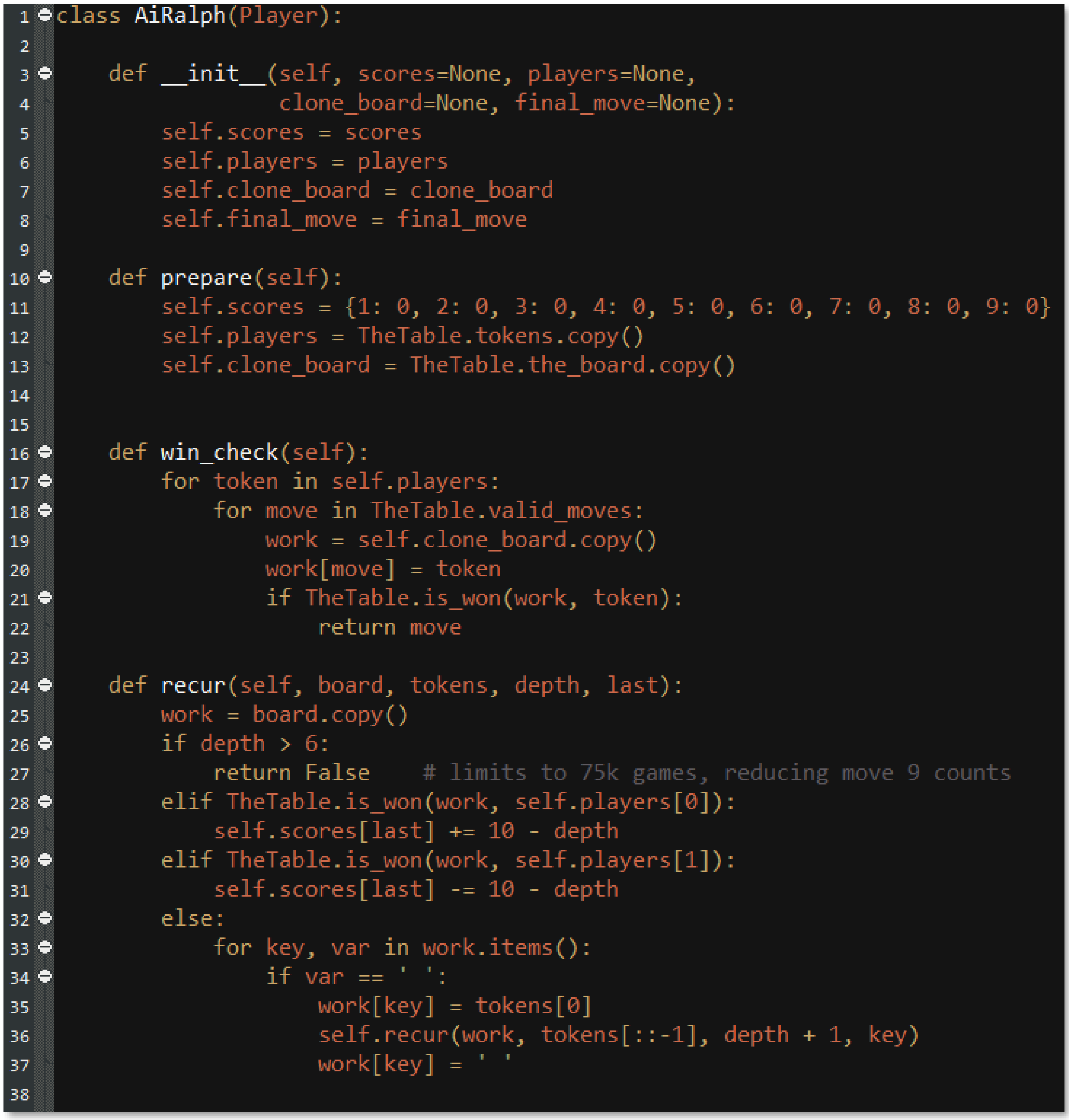
Figure 3.6: Depth in Recursion (Fox, 2013)

Once scores are assigned Score Check is used to identify priority scoring. Score Check is based on a minimax system, a game theory rule used for minimizing the possible losses while maximizing chances of victory.

Analysis on the final score tallies identifies either the maximum (cumulative wins) or minimum (cumulative losses), and the move corresponding to that score is selected. If scores are tied (i.e. symmetrical corners) a random tie breaker selects any of the high scoring moves.

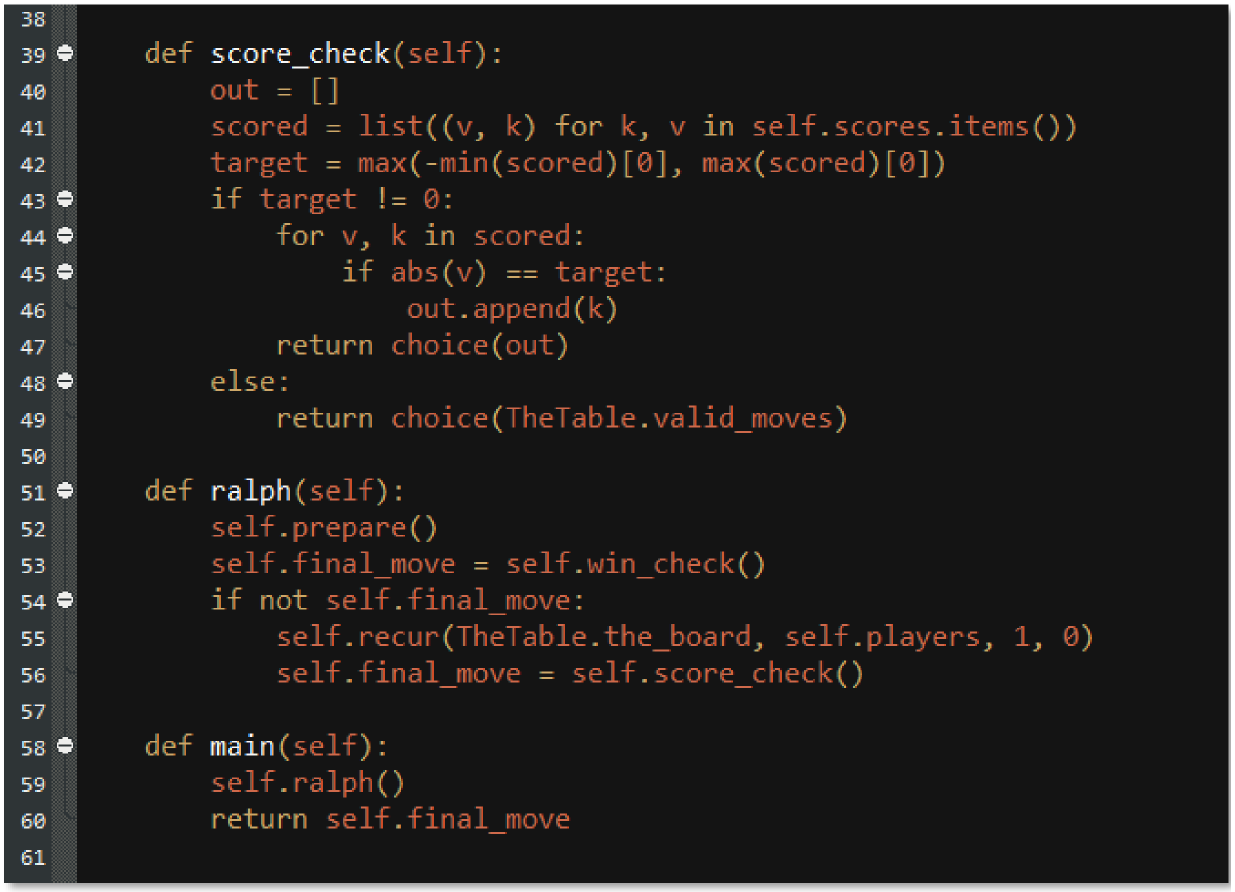
Final Move prepared, Ralph prepares to return the move to the Referee

Ralph's Code



```python
class AiRalph(Player):

    def __init__(self, scores=None, players=None,
                 clone_board=None, final_move=None):
        self.scores = scores
        self.players = players
        self.clone_board = clone_board
        self.final_move = final_move

    def prepare(self):
        self.scores = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0}
        self.players = TheTable.tokens.copy()
        self.clone_board = TheTable.the_board.copy()


    def win_check(self):
        for token in self.players:
            for move in TheTable.valid_moves:
                work = self.clone_board.copy()
                work[move] = token
                if TheTable.is_won(work, token):
                    return move

    def recur(self, board, tokens, depth, last):
        work = board.copy()
        if depth > 6:
            return False    # limits to 75k games, reducing move 9 counts
        elif TheTable.is_won(work, self.players[0]):
            self.scores[last] += 10 - depth
        elif TheTable.is_won(work, self.players[1]):
            self.scores[last] -= 10 - depth
        else:
            for key, var in work.items():
                if var == ' ':
                    work[key] = tokens[0]
                    self.recur(work, tokens[::-1], depth + 1, key)
                    work[key] = ' '

```

Figure 3.7: Ralph's code, Part 1



```
    def score_check(self):
        out = []
        scored = list((v, k) for k, v in self.scores.items())
        target = max(-min(scored)[0], max(scored)[0])
        if target != 0:
            for v, k in scored:
                if abs(v) == target:
                    out.append(k)
            return choice(out)
        else:
            return choice(TheTable.valid_moves)

    def ralph(self):
        self.prepare()
        self.final_move = self.win_check()
        if not self.final_move:
            self.recur(TheTable.the_board, self.players, 1, 0)
            self.final_move = self.score_check()

    def main(self):
        self.ralph()
        return self.final_move
```

Figure 3.8: Ralph's code, Part 2

Ralph's Dysfunctional Scoring System

The scoring system is currently functional but carries a crippling flaw: because the score sheet tracks scores in a single integer, immediate moves can be lost against later moves (one immediate blocking move worth -10 points is lost against 4 later winning moves worth +3 points each) Score overwriting became a significant issue when where tens of thousands of move 9 states games obfuscate early game analysis. Two significant changes were made to satisfy immediate needs:

1) A depth limiter was implemented to cut the quarter million game analysis down to 75 thousand moves, ensuring proper early game preference of center and corner square moves

2) For late game performance, Levi's single move win check procedure was incorporated with higher priority than the imperfect scoring system, ensuring high priority wins and blocks are not lost.

With the combined procedures, Ralph is a successful artificial opponent who opens strategically and takes blocking or winning moves as they appear.

(Note: Preliminary testing on a revised scoring system with dedicated win and loss tracking is promising. The revised system will be fully implemented into the next release.)

IV. The Referee

- **The Setup**
- **Spectator Mode**
- **Play Game**
- **Game Over**
- **Play Again**

The Referee is the game master of our code, and acts as rule keeper, coordinator, and is responsible for management of the Table. The Referee accomplishes this by running through its many tools, each with a unique subset of rules and procedures. The Referee's philosophy is that of an impartial agent between players and game, indifferent of who the players are. The Referee is the only code segment authorized to work the Table while ensuring stability and security for the Table

The Setup

The setup runs at the beginning of each round to create a fresh Table, player assignments, and sets the play order each round. This is accomplished through 4 steps:

1) First step is to set the Table. The board and list of valid moves is cleared and set, ensuring any previous game states are cleared. The board dictionary and list of valid moves are set simultaneously with an iterator to work through numbers 1-9.

2) Player are assigned to tokens using a simple human input prompt. Player calls are assigned to tokens locally, to indicate who to get the next move from. A game between a human player and artificial player Levi creates the following:

```
get_from = {'X': 'Human', 'O': 'AiLevi'}
```

The modular build of player modules means any player can be assigned to any token or the same player module can play both tokens. Modules are sanitized each turn, meaning no information carries over between the two.

3) After player calls are set, a True/False flag called No Human is set depending on if any human players were selected. This flag will activate spectator mode defined later.

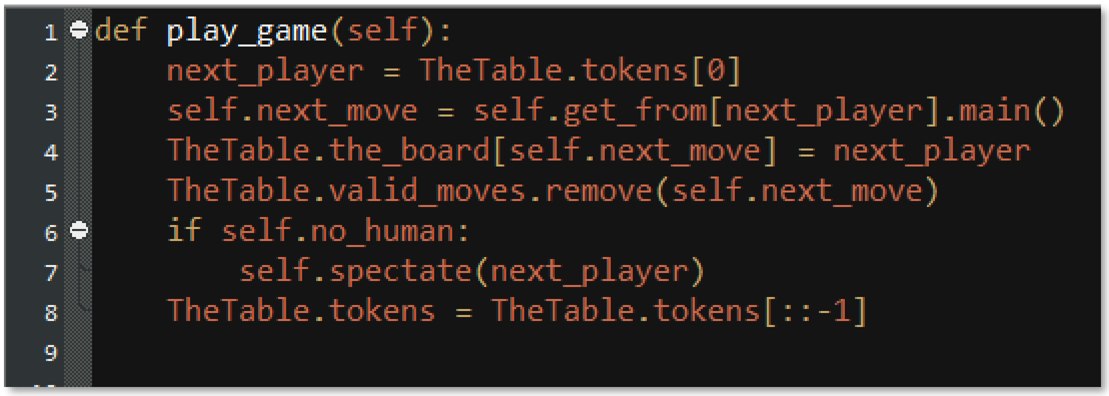
4) With the board set and player calls assigned, a simple 50/50 random chance is used to set the player order, ensuring a random start for 'X' or 'O'.

Spectator Mode

Spectator mode is a simple tool used to allow human players to watch a round between two artificial opponents. Because the Human player module is the only one regularly drawing the board, a match between two artificial opponents, neither required to visualize play, would be impossible to watch. Spectator mode draws the board, pauses the game, and indicates the last player's move made, allowing the spectator to observe the game.

Play Game

The Play Game function runs each move and is responsible for active components of the game. As a crucial component for the Referee, this function deserves a line by line break down.



```
1 def play_game(self):
2     next_player = TheTable.tokens[0]
3     self.next_move = self.get_from[next_player].main()
4     TheTable.the_board[self.next_move] = next_player
5     TheTable.valid_moves.remove(self.next_move)
6     if self.no_human:
7         self.spectate(next_player)
8     TheTable.tokens = TheTable.tokens[::-1]
9
```

Figure 4.1: The Play Game Function

1) The first line announces that we are defining a new function, play_game. The parameter 'self' instructs the code to reference where it is placed when seeking data. Since this function is placed in the Referee, any 'self' prefixes indicate data stored in the Ref.

2) Identifies the token queued from the Tokens stored on the Table (this code begins counting at zero, so [0] means first item listed). Identifying the next player's token, 'X' or 'O', is crucial to calling the correct player

3) Calls the next player to indicate their desired move. This line uses the token from step 1 to call the appropriate player, using the get_from dictionary created in Set Up. Every player is built with a tool main() designed to answer to a standardized call and return their move to the Ref, so this line operates identically regardless of player. The returned move is received by the Ref and saved as the Next Move.

4) The Ref goes to the board (stored on the Table) and writes the next player's token on the designated space. For simplicity of code and security of the board, the Ref is the only one authorized to write changes to the board, players are only allowed to read the board.

5) With a move written, the board then clears the grid entry from the Table's valid move list, ensuring player modules will not attempt to return duplicate moves.

6-7) Line 6 checks the No Human flag, either True or False, on whether to run spectator mode. If No Human is false (a human *is* playing) line 7, which runs spectator mode, is skipped.

8) Finally, the tokens are flipped. The suffix [::-1] reverses a list, so our two-item list of tokens reverses.

Game Over

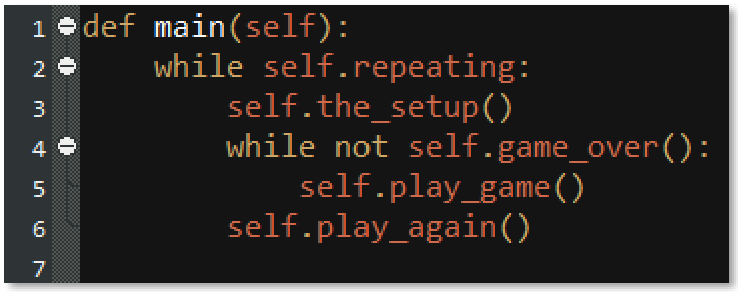
Game Over is comprised of two smaller tools, both dedicated to recognizing terminal states and triggering True/False flags. The first identifies a victory made by the previous player using the Is Won tool from the Table, and saves the Game Won flag as flag True or False. The second tool works in similar but dedicated to identifying a lack of remaining blank spaces, with a True or False flag saved as Game Tied. Game Over then checks both these flags, Game Won and Game Tied, and uses the first True flag found to signal a terminal state. A quick summary indicating the winning player, or a tie game is saved to a variable called Results for post-game display.

Play Again

Play again is a simple human prompt function, setting the True/False flag Repeating, the flag used to start a new game after a previous one. By default, the flag is set as True.

Main

The Ref's main function sets and calls the defined tools to run the game.



```
1  def main(self):
2      while self.repeating:
3          self.the_setup()
4          while not self.game_over():
5              self.play_game()
6          self.play_again()
7
```

Figure 4.2: The Main Function

2) Line 2 calls on the Repeating flag, the True/False defined above. This ensures the following lines will loop while the Repeating flag is True

3) Runs the Setup

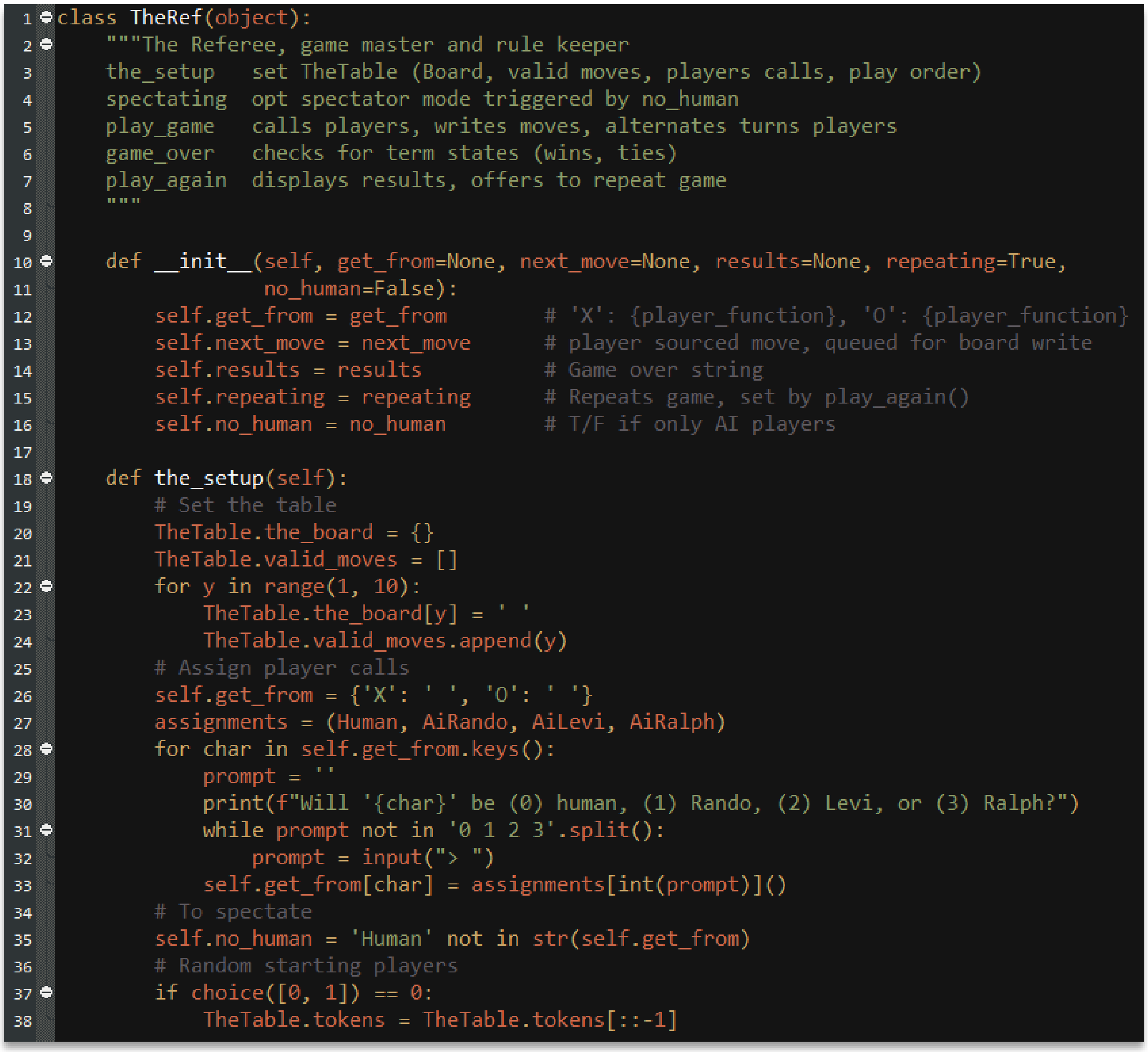
4) Line 4 functions similarly to line 2 but with the Game Over tool. While the game is *not* over, line 5 repeats

5) Runs the game. Recall that each time Play Game runs its 6 steps repeat, calling players, writing moves, and flipping tokens.

6) Line 4 keeps line 5 repeating until terminal state. Once the game is over, Play Again is called to keep or change the Repeating flag's True/False status.

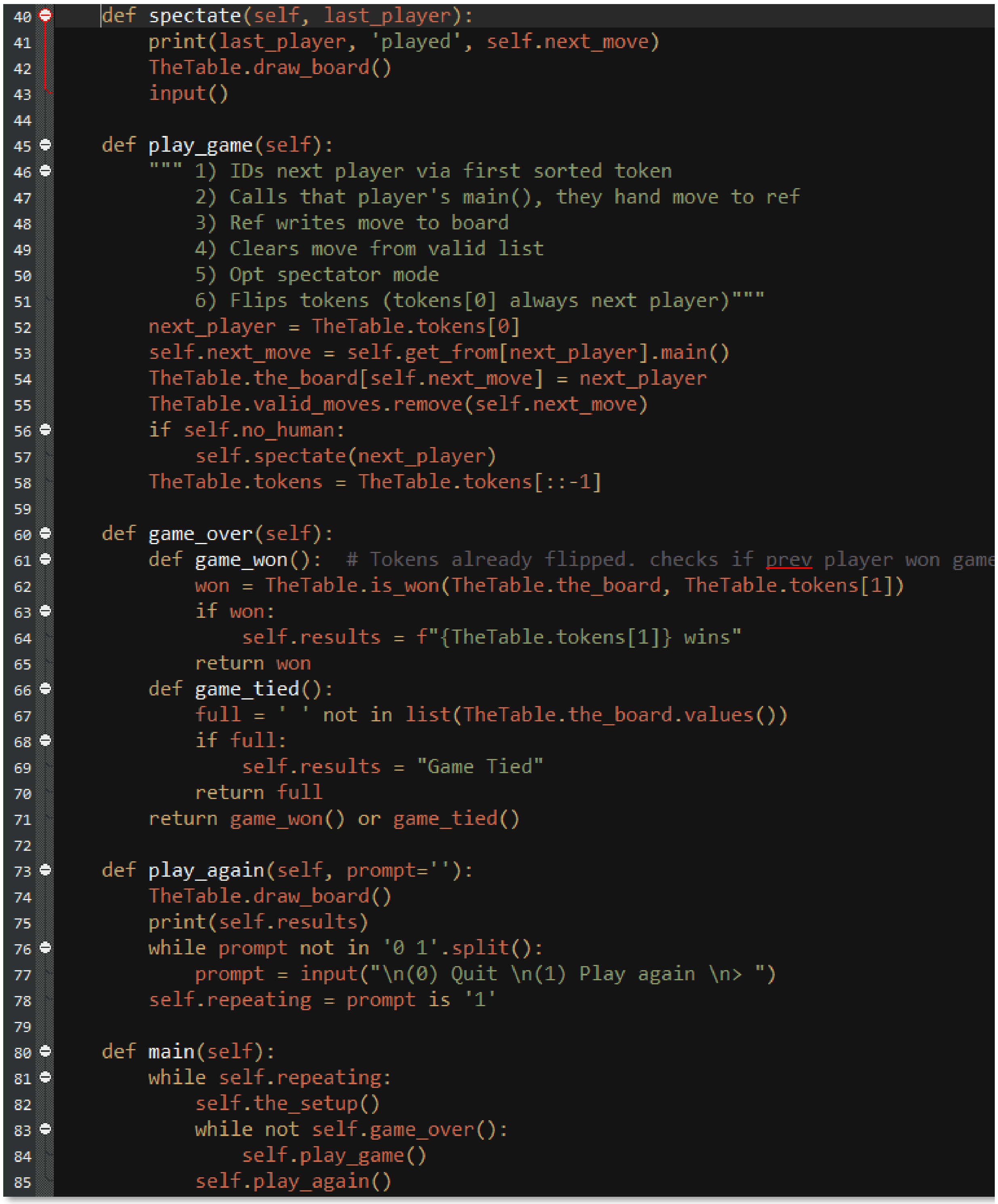
The Code

The Referee is by necessity the most complex piece of code in the game. Many new tools were devised for this block of code, especially compared to earlier prototypes. The Board was originally contained within the board and given to players as they were called. As mentioned in the Board's code review, this was highly inefficient and hampered scalability. *Figure 4.3* shows the flags used in other tools being built, and the setup. *Figure 4.4* shows the remaining tools, including the Main tool responsible for running all of the Referee's tools.



```python
class TheRef(object):
    """The Referee, game master and rule keeper
    the_setup   set TheTable (Board, valid moves, players calls, play order)
    spectating  opt spectator mode triggered by no_human
    play_game   calls players, writes moves, alternates turns players
    game_over   checks for term states (wins, ties)
    play_again  displays results, offers to repeat game
    """

    def __init__(self, get_from=None, next_move=None, results=None, repeating=True,
                 no_human=False):
        self.get_from = get_from        # 'X': {player_function}, 'O': {player_function}
        self.next_move = next_move      # player sourced move, queued for board write
        self.results = results          # Game over string
        self.repeating = repeating      # Repeats game, set by play_again()
        self.no_human = no_human        # T/F if only AI players

    def the_setup(self):
        # Set the table
        TheTable.the_board = {}
        TheTable.valid_moves = []
        for y in range(1, 10):
            TheTable.the_board[y] = ' '
            TheTable.valid_moves.append(y)
        # Assign player calls
        self.get_from = {'X': ' ', 'O': ' '}
        assignments = (Human, AiRando, AiLevi, AiRalph)
        for char in self.get_from.keys():
            prompt = ''
            print(f"Will '{char}' be (0) human, (1) Rando, (2) Levi, or (3) Ralph?")
            while prompt not in '0 1 2 3'.split():
                prompt = input("> ")
            self.get_from[char] = assignments[int(prompt)]()
        # To spectate
        self.no_human = 'Human' not in str(self.get_from)
        # Random starting players
        if choice([0, 1]) == 0:
            TheTable.tokens = TheTable.tokens[::-1]
```

Figure 4.3: Code for The Referee, part 1

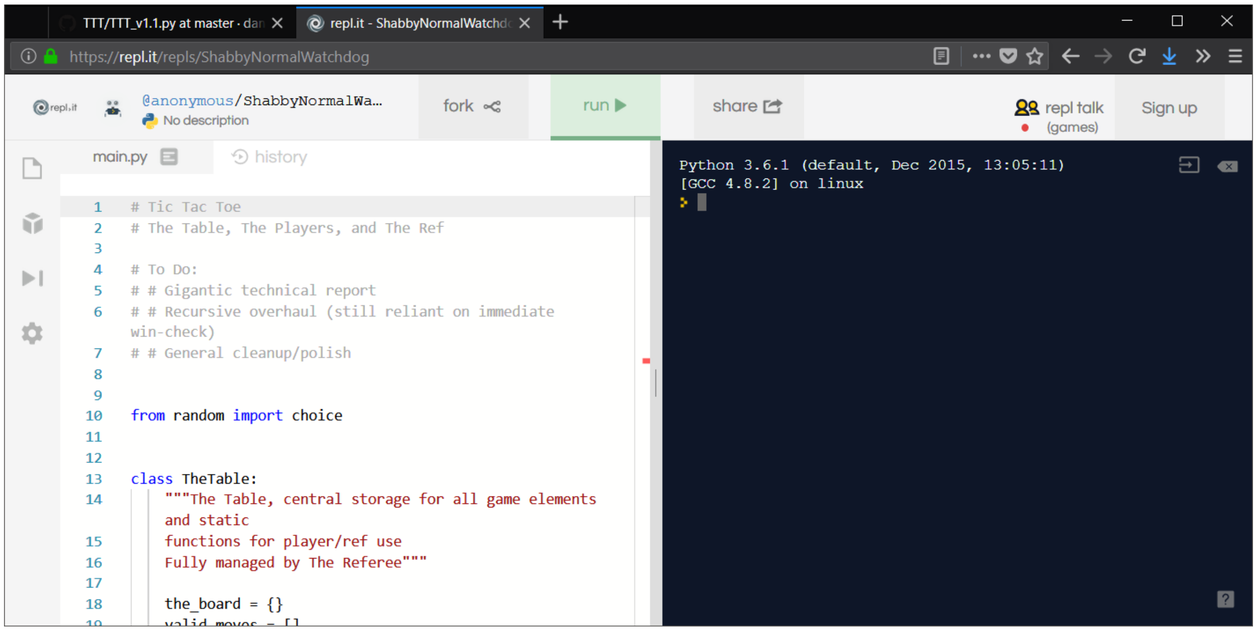


```python
    def spectate(self, last_player):
        print(last_player, 'played', self.next_move)
        TheTable.draw_board()
        input()

    def play_game(self):
        """ 1) IDs next player via first sorted token
            2) Calls that player's main(), they hand move to ref
            3) Ref writes move to board
            4) Clears move from valid list
            5) Opt spectator mode
            6) Flips tokens (tokens[0] always next player)"""
        next_player = TheTable.tokens[0]
        self.next_move = self.get_from[next_player].main()
        TheTable.the_board[self.next_move] = next_player
        TheTable.valid_moves.remove(self.next_move)
        if self.no_human:
            self.spectate(next_player)
        TheTable.tokens = TheTable.tokens[::-1]

    def game_over(self):
        def game_won():  # Tokens already flipped. checks if prev player won game
            won = TheTable.is_won(TheTable.the_board, TheTable.tokens[1])
            if won:
                self.results = f"{TheTable.tokens[1]} wins"
            return won
        def game_tied():
            full = ' ' not in list(TheTable.the_board.values())
            if full:
                self.results = "Game Tied"
            return full
        return game_won() or game_tied()

    def play_again(self, prompt=''):
        TheTable.draw_board()
        print(self.results)
        while prompt not in '0 1'.split():
            prompt = input("\n(0) Quit \n(1) Play again \n> ")
        self.repeating = prompt is '1'

    def main(self):
        while self.repeating:
            self.the_setup()
            while not self.game_over():
                self.play_game()
            self.play_again()
```

Figure 4.4: Code for The Referee, part 2

V. The Game in Action

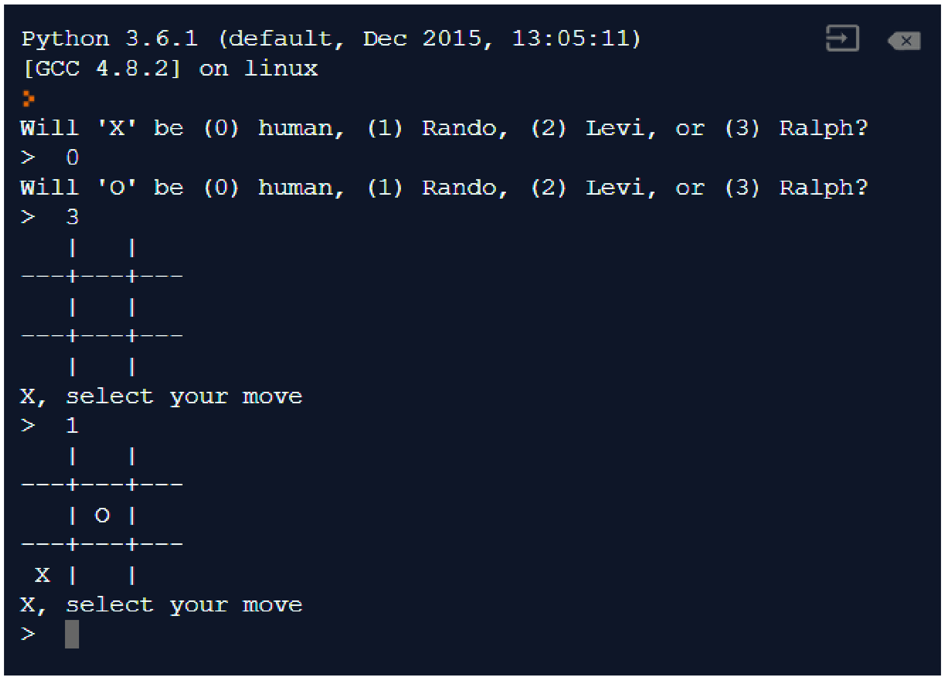
With theory handled and code constructed, it is now time to see this code in action. Python 3's versatility as a language means this code can be run from nearly any modern computer, even from its web browser. To best showcase the language's ability and enable readers to follow along if desired, the code will be run in the Replit online coding environment for Python 3, with the game code stored publicly on this writer's GitHub repository.


TTT/TTT_v1.1.py at master · dan
repl.it - ShabbyNormalWatchdo
https://repl.it/repls/ShabbyNormalWatchdog
repl.it
@anonymous/ShabbyNormalWa...
No description
fork
run
share
repl talk
(games)
Sign up
main.py
history

```
# Tic Tac Toe
# The Table, The Players, and The Ref

# To Do:
# # Gigantic technical report
# # Recursive overhaul (still reliant on immediate
win-check)
# # General cleanup/polish


from random import choice


class TheTable:
    """The Table, central storage for all game elements
    and static
    functions for player/ref use
    Fully managed by The Referee"""

    the_board = {}
    valid_moves = []
```

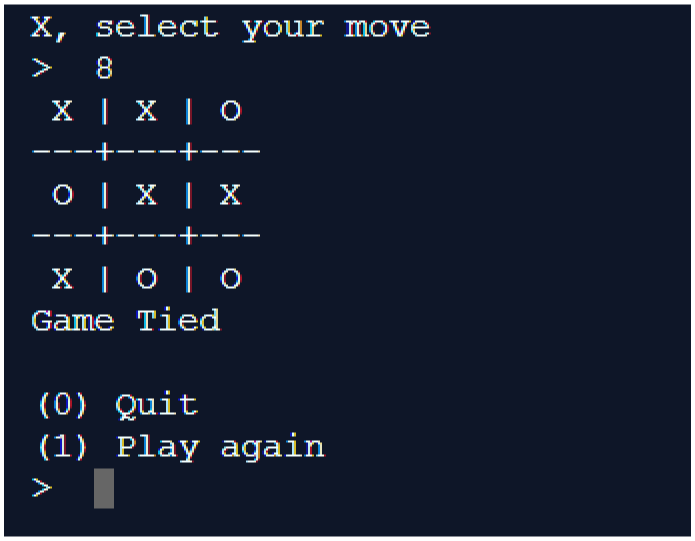
```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
>
```

Figure 5.1: Code Copied to Replit, Ready to Run



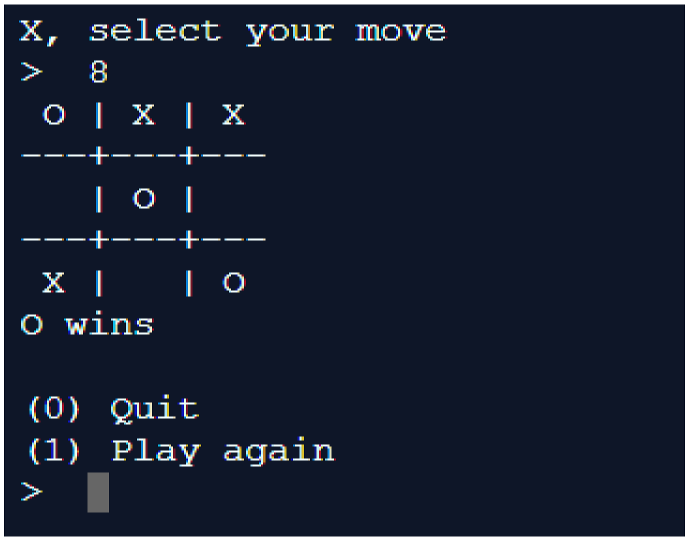
```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
>
Will 'X' be (0) human, (1) Rando, (2) Levi, or (3) Ralph?
>  0
Will 'O' be (0) human, (1) Rando, (2) Levi, or (3) Ralph?
>  3
   |   |
---+---+---
   |   |
---+---+---
   |   |
X, select your move
>  1
   |   |
---+---+---
   | O |
---+---+---
 X |   |
X, select your move
>
```

Figure 5.2: A Game in Progress



```
X, select your move
>  8
 X | X | O
---+---+---
 O | X | X
---+---+---
 X | O | O
Game Tied

 (0) Quit
 (1) Play again
>
```

Figure 5.3: A Tie Game



```
X, select your move
>  8
 O | X | X
---+---+---
   | O |
---+---+---
 X |   | O
O wins

 (0) Quit
 (1) Play again
>
```

Figure 5.4: Computer 'O' wins Game

Figure 5.1 shows the code copied from GitHub and into the online editor Replit. The Run button starts the program, as displayed in *figure 5.2*. The very first line, 'Will 'X' be (0) human, (1) Rando, (2) Levl, or (3) Ralph?" is the Referee running its Setup tool, seen in section V, referencing the player modules. A match is set using the Human prompt and Ralph modules, and the Table's board is drawn. The game begins by asking for the initial move and 'X' plays square 3, left corner. AI Ralph responds with 5, based of its recursive algorithms. *Figure 5.2* and *figure 5.3* show two terminal states, both properly recognized, and results displayed using the Game Over tool. The game offers to play again, using the Ref's Play Again tool, and the prompt for players populates again.

The only time code takes more than a split second to execute is when Ralph analyzes its first move; using its recursive algorithms on a blank board takes the most time as it works through as many terminal states as we've allowed it.

Environment	repl.it/languages/python3
Code	github.com/daniel-avalos/TTT/blob/master/TTT_v1.1.py

VI. Conclusion

What Was Accomplished

Objectively, this project was a success. A simple game to play Tic-Tac-Toe was created based on substantial game theory and programming research, and the addition of artificial opponents a point of particular pride. Keeping elements separate with limited interaction was a central philosophy, the use of the Table a point of importance, as keeping the crucial data stored in one place allowed players to only need pointers on where to find the game data. The Referee successfully runs the game without any issue, and properly sanitizes data between every game. With code posted online and online environments available, this script stands as a public declaration of the accomplishment here.

Lessons Learned

Above the joy of programming and the excuse to write a very long report, this project was also a substantial learning experience. The artificial players' rudimentary state processing will come in handy in any situation where rules can be defined, and results are measurable (namely, code testing and debugging). The research stage was full of game theory, and even some data crunching (the parsing of Juul's 2-million-line document was definitive data science, one of the most in demand fields of computer science). The modular design of the code, keeping elements separate and interacting with standard protocols, was heavily inspired by internet protocols. These sorts of protocols operate on fundamental theories of separate levels of code interacting with standardized methods. Implementing such a system into the core game greatly increased its robustness and scalability, any code constructed with similar principles will enable greater growth in the future. Finally, the modular structure has one more benefit: many sections of this code can be reused and integrated into other non-game programs. The Board is simple data storage with access rights, crucial in any sort of large data storage system. Recursive Ralph was originally developed to create all combinations of numbers from a list and can easily be reintegrated into a feature. The Referee's tools will see future use in non-game code, namely any situation where tools must be called on in a particular order

Guiding Research

This software was not developed in a vacuum. Crowley's "Flexible Strategy Use in Young Childrens Tic-Tac-Toe" was key in understanding the crucial strategies involved in playing the game. Two books formed heavy inspiration for the code's design: Sweigart's *Invent Your Own Computer Games with Python* and Heineman's *Algorithm in a Nutshell*. The Table's Win Check is a modified function from Sweigart's book, gracefully borrowed under the open source license; Heineman's *Algorithm* was instrumental guidance in constructing the recursive algorithm used for AI player Ralph, along with Jason Fox's "Understanding the Minimax Algorithm"

Future Plans

This code has seen substantial overhauls and adjustments since initial prototype and will undoubtedly see more improvements using the lessons learned here. There are four main goals for future releases:

1) Implementation of a more robust scoring system for Ralph

   While Ralph is a functional player, he is not perfect. Plans for an improved system exist, and will be fully developed for future releases

2) Player names

   Currently the Human modules do not have any assignable names to them, so matches between two people have no identifiable names, only 'X' and 'O'. A restructure of the Referee's Setup will allow optional name assignments to display during game. The creation of a better name system will also allow AI names to display on screen.

3) Develop read/write access into the Board

   The Board was developed in the same lines of code as the players and Ref. While in theory the Referee is the only module allowed to modify the Board, in practice this only happens because the player modules do not have any tools that CAN modify the board. While this was a feasible build for something 100% within the developer's hands, any future build or similar projects that involve other people will require more exclusive rights for the board. Read/Write access rights will see use in the board, as well as separate scripts for the elements with specified rights.

4) Further modular structure

   This is more of a cleanup step, as some lines of code still rely on less modular code (namely the Referee's Setup, which is comprised from one block of code rather than three separate elements) Enacting a more modular design will grant two key advantages: 1) create cleaner code, and 2) allow more of this game's code to be reused elsewhere.

Works Cited

Cohen, D.S. "OXO aka Noughts and Crosses - The First Video Game." lifewire.com/oxo-aka-noughts-and-crosses-729624 May 2018

Crowley, Kevin, and Robert S. Siegler. "Flexible Strategy Use in Young Childrens Tic-Tac-Toe." *Cognitive Science*, vol. 17, Oct 1993, 531-561.

Fox, Jason. "Understanding the Minimax Algorithm." Dec 2013, neverstopbuilding.com/blog/2013/12/13/tic-tac-toe-understanding-the-minimax-algorithm13 June 2018.

Heineman, George T., Gary Pollice and Stanley Selkow. *Algorithms in a Nutshell*. Sebastopol; O'Reilly, 2016.

Juul, Jesper. "255,168 Ways of Playing Tic Tac Toe." July 2015, jesperjuul.net/ludologist/2003/12/28/255168-ways-of-playing-tic-tac-toe June 2018.

Munroe, Randall. "Tic-Tac-Toe." Dec 2010, xkcd.com/832 June 2018.

Schaefer, Steve. "Tic-Tac-Toe" Jan 2002, mathrec.org/old/2002jan/solutions.html May 2018

Sweigart, Al. *Invent Your Own Computer Games with Python*. San Francisco; No Starch Press, 2017

Turing, Alan. "Digital Computers Applied to Games." *Faster Than Thought (A Symposium on Digital Computing Machines)*, May 1953

Zaslavsky, C. (1982). *Tic-Tac-Toe and Other Three in a Row Games from Ancient Egypt to the Modern Computer*. New York; HarperCollins, 1982