Yahtzee Strategy Analysis

Using Simulation to Evaluate the Performance of Various Gameplay Strategies

Eric J. Wissner (Group 48)

Abstract

This paper presents a simulation-based analysis of gameplay strategies in the classic dice game, Yahtzee. The study explores two novel approaches that attempt to leverage bonus opportunities in the game, namely the Upper Focus strategy and the Yahtzee Focus strategy. The simulation models different variants of these strategies alongside a standard Dice-Driven approach and a One-Roll baseline. The results indicate that the Dice-Driven strategy outperforms all other approaches, including the two bonus-focused strategies. The paper provides detailed methodology and output analysis, including input validation and statistical testing of the simulated data. These findings demonstrate that, despite their potential benefits, the bonus-focused strategies do not represent viable alternatives to traditional gameplay in Yahtzee.

Background and Description of Problem

According to Board Game Geek, Yahtzee is a classic turn-based dice game first introduced in 1956 (BoardGameGeek, n.d.). It is played with five standard dice, with values on each from one to six. During each turn, a player may take up to three rolls (electing to keep or roll each die at their discretion) to earn scores in one of thirteen categories. At the end of the turn, the player must record a score in one of the categories – taking a zero in the category of their choice if the result of their final roll allows no viable scoring options. The highest scoring category is called "Yahtzee" and it is earned by the player ending their turn with five of the same dice value. Figure 1 shows a sample scorecard.

Figure 1: Sample scorecard (The Yahtzee Manifesto, n.d.)

Although the directions are simple enough, the variety of categories forces the player to make decisions before each roll about which dice to hold and then later as to where to record the points from their turn. For instance, while several of the categories are based on how many of a specific value the player has at the end of their turn, a couple categories (Small Straight and Large Straight) instead require a sequence of four or five dice, respectively.

There are two types of bonuses to also consider: the Upper Section bonus and Yahtzee bonuses. As the name implies, the Upper Section of categories is at the top of the page and has a category for each value from one to six. The player can earn

points for the sum of the dice that match the respective category. For instance, with a final roll of [1, 3, 3, 4, 4], the player could earn one point in the "Ones" category, six points in the "Threes" category (3 + 3), or eight points in the "Fours" category (4 + 4). At the end of the game, the player will receive a 35-point bonus if the sum of the scores in the Upper Section is greater than or equal to 63 points. This is not an arbitrary threshold; but instead represents the score the player would achieve with exactly three of the correct dice for each of the Upper Section categories.

A Yahtzee bonus of 100 points is earned for every Yahtzee scored after the first one. Additionally, having five of the same value dice puts the player in a good position to earn a high score in one of the normal categories as well.

Given those opportunities for bonus points, this analysis seeks to determine if either or both of two focused strategies can outperform a standard strategy based on the results of each roll and the categories remaining at that point in time. The first novel approach is called Upper Focus and compels the player to maximize points in the Upper Section, allowing them to earn the associated bonus. Recognizing that the bonus is achieved with, on average, three of each of those category's dice values, this approach will at times take zeroes in the other categories to preserve the opportunity for the bonus. Once the categories in the Upper Section have been completed, the strategy then reverts to a standard dice-driven strategy.

The other special strategy is called Yahtzee Focus and strives to maximize the player's score by achieving as many Yahtzees as possible. In this strategy, the Small and Large Straight categories (worth 30 and 40 points, respectively) are effectively ignored in the interest of earning the 50-point Yahtzee and 100-point bonuses.

Methodology

Due to the complexity of the game and the branching decisions facing a player on each roll, an analytic solution for comparing Yahtzee strategies could not be easily derived. Instead, simulation was used to model the game environment, to codify the logic for each strategy, to record the scores for each turn and game played, and to ultimately compare the performances of the different approaches.

Each of the following strategies and strategy variants were modeled and included in the simulation:

- **Dice-driven strategy**. This approach represents standard gameplay.
- **Upper Focus – High**. This version of the Upper Focus strategy uses the higher valued dice when two otherwise equal choices exist.
- **Upper Focus – Low**. This version of the Upper Focus strategy uses the lower valued dice when two equal otherwise equal choices exist.
- **Yahtzee Focus – High**. This version of the Yahtzee Focus strategy uses the higher valued dice when two otherwise equal choices exist.
- **Yahtzee Focus – Low**. This version of the Yahtzee Focus strategy uses the lower valued dice when two otherwise equal choices exist.
- **One Roll**. This approach was included as a baseline of sorts; it uses the dice from the first roll for the entire turn.

Item 1 in the Appendix contains the flowchart created to model the flow of a game of Yahtzee. The Upper Focus, Yahtzee Focus, and One Roll strategies were straightforward enough to include in this overall model. The logic for which dice to hold in the Dice Driven approach, however, required its own flowchart, which is illustrated in Item 2. Decisions involving where to record scores were based on a Default Priority Scoring list. This list, along with the official Joker Scoring rules (used when multiple Yahtzees are rolled), are included in Item 3.

A Python program (version 3.7) was developed using the PyCharm Community Edition IDE to simulate 10,000 games using each of the strategy variants. To ensure each strategy faced the same dice at the start of each turn, the technique of common random numbers was employed. Instead of generating random dice values as needed within each strategy's gameplay, a series of 15 random dice values was generated and set aside for use during each turn. In many cases, this meant that some of the reserved dice values were not used. This was acceptable as the common dice values across strategies were expected to return smaller confidence intervals when evaluating the differences between average scores for each strategy.

The program, named YahtzeeSim.py, is included in the project deliverables (with code included in the Appendix) and can be executed using a local Python environment. Runtime parameters include choice of random seed, number of simulations to run, the strategy to use, and the choice of a high or low tiebreaker as described earlier.

The information in Figure 2 is available using the program's "help" parameter (-h) and guides the user on how to specify those parameters when running the program:

```
> python YahtzeeSim.py -h
usage: YahtzeeSim.py [-h] [-s SEED] [-g GAMES] [-o | -y | -u | -d]
                     [--high | --low]

This program will simulate Yahtzee games using the strategy of your choice
(see below for options). The average scores for each slot will be displayed
along with the average total score. Several log files will be saved to the
working directory to view details or for additional analysis.

optional arguments:
  -h, --help            show this help message and exit
  -s SEED, --seed SEED  Integer used for random seed. (default: 290)
  -g GAMES, --games GAMES
                        Number of games to simulate. (default: 1000)
  -o, --oneroll         Baseline game; only one roll per turn (default: False)
  -y, --yahtzee         Go for Yahtzee every turn (default: False)
  -u, --upper           The Upper Section is the first priority (default:
                        False)
  -d, --dicedriven      Default strategy; Choices driven by dice rolled
                        (default: True)
  --high                Given the choice between dice, go with the higher
                        value (default: True)
  --low                 Given the choice between dice, go with the lower value
                        (default: False)
```

Figure 2: YahtzeeSim.py help output

For each series of simulated games, the following five log files were generated:

- **Log <Strategy> Sets of Dice.** The collection of 15 dice values generated and available for each game and turn.
- **Log <Strategy> Dice Values.** Each individual dice value rolled by game, turn, and roll.
- **Log <Strategy> Rolls.** For each game, turn, and roll, the dice "on the table", the hold/roll decisions for each die, and the strategy justification for those decisions.
- **Log <Strategy> Scoring.** The results of each scoring decision by game and turn; including final dice values, category, and points earned.
- **Log <Strategy> Master Score Sheet.** The points scored in each category and bonus slot by game.

Data from these files were used to validate the models and the code itself and then were used to support the necessary input and output analysis for the project. A sample set of log files for the Dice Driven approach are included in the project deliverables.

Main Findings

Input Analysis

The dice values generated within the Python program are *pseudo* random numbers. When dealing with pseudo random numbers, it is important to perform input analysis to ensure that the data used by a simulation accurately reflect the environment being modeled. For Yahtzee gameplay, that means ensuring the concept of fair dice – that any number from one to six had an equal chance of being rolled. It also means that the dice values should be independent without patterns associated with when that equal distribution of dice values was generated.

For each series of simulated games, the same set of 1,950,000 dice values were generated (15 dice reserved for 13 turns within 10,000 games). Figure 3 displays the distribution of those dice values.

Figure 3: Distribution of generated dice values

Although a visual inspection suggests the dice values were approximately equally distributed, a chi-square goodness-of-fit test was conducted to confirm that expectation. Using the formula below, with k=6 discrete possible values and E_i=325,000 observations for each value, the chi-squared goodness-of-fit test statistic was calculated to be 2.77.

$$x_0^2 \equiv \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i} = 2.76544$$

Evaluating that statistic against the chi-square statistic at α=0.05 and 5 degrees of freedom (11.07), the null hypothesis that the observed distribution of dice values did not match the expected distribution of dice values was rejected.

Next, a Wald-Wolfowitz Runs Test (above and below the mean) was conducted using the `runs.test` function in R to determine if the dice values were independent. With a p-value of 0.8489, the null hypothesis of randomness could not be rejected, indicating that the dice values were generated in a random manner.

Output Analysis

Before the performance of the different strategies could be evaluated and compared, the output from the simulations needed to be reviewed, particularly to understand if the data were independent, identically distributed, and normally distributed. Within each series of simulated games, the final scores were independent in that they did not depend on the previous game's score, nor were they impacted by it. Furthermore, because the probabilities of the six dice values did not change from game to game, their scores were also identically distributed and did not change over time.

Unfortunately, as illustrated in Figure 4, the scores over the 10,000 games were not normally distributed. The One Roll approach was closest to normal; but an Anderson-Darling Test confirmed that its scores were in fact not normally distributed (p-value = $2.2e^{-16}$).

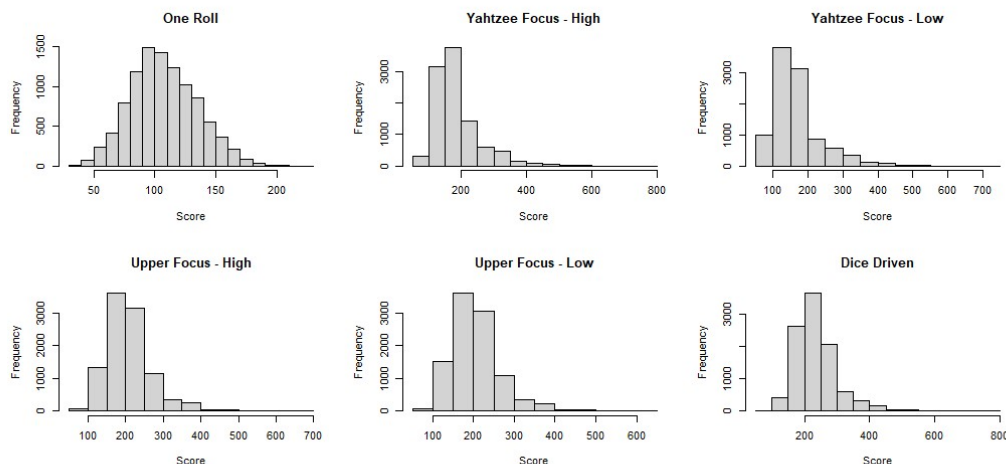


Figure 4: Distribution of scores by strategy

The Central Limit Theorem can be used to account for this and to help determine more reliable confidence intervals for the various strategies' scores. Accordingly, the 10,000 games were divided into 400 batches of 25 games each. Figure 5 displays the resulting (more normally distributed) histograms for each approach.

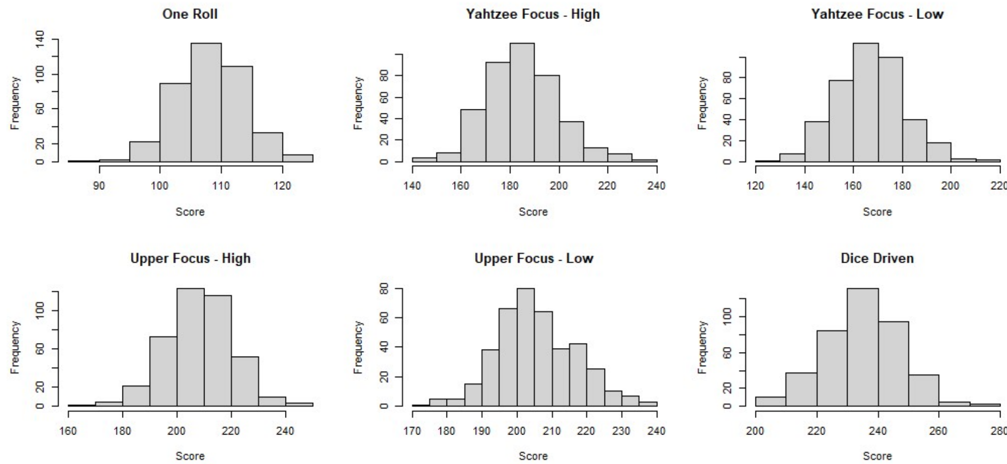


Figure 5: Distribution of batched scores by strategy

Using the following formulas for calculating the sample mean ($\bar{Z}_{400}$) and sample variance (S_Z^2) for each strategy across the 400 batches (Figure 6), the 95% confidence intervals were calculated using $t_{0.025,399} = 1.966$ and are visualized in Figure 7 and listed in Figure 8.

$$\bar{Z}_{400} = \frac{1}{400} * \sum_{i=1}^{400} Z_i \qquad S_Z^2 = \frac{1}{399} * \sum_{i=1}^{400} (Z_i - \bar{Z}_{400})^2$$

Figure 6: Calculating sample mean and variance using the mean scores from the 400 batches

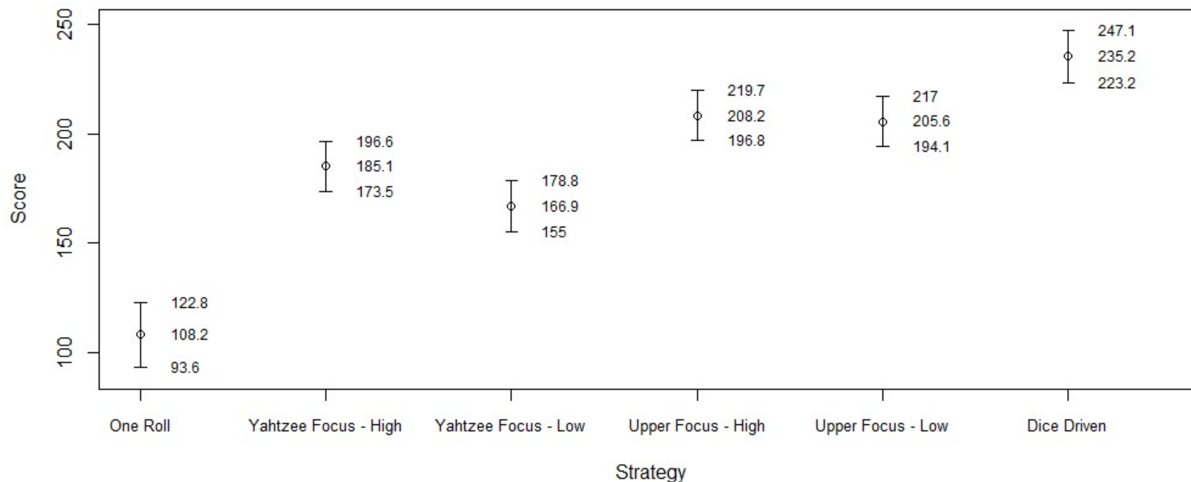


Figure 7: 95% confidence intervals by strategy

Strategy	Mean Batch Score
One Roll	108.2 ± 14.6
Yahtzee Focus - High	185.1 ± 11.6
Yahtzee Focus - Low	166.9 ± 11.9
Upper Focus - High	208.2 ± 11.5
Upper Focus - Low	205.6 ± 11.4
Dice Driven	235.2 ± 11.9

Figure 8: Point estimates and margins of error (95% CI) by strategy

The Yahtzee Focus and Upper Focus strategies did not appear to return higher scores than the Dice Driven approach, despite focusing on the bonuses. On the contrary, all four novel strategies as well as the One Roll baseline approach seemed to return scores that were in fact *lower* than the standard approach.

To compare the performance of the different strategies more definitively, a ranking and selection procedure was employed to understand which approach was more likely to return the highest score. Because each strategy faced the same sets of dice for each turn within each game (recall the use of common random numbers), the game-by-game scores for each approach were first compared to determine a "winner" for each of the 10,000 simulated games, effectively having the strategies compete against each other. A single stage procedure to select the most probable winner was then able to be utilized on this multinomial selection problem.

In the procedure based on the work of Bechhofer, Elmaghraby, and Morse (1959), a certain number of observations (games) were evaluated based on the number of strategies (k), the specified probability requirement ($P^\star$), and the smallest ratio of the best and second-best strategy probabilities ($\theta^\star$). The One Roll baseline model was not included in the procedure, meaning k equaled 5. A 95% overall probability requirement was used and a ratio of 1.4 was used for $\theta^\star$. Given these inputs, the procedure recommended a sample size of 374 games, the results of which are displayed in Figure 9.

Strategy	Games Won
Dice Driven	210
Upper Focus - High	70
Upper Focus - Low	51
Yahtzee Focus - High	29
Yahtzee Focus - Low	14

Figure 9: Results of Single Stage Multinomial Procedure

Based on these results, the Dice Driven strategy can be considered the best approach and, by extension, it can be concluded that the novel strategies do not represent viable alternatives to traditional gameplay despite their focus on bonus opportunities.

Conclusion

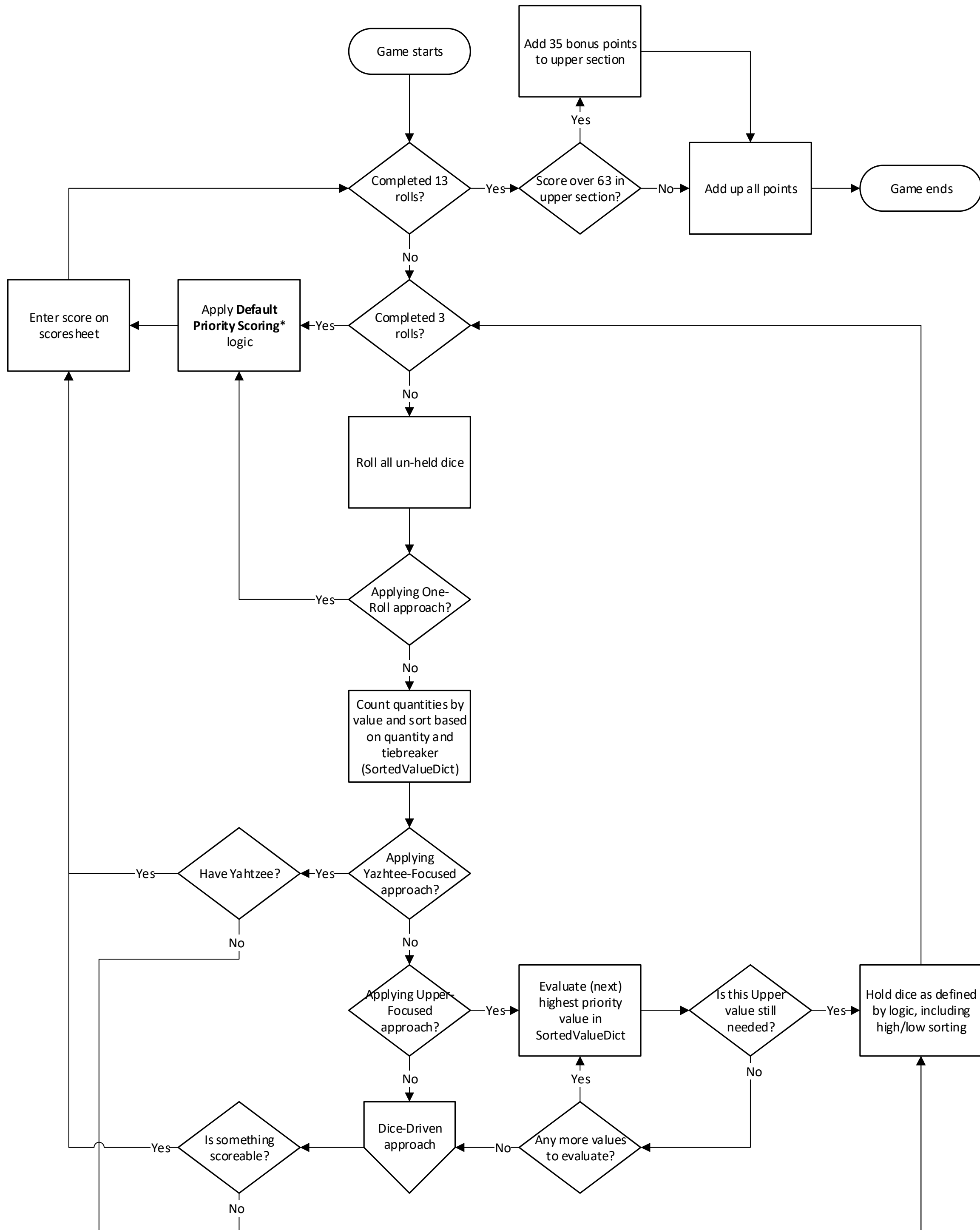
Despite its age, Yahtzee remains a compelling game. It is easy to learn but offers its players challenges associated with the randomness of the dice, the differences in the scoring categories, and the opportunities for bonus points. These factors require thoughtful decision-making, boldness, and ultimately a little bit of luck.

This analysis explored two types of alternative gameplay strategies that try to exploit the two bonuses in the game: Yahtzee-focused strategies and Upper Section-focused strategies. Despite logic intended to "ensure" the bonuses are achieved, these novel approaches cannot consistently outscore a standard Dice Driven strategy.

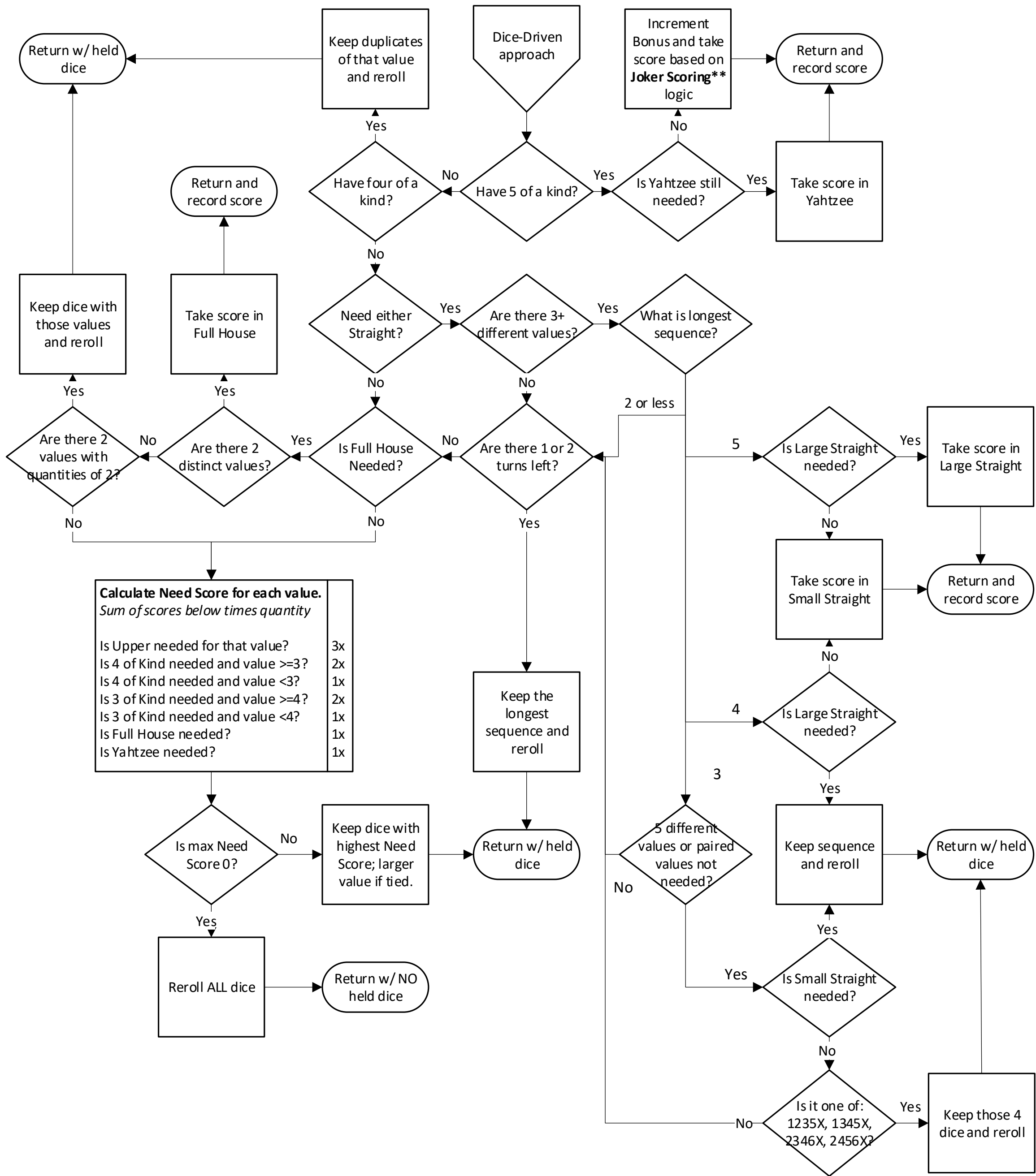
As games of Yahtzee are typically played with other players, with scoring choices for all players publicly known, it might be interesting for future analyses on this topic to include elements of game theory. Would overall strategies need to be flexible considering what other players do? Would the response be subtle, perhaps in terms of the scoring priorities?

Appendix

Item 1 – Game Flow Logic



Item 2 – Dice Driven Strategy Logic



Item 3 – Default Priority and Joker Scoring

***Default Priority Scoring**
Take the first of these that apply

Yahtzee (50) or Yahtzee Bonus (100) + Joker Scoring**
Large Straight (40)
Small Straight (30)
Full House (25)
Upper 6's; if 3 or more (sum of 6's)
Upper 5's; if 3 or more (sum of 5's)
Upper 4's; if 3 or more (sum of 4's)
Upper 3's; if 3 or more (sum of 3's)
Upper 2's; if 3 or more (sum of 2's)
Upper 1's; if 3 or more (sum of 1's)
4 of a Kind (sum of all dice)
3 of a Kind (sum of all dice)
Chance; if >= 15 points (sum of all dice)
Upper 1's; if any (sum of 1's)
Upper 2's; if any (sum of 2's)
Upper 3's; if any (sum of 3's)
Upper 4's; if any (sum of 4's)
Upper 5's; if any (sum of 5's)
Upper 6's; if any (sum of 6's)
Chance
Large Straight (take a 0)
Small Straight (take a 0)
Full House (take a 0)
4 of a Kind (take a 0)
3 of a Kind (take a 0)
Upper 1's (take a 0)
Upper 2's (take a 0)
Upper 3's (take a 0)
Upper 4's (take a 0)
Upper 5's (take a 0)
Upper 6's (take a 0)
Yahtzee (take a 0)

****Joker Scoring**
Take the first of these that apply

Upper Section Category
3 of a Kind
4 of a Kind
Full House
Small Straight
Large Straight
Chance
Upper 1's (take a 0)
Upper 2's (take a 0)
Upper 3's (take a 0)
Upper 4's (take a 0)
Upper 5's (take a 0)
Upper 6's (take a 0)

Citations

Bechhofer, R. E., S. A. Elmaghraby, and N. Morse (1959). A single-sample multiple-decision procedure for selecting the multinomial event which has the largest probability. Ann. Math. Statist. 30 102-119

Yahtzee. BoardGameGeek. (n.d.). Retrieved from https://boardgamegeek.com/boardgame/2243/yahtzee

Python Code

```
import argparse
parser = argparse.ArgumentParser(description="This program will simulate Yahtzee games using the strategy of
your choice (see below for options).  The average scores for each slot will be displayed along with the average
total score.  Several log files will be saved to the working directory to view details or for additional
analysis.",
                                 formatter_class=argparse.ArgumentDefaultsHelpFormatter)
seedparser=parser.add_argument("-s", "--seed", help='Integer used for random seed.', type = int, default=290)
gamesparser=parser.add_argument("-g", "--games", help='Number of games to simulate.', type = int, default=1000)
groupapproach = parser.add_mutually_exclusive_group()
groupapproach.add_argument( "-o", "--oneroll", action="store_true", help='Baseline game; only one roll per
turn')
groupapproach.add_argument( "-y", "--yahtzee", action="store_true", help='Go for Yahtzee every turn')
groupapproach.add_argument( "-u", "--upper", action="store_true", help='The Upper Section is the first
priority')
groupapproach.add_argument( "-d", "--dicedriven", action="store_false", help='Default strategy; Choices driven
by dice rolled')
grouptiebreaker = parser.add_mutually_exclusive_group()
grouptiebreaker.add_argument("--high", action="store_false", help='Given the choice between dice, go with the
higher value')
grouptiebreaker.add_argument("--low", action="store_true", help='Given the choice between dice, go with the
lower value')
args = parser.parse_args()
config = vars(args)

Approach = 'Dice Driven'
Tiebreaker = 'High'
SeedtoUse = args.seed
GamesToPlay = args.games

if args.low:
    Tiebreaker = 'Low'

if args.oneroll:
    Approach="One Roll"
if args.yahtzee:
    Approach='Yahtzee Focused'
if args.upper:
    Approach = 'Upper Focused'

if Approach == 'One Roll' or Approach == 'Dice Driven':
    ApproachName = Approach
else:
    ApproachName = Approach + ' ' + Tiebreaker

from collections import Counter
import random
import pandas as pd

import datetime
ct= datetime.datetime.now()
ct = str(ct).replace(':', '-')
ct = ct.replace('.', '-')

MasterScoreSheet = pd.DataFrame(columns = ['Approach', 'Game', 'Ones', 'Twos', 'Threes', 'Fours', 'Fives',
'Sixes', 'UpperBonus', 'TofK', 'FofK', 'FH', 'SS', 'LS', 'Yahtzee', 'Chance', 'YBonus', 'Total'])
Log_Scoring = []
Log_Rolls = []
Log_DiceValues = []
Log_SetsofDice = []

ThisScoreSheet = {}
ThisScoreFlags = {}

ValueNames = {1:'Ones', 2:'Twos', 3:'Threes', 4:'Fours', 5:'Fives', 6:'Sixes'}
DiceForTurn = []
DicePointer = 0

random.seed(SeedtoUse)


def GetNewSetofDice():
    DiceForTurn=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
    for dnum in range(0, 15):
        DiceForTurn[dnum] = random.randint(1, 6)
```

```
    return DiceForTurn


def RandomRoll(StartingDice, DiceHeld, DiceForTurn, DicePointer):
    DiceArrow = DicePointer
    EndingDice = StartingDice
    for d in range(0, 5):
        if not DiceHeld[d]:
            EndingDice[d] = DiceForTurn[DiceArrow]          # random.randint(1, 6)
            DiceArrow = DiceArrow + 1
            Log_DiceValues.append([Game, Turn, Roll, EndingDice[d]])
    return (EndingDice, DiceArrow)


def GetSequenceInfo(InputValues):
    SortedVal = sorted(InputValues)
    BestLen = 0
    BestStartVal = 0
    BestEndVal = 0
    Leni = 1
    StartVal = SortedVal[0]
    EndVal = SortedVal[0]

    for i in range(1,5):
        if SortedVal[i] > SortedVal[i-1]:
            if SortedVal[i] == SortedVal[i-1] + 1:
                Leni = Leni + 1
                EndVal = SortedVal[i]
            else:
                if Leni > BestLen:
                    BestLen = Leni
                    BestStartVal = StartVal
                    BestEndVal = EndVal
                Leni = 1
                StartVal = SortedVal[i]
                EndVal = SortedVal[i]

    if Leni > BestLen:
        BestLen = Leni
        BestStartVal = StartVal
        BestEndVal = EndVal

    return BestLen, BestStartVal, BestEndVal


def DefaultPriorityScoring(Game, Turn, Roll, DiceValue, ValueDict):
    FiveDice = DiceValue
    if max(ValueDict.values()) == 5:
        if not ThisScoreFlags.get('Yahtzee', False):
            ThisScoreSheet['Yahtzee'] = 50
            ThisScoreFlags['Yahtzee'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Yahtzee', 50])
            return
        else:
            if ThisScoreSheet['Yahtzee'] == 50:
                ThisScoreSheet['YBonus'] = ThisScoreSheet.get('YBonus', 0) + 100
                Log_Scoring.append([Game, Turn, Roll, FiveDice, 'YBonus', 100])
                ## Joker Rules...
                YahtzeeValue = max(ValueDict, key=ValueDict.get)
                YahtzeeName = ValueNames[YahtzeeValue]
                if not ThisScoreFlags.get(YahtzeeName, False):
                    ThisScoreSheet[YahtzeeName] = YahtzeeValue * 5
                    ThisScoreFlags[YahtzeeName] = True
                    Log_Scoring.append([Game, Turn, None, FiveDice, YahtzeeName, YahtzeeValue * 5])
                    return
                if not ThisScoreFlags.get('TofK', False):
                    ThisScoreSheet['TofK'] = YahtzeeValue * 5
                    ThisScoreFlags['TofK'] = True
                    Log_Scoring.append([Game, Turn, None, FiveDice, 'TofK', YahtzeeValue * 5])
                    return
                if not ThisScoreFlags.get('FofK', False):
                    ThisScoreSheet['FofK'] = YahtzeeValue * 5
                    ThisScoreFlags['FofK'] = True
                    Log_Scoring.append([Game, Turn, None, FiveDice, 'FofK', YahtzeeValue * 5])
                    return
                if not ThisScoreFlags.get('FH', False):
                    ThisScoreSheet['FH'] = 25
                    ThisScoreFlags['FH'] = True
                    Log_Scoring.append([Game, Turn, None, FiveDice, 'FH', 25])
```

```
                return
            if not ThisScoreFlags.get('SS', False):
                ThisScoreSheet['SS'] = 30
                ThisScoreFlags['SS'] = True
                Log_Scoring.append([Game, Turn, None, FiveDice, 'SS', 30])
                return
            if not ThisScoreFlags.get('LS', False):
                ThisScoreSheet['LS'] = 40
                ThisScoreFlags['LS'] = True
                Log_Scoring.append([Game, Turn, None, FiveDice, 'LS', 40])
                return
            if not ThisScoreFlags.get('Chance', False):
                ThisScoreSheet['Chance'] = YahtzeeValue * 5
                ThisScoreFlags['Chance'] = True
                Log_Scoring.append([Game, Turn, None, FiveDice, 'Chance', YahtzeeValue * 5])
                return
            for bonfill in range(1, 7):
                bonname = ValueNames[bonfill]
                if not ThisScoreFlags.get(bonname, False):
                    ThisScoreSheet[bonname] = 0
                    ThisScoreFlags[bonname] = True
                    Log_Scoring.append([Game, Turn, None, FiveDice, bonname, 0])
                    break
            return
    if Approach == 'Upper Focused':
        if not ThisScoreFlags.get('Sixes', False):
            if ValueDict.get(6, 0) >= 3:
                ThisScoreSheet['Sixes'] = ValueDict[6] * 6
                ThisScoreFlags['Sixes'] = True
                Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Sixes', ValueDict[6] * 6])
                return
        if not ThisScoreFlags.get('Fives', False):
            if ValueDict.get(5, 0) >= 3:
                ThisScoreSheet['Fives'] = ValueDict[5] * 5
                ThisScoreFlags['Fives'] = True
                Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Fives', ValueDict[5] * 5])
                return
        if not ThisScoreFlags.get('Fours', False):
            if ValueDict.get(4, 0) >= 3:
                ThisScoreSheet['Fours'] = ValueDict[4] * 4
                ThisScoreFlags['Fours'] = True
                Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Fours', ValueDict[4] * 4])
                return
        if not ThisScoreFlags.get('Threes', False):
            if ValueDict.get(3, 0) >= 3:
                ThisScoreSheet['Threes'] = ValueDict[3] * 3
                ThisScoreFlags['Threes'] = True
                Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Threes', ValueDict[3] * 3])
                return
        if not ThisScoreFlags.get('Twos', False):
            if ValueDict.get(2, 0) >= 3:
                ThisScoreSheet['Twos'] = ValueDict[2] * 2
                ThisScoreFlags['Twos'] = True
                Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Twos', ValueDict[2] * 2])
                return
        if not ThisScoreFlags.get('Ones', False):
            if ValueDict.get(1, 0) >= 3:
                ThisScoreSheet['Ones'] = ValueDict[1] * 1
                ThisScoreFlags['Ones'] = True
                Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Ones', ValueDict[1] * 1])
                return
    if not ThisScoreFlags.get('LS', False):
        if GetSequenceInfo(DiceValue)[0] == 5:
            ThisScoreSheet['LS'] = 40
            ThisScoreFlags['LS'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'LS', 40])
            return
    if not ThisScoreFlags.get('SS', False):
        if GetSequenceInfo(DiceValue)[0] >= 4:
            ThisScoreSheet['SS'] = 30
            ThisScoreFlags['SS'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'SS', 30])
            return
    if not ThisScoreFlags.get('FH', False):
        if max(ValueDict.values())==3 and min(ValueDict.values())==2:
            ThisScoreSheet['FH'] = 25
            ThisScoreFlags['FH'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'FH', 25])
            return
```

```
    if not ThisScoreFlags.get('Sixes', False):
        if ValueDict.get(6, 0) >= 3:
            ThisScoreSheet['Sixes'] = ValueDict[6] * 6
            ThisScoreFlags['Sixes'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Sixes', ValueDict[6] * 6])
            return
    if not ThisScoreFlags.get('Fives', False):
        if ValueDict.get(5, 0) >= 3:
            ThisScoreSheet['Fives'] = ValueDict[5] * 5
            ThisScoreFlags['Fives'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Fives', ValueDict[5] * 5])
            return
    if not ThisScoreFlags.get('Fours', False):
        if ValueDict.get(4, 0) >= 3:
            ThisScoreSheet['Fours'] = ValueDict[4] * 4
            ThisScoreFlags['Fours'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Fours', ValueDict[4] * 4])
            return
    if not ThisScoreFlags.get('Threes', False):
        if ValueDict.get(3, 0) >= 3:
            ThisScoreSheet['Threes'] = ValueDict[3] * 3
            ThisScoreFlags['Threes'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Threes', ValueDict[3] * 3])
            return
    if not ThisScoreFlags.get('Twos', False):
        if ValueDict.get(2, 0) >= 3:
            ThisScoreSheet['Twos'] = ValueDict[2] * 2
            ThisScoreFlags['Twos'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Twos', ValueDict[2] * 2])
            return
    if not ThisScoreFlags.get('Ones', False):
        if ValueDict.get(1, 0) >= 3:
            ThisScoreSheet['Ones'] = ValueDict[1] * 1
            ThisScoreFlags['Ones'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Ones', ValueDict[1] * 1])
            return
    if not ThisScoreFlags.get('FofK', False):
        if max(ValueDict.values()) >= 4:
            ThisScoreSheet['FofK'] = sum(DiceValue)
            ThisScoreFlags['FofK'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'FofK', sum(DiceValue)])
            return
    if not ThisScoreFlags.get('TofK', False):
        if max(ValueDict.values()) >= 3:
            ThisScoreSheet['TofK'] = sum(DiceValue)
            ThisScoreFlags['TofK'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'TofK', sum(DiceValue)])
            return
    if not ThisScoreFlags.get('Chance', False):
        if sum(DiceValue) >= 15:
            ThisScoreSheet['Chance'] = sum(DiceValue)
            ThisScoreFlags['Chance'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Chance', sum(DiceValue)])
            return
    if not ThisScoreFlags.get('Ones', False):
        if ValueDict.get(1, 0) > 0:
            ThisScoreSheet['Ones'] = ValueDict[1] * 1
            ThisScoreFlags['Ones'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Ones', ValueDict[1] * 1])
            return
    if not ThisScoreFlags.get('Twos', False):
        if ValueDict.get(2, 0) > 0:
            ThisScoreSheet['Twos'] = ValueDict[2] * 2
            ThisScoreFlags['Twos'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Twos', ValueDict[2] * 2])
            return
    if not ThisScoreFlags.get('Threes', False):
        if ValueDict.get(3, 0) > 0:
            ThisScoreSheet['Threes'] = ValueDict[3] * 3
            ThisScoreFlags['Threes'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Threes', ValueDict[3] * 3])
            return
    if not ThisScoreFlags.get('Fours', False):
        if ValueDict.get(4, 0) > 0:
            ThisScoreSheet['Fours'] = ValueDict[4] * 4
            ThisScoreFlags['Fours'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Fours', ValueDict[4] * 4])
            return
    if not ThisScoreFlags.get('Fives', False):
```

```
        if ValueDict.get(5, 0) > 0:
            ThisScoreSheet['Fives'] = ValueDict[5] * 5
            ThisScoreFlags['Fives'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Fives', ValueDict[5] * 5])
            return
    if not ThisScoreFlags.get('Sixes', False):
        if ValueDict.get(6, 0) > 0:
            ThisScoreSheet['Sixes'] = ValueDict[6] * 6
            ThisScoreFlags['Sixes'] = True
            Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Sixes', ValueDict[6] * 6])
            return
    if not ThisScoreFlags.get('Chance', False):
        ThisScoreSheet['Chance'] = sum(DiceValue)
        ThisScoreFlags['Chance'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Chance', sum(DiceValue)])
        return
    if not ThisScoreFlags.get('LS', False):
        ThisScoreSheet['LS'] = 0
        ThisScoreFlags['LS'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'LS', 0])
        return
    if not ThisScoreFlags.get('SS', False):
        ThisScoreSheet['SS'] = 0
        ThisScoreFlags['SS'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'SS', 0])
        return
    if not ThisScoreFlags.get('FH', False):
        ThisScoreSheet['FH'] = 0
        ThisScoreFlags['FH'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'FH', 0])
        return
    if not ThisScoreFlags.get('FofK', False):
        ThisScoreSheet['FofK'] = 0
        ThisScoreFlags['FofK'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'FofK', 0])
        return
    if not ThisScoreFlags.get('TofK', False):
        ThisScoreSheet['TofK'] = 0
        ThisScoreFlags['TofK'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'TofK', 0])
        return
    if not ThisScoreFlags.get('Ones', False):
        ThisScoreSheet['Ones'] = 0
        ThisScoreFlags['Ones'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Ones', 0])
        return
    if not ThisScoreFlags.get('Twos', False):
        ThisScoreSheet['Twos'] = 0
        ThisScoreFlags['Twos'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Twos', 0])
        return
    if not ThisScoreFlags.get('Threes', False):
        ThisScoreSheet['Threes'] = 0
        ThisScoreFlags['Threes'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Threes', 0])
        return
    if not ThisScoreFlags.get('Fours', False):
        ThisScoreSheet['Fours'] = 0
        ThisScoreFlags['Fours'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Fours', 0])
        return
    if not ThisScoreFlags.get('Fives', False):
        ThisScoreSheet['Fives'] = 0
        ThisScoreFlags['Fives'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Fives', 0])
        return
    if not ThisScoreFlags.get('Sixes', False):
        ThisScoreSheet['Sixes'] = 0
        ThisScoreFlags['Sixes'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Sixes', 0])
        return
    if not ThisScoreFlags.get('Yahtzee', False):
        ThisScoreSheet['Yahtzee'] = 0
        ThisScoreFlags['Yahtzee'] = True
        Log_Scoring.append([Game, Turn, Roll, FiveDice, 'Yahtzee', 0])
        return
    return
```

```
for Game in range(1, GamesToPlay + 1):
    ThisScoreSheet = {}
    ThisScoreFlags = {}
    for Turn in range(1,14):
        Roll = 0
        StillRolling = True
        DiceForTurn = GetNewSetofDice()
        Log_SetsofDice.append([Game, Turn, DiceForTurn])
        DicePointer = 0
        DiceValue = [0, 0, 0, 0, 0]
        DiceHold = [False, False, False, False, False]
        while StillRolling:
            Roll = Roll + 1
            holdreason = ''
            DiceValue, DicePointer = RandomRoll(DiceValue, DiceHold, DiceForTurn, DicePointer)
            PublicDiceValue = DiceValue[:]
            DiceHold = [False, False, False, False, False]
            ValueDict = {}
            ValueDict = Counter(DiceValue)
            if Tiebreaker == 'Low':
                sortedValueDict = {val[0]: val[1] for val in sorted(ValueDict.items(), key=lambda x: (-x[1], 
x[0]))}
            else:
                sortedValueDict = {val[0]: val[1] for val in sorted(ValueDict.items(), key=lambda x: (-x[1], -
x[0]))}
            DecisionMade = False
            if Approach == 'One Roll':
                StillRolling = False
                DecisionMade = True
            if Approach == 'Yahtzee Focused':
                FocusVal = list(sortedValueDict.keys())[0]
                FocusName = ValueNames[FocusVal]
                for holdi in range(0, 5):
                    if DiceValue[holdi] == FocusVal:
                        DiceHold[holdi] = True
                        holdreason = 'Going for Yahtzee'
                if Roll == 3 or max(ValueDict.values()) == 5:
                    StillRolling = False
                DecisionMade = True
            if Approach == 'Upper Focused':
                for FocusVal in sortedValueDict.keys():
                    FocusName = ValueNames[FocusVal]
                    if not ThisScoreFlags.get(FocusName, False):
                        DecisionMade = True
                        for holdi in range(0, 5):
                            if DiceValue[holdi] == FocusVal:
                                DiceHold[holdi] = True
                                holdreason = 'Upper focused: ' + str(FocusVal)
                        break
                if Roll == 3 or max(ValueDict.values()) == 5:
                    StillRolling = False
            if Approach == 'Dice Driven' or not DecisionMade:
                if max(ValueDict.values()) == 5:
                    StillRolling = False
                    DecisionMade = True
                else:
                    if max(ValueDict.values()) == 4:
                        FocusVal = max(ValueDict, key=ValueDict.get)
                        FocusName = ValueNames[FocusVal]
                        for holdi in range(0, 5):
                            if DiceValue[holdi] == FocusVal:
                                DiceHold[holdi] = True
                                holdreason = 'Have four of a kind'
                        DecisionMade = True
                    else:
                        if not ThisScoreFlags.get('LS', False) or not ThisScoreFlags.get('SS', False):
                            if len(ValueDict) >= 3:
                                SeqLen = GetSequenceInfo(DiceValue)[0]
                                if SeqLen == 5:
                                    StillRolling = False
                                    DecisionMade = True
                                if SeqLen == 4:
                                    DecisionMade = True
                                    if ThisScoreFlags.get('LS', False):
                                        StillRolling = False
                                    else:
                                        for seqnum in range(GetSequenceInfo(DiceValue)[1], 
GetSequenceInfo(DiceValue)[2]+1):
                                            for holdi in range(0, 5):
```

```
                                                if DiceValue[holdi] == seqnum:
                                                    DiceHold[holdi] = True
                                                    holdreason = 'Going for LS; Seq=4'
                                                    break
                                if SeqLen == 3 and (max(ValueDict.values()) < 2 or
ThisScoreFlags.get(ValueNames[max(ValueDict, key=ValueDict.get)], False)):
                                    if not ThisScoreFlags.get('SS', False):
                                        DecisionMade = True
                                        for seqnum in range(GetSequenceInfo(DiceValue)[1],
GetSequenceInfo(DiceValue)[2]+1):
                                            for holdi in range(0, 5):
                                                if DiceValue[holdi] == seqnum:
                                                    DiceHold[holdi] = True
                                                    holdreason = 'Going for at least SS; Seq=3'
                                                    break
                                    else:
                                        sublists = [[1,2,3,5], [1,3,4,5], [2,3,4,6], [2,4,5,6]]
                                        for sublist in sublists:
                                            if ValueDict.get(sublist[0],0) > 0 and ValueDict.get(sublist[1],0) >
0 and ValueDict.get(sublist[2],0) > 0 and ValueDict.get(sublist[3],0) > 0:
                                                DecisionMade = True
                                                for seqnum in sublist:
                                                    for holdi in range(0, 5):
                                                        if DiceValue[holdi] == seqnum:
                                                            DiceHold[holdi] = True
                                                            holdreason = 'Going for LS; Need inside number'
                                                            break
                            if not DecisionMade and Turn >= 12:
                                DecisionMade = True
                                for seqnum in range(GetSequenceInfo(DiceValue)[1], GetSequenceInfo(DiceValue)[2]
+ 1):
                                    for holdi in range(0, 5):
                                        if DiceValue[holdi] == seqnum:
                                            DiceHold[holdi] = True
                                            holdreason = 'Going for straight; last two turns'
                                            break
                        if not DecisionMade and not ThisScoreFlags.get('FH', False):
                            if len(ValueDict) == 2:
                                DecisionMade = True
                                StillRolling = False
                            ValDictCounter = Counter(ValueDict.values())
                            if ValDictCounter.get(2, 0) == 2:
                                fhsorted = {val[0]: val[1] for val in sorted(ValueDict.items(), key=lambda x: (-
x[1], -x[0]))}
                                highername = ValueNames[list(fhsorted.keys())[0]]
                                if not ThisScoreFlags.get(highername, False):
                                    DecisionMade = True
                                    for holdi in range(0, 5):
                                        if DiceValue[holdi] == list(fhsorted.keys())[0]:
                                            DiceHold[holdi] = True
                                            holdreason = 'Going for Upper instead of FH: ' +
str(list(fhsorted.keys())[0])
                                else:
                                    lowername = ValueNames[list(fhsorted.keys())[1]]
                                    if not ThisScoreFlags.get(lowername, False):
                                        DecisionMade = True
                                        for holdi in range(0, 5):
                                            if DiceValue[holdi] == list(fhsorted.keys())[1]:
                                                DiceHold[holdi] = True
                                                holdreason = 'Going for Upper instead of FH: ' +
str(list(fhsorted.keys())[1])
                                    else:
                                        DecisionMade = True
                                        for holdi in range(0, 5):
                                            if DiceValue[holdi] == list(fhsorted.keys())[0] or DiceValue[holdi]
== list(fhsorted.keys())[1]:
                                                DiceHold[holdi] = True
                                                holdreason = 'Going for FH'
                        if not DecisionMade:
                            NeedScoreDict = {}
                            for EachVal in ValueDict.keys():
                                EachName = ValueNames[EachVal]
                                if not ThisScoreFlags.get(EachName, False):
                                    NeedScoreDict[EachVal] = NeedScoreDict.get(EachVal, 0) + (3 *
ValueDict[EachVal])
                                if not ThisScoreFlags.get('FofK', False):
                                    if EachVal >= 3:
                                        NeedScoreDict[EachVal] = NeedScoreDict.get(EachVal, 0) + (2 *
ValueDict[EachVal])
```

```
                                    else:
                                        NeedScoreDict[EachVal] = NeedScoreDict.get(EachVal, 0) + (1 *
ValueDict[EachVal])
                                if not ThisScoreFlags.get('TofK', False):
                                    if EachVal >= 4:
                                        NeedScoreDict[EachVal] = NeedScoreDict.get(EachVal, 0) + (2 *
ValueDict[EachVal])
                                    else:
                                        NeedScoreDict[EachVal] = NeedScoreDict.get(EachVal, 0) + (1 *
ValueDict[EachVal])
                                if not ThisScoreFlags.get('FH', False):
                                    NeedScoreDict[EachVal] = NeedScoreDict.get(EachVal, 0) + (1 *
ValueDict[EachVal])
                                if not ThisScoreFlags.get('Yahtzee', False):
                                    NeedScoreDict[EachVal] = NeedScoreDict.get(EachVal, 0) + (1 *
ValueDict[EachVal])
                            if len(NeedScoreDict) > 0:
                                DecisionMade = True
                                sortedNeedScoreDict = {val[0]: val[1] for val in sorted(NeedScoreDict.items(),
key=lambda x: (-x[1], -x[0]))}
                                MaxNeedVal = list(sortedNeedScoreDict.keys())[0]
                                for holdi in range(0, 5):
                                    if DiceValue[holdi] == MaxNeedVal:
                                        DiceHold[holdi] = True
                                        holdreason = 'Holding based on Total Need Score for: ' + str(MaxNeedVal)
                if Roll == 3:
                    StillRolling = False
            if StillRolling == True:
                helddicevar = DiceHold
            else:
                helddicevar = []
                holdreason = ''
            Log_Rolls.append([Game, Turn, Roll, PublicDiceValue, helddicevar, holdreason])
        DefaultPriorityScoring(Game, Turn, Roll, DiceValue, ValueDict)

    if ThisScoreSheet['Ones'] + ThisScoreSheet['Twos'] + ThisScoreSheet['Threes'] + ThisScoreSheet['Fours'] +
ThisScoreSheet['Fives'] + ThisScoreSheet['Sixes'] >= 63:
        ThisScoreSheet['UpperBonus'] = 35
        Log_Scoring.append([Game, None, None, None, 'UpperBonus', 35])
    else:
        ThisScoreSheet['UpperBonus'] = 0
        Log_Scoring.append([Game, None, None, None, 'UpperBonus', 0])
    ThisScoreSheet['YBonus'] = ThisScoreSheet.get('YBonus', 0)
    thistotal = sum(ThisScoreSheet.values())
    ThisScoreSheet['Total'] = thistotal
    ThisScoreSheet['Approach'] = ApproachName
    ThisScoreSheet['Game'] = Game
    ThisScoreSheetDF = pd.DataFrame([ThisScoreSheet])
    MasterScoreSheet = MasterScoreSheet.append(ThisScoreSheetDF)

pd.set_option('max_columns', None)

SetsofDiceDF = pd.DataFrame(Log_SetsofDice)
SetsofDiceDF.columns = ['Game', 'Turn', 'SetofDice']
SetsofDiceDF.to_csv('Log - ' + ApproachName + ' - SetsofDice.csv')

DiceValueDF = pd.DataFrame(Log_DiceValues)
DiceValueDF.columns = ['Game', 'Turn', 'Roll', 'DiceValue']
DiceValueDF.to_csv('Log - ' + ApproachName + ' - DiceValues.csv')

ScoringLogDF = pd.DataFrame(Log_Scoring)
ScoringLogDF.columns=['Game', 'Turn', 'LastRoll', 'Dice', 'Slot', 'Points']
ScoringLogDF.to_csv('Log - ' + ApproachName + ' - Scoring.csv')

RollLogDF = pd.DataFrame(Log_Rolls)
RollLogDF.columns=['Game', 'Turn', 'Roll', 'Dice', 'HeldDice', 'HoldReason']
RollLogDF.to_csv('Log - ' + ApproachName + ' - Rolls.csv')

print('Average scores over', GamesToPlay, 'games of Yahtzee using the', ApproachName, 'strategy:')
print('Seed:', str(SeedtoUse))
print(MasterScoreSheet[MasterScoreSheet.columns[~MasterScoreSheet.columns.isin(['Approach', 'Game'])]].mean())
print(ApproachName)
MasterScoreSheet.to_csv('Log - ' + ApproachName + ' - MasterScoreSheet.csv')
```