SDSC4026 Sys Simulation & Modelling

Title: Simulation of CNY Blackjack Games for the Most Optimal Strategy

Group 20

Tan Hong Han	[redacted]	Player 1, Player 2, Player 3, Player4
Jacinta Quek	[redacted]	Motivation, Background, Player 5
Ng Wei Hao	[redacted]	Game Rules and Assumptions, Methodology, Player 6, Discussion

I Introduction/Motivation

Blackjack is a familiar fixture in many Chinese New Year gatherings—a game of chance, skill, and festive cheer. With limited decks at home and simplified rules for accessibility, the game evolves into a unique version shaped by house traditions. Given the game's cultural popularity, many house-grown strategies and "sure-win" methods have surfaced over time. In this project, we seek to investigate and compare these strategies to uncover the most optimal approach to win CNY Blackjack. This paper documents our efforts to identify the most optimal gameplay strategy within this uniquely festive and constrained setting.

II Background

With roots in the 18th-century French game "Vingt-et-Un" (Twenty-One), Blackjack has grown into one of the most popular casino games and a staple in household gatherings. The objective is to beat the dealer by achieving a hand value as close to 21 as possible without exceeding it. Players lose if their hand value surpasses 21 (bust). Face cards are worth 10, numbered cards retain their values, and Aces can count as 1 or 11. Each round begins with two cards dealt to both player and dealer, the latter's second card face-down. Players can Hit (take additional cards) or Stand (keep their hand), while the dealer must Hit until reaching a hand value of at least 16. Blackjack's enduring popularity has inspired the development of strategies to increase players' odds of winning. While early approaches were heuristic, Edward O. Thorp revolutionised the game with his card-counting techniques, published in Beat the Dealer. These methods leverage probability to give skilled players a statistical advantage, cementing Blackjack's appeal as a game of both chance and strategy.

III Game Rules and Assumptions

Game Rules	Assumptions
1. Deck Setup: game is played with **1 deck**. 2. Game Mechanics: **No splitting or doubles**. 3. Game Style: player plays only house, **1v1**. 4. Win Priority: 1. Pair Aces (AA) 2. Natural Blackjack (A + 10/J/Q/K) 3. 5 Card Charlie (5 cards ≤ 21) 4. Standard Win (under 21 and beats dealer) **Minimum valid hand value = 16**	1. For Decision Focused Strategies, Bet size is fixed at \$10, unless informed by strategy (card counting). 2. For Bet Sizing Strategies, Players are rational hence assumes the risk profile of the avg rational player, unless informed by strategy (card counting). 3. All players have unlimited money.

IV Methodology

We model five player profiles after common Chinese New Year (CNY) Blackjack archetypes, along with a sixth profile powered by Monte Carlo Reinforcement Learning (MCRL), serving as a theoretical benchmark for optimal decision-making. To evaluate the effectiveness of various Blackjack strategies under our custom Chinese New Year house rules, we simulate 300 rounds for each player profile in Python. Players are categorized into two groups: decision-focused (Profiles 1,2,3,5,6) and bet-sizing strategies (Profiles 4,5). All players operate under the same win resolution system whenever both player and house hits 21. Each profile is evaluated based on two key metrics: winning percentage against the house and average profit per round. We then synthesise our findings and identify the best strategy.

V. Player Strategy Dive (Simulation, Results Analysis)

5. 1 First 4 player profiles

Simulation

For the first 4 player profiles, start by importing the necessary libraries such as random and panda. Next, define the first three player profiles' decision making as data frames. For the average rational player, if the player's total points is less than 15, the player will hit. If the player's total points is 16 or 17 and the dealer's upcard is "7,8,9,10,A" the player will hit as well. Otherwise, the player will stand. For the overly conservative strategy, once the player hits the minimum requirement of 16 points, the player will stand. For the overly aggressive strategy, the player will hit as long as it does not get a blackjack or the player's total points is 20 and the dealer's upcard is an Ace. Lastly, based on assumptions made, the betting god strategy uses the average rational player's dataframe as well. However, the bet size will double in the next round everytime the player loses.

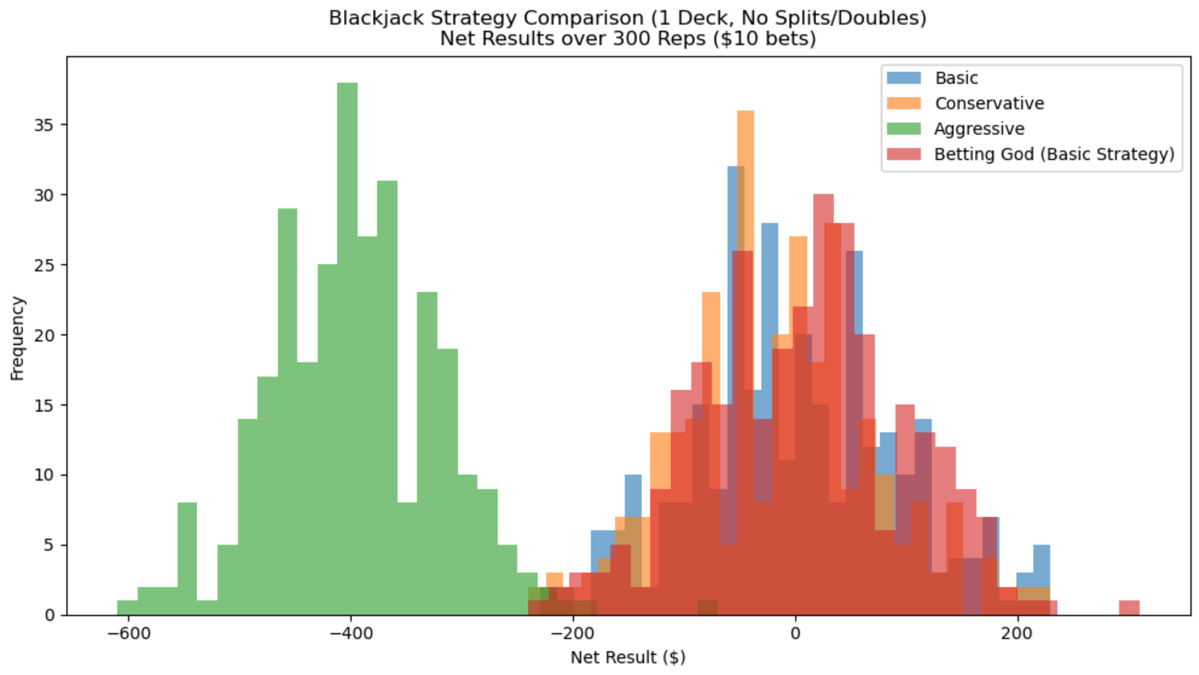
Next, introduce the Player and Deck class to set up the game. Next, introduce functions to shuffle, deal and simulate each strategy accordingly, as well as to determine the winner of each simulation. Lastly, run the simulations and display the results. For easier reference, the pseudocode and dataframes for each strategy is provided in the appendix.

Simulation Results

After 300 runs, the Average Win Rate of the average rational player is 42.1%, while the Expected Return per Game is $0.002. The Average Win Rate of the overly conservative player is 42.0%, while the Expected Return per Game is $-0.158. The Average Win Rate of the overly aggressive player is 18.6%, while the Expected Return per Game is $-3.967. The Average Win Rate of the betting god strategy is 42.4%, while the Expected Return per Game is $0.075. Sample simulations for each strategy can be found in the appendix.

Result Analysis

For the first 4 player profiles, the net result over 300 repetitions can be seen in the graph below.



From this graph, it is clear that the Overly Aggressive strategy leads to the worst net results, while the other three strategies are generally similar in terms of results. However, the Betting God strategy has the highest potential to lead to the greatest positive net results, due to the ability to win back the player's losses after implementing the martingale strategy.

5.2. Card Counting

Modelling Steps

The **card valuation function** consist of two main components: get_card_value(card), which assigns a Hi-Lo count value to a given card based on its impact on the game—cards **2 to 6** receive a **+1** as they are favourable to the player, **7 to 9 hold** a neutral value of **0**, and high-value cards (**10, J, Q, K, A**) are marked **-1**, as they favour the dealer. Additionally, convert_card_to_value(card) translates individual cards into their **blackjack numerical values**, where **face cards (J, Q, K) equate to 10**, and **Ace defaults to 11**, though its value may vary depending on the game context.

Card Value	Value	Reason
2 - 6	+1	Favourable to the player
7 - 9	0	Neutral impact
10, J, Q, K, A	-1	Favourable to dealer

Deck Simulation

The Deck class is responsible for managing multi-deck shuffling, which defaults to 1 deck, and incorporates a penetration threshold of 75%, reshuffling after 75% of the cards have been dealt. It tracks the cumulative Hi-Lo values using a running count and calculates the true count by dividing the running count by the number of remaining decks ($\frac{Running\ Count}{Remaining\ Decks}$) . This process is illustrated in Figure 1.

Player Strategy

The Player class is designed to strategically manage gameplay decisions. It tracks the player's hand value, ensuring adjustments are made for Aces, which can be valued as either 11 or 1 depending on the situation. To decide the player's actions, it relies on predefined strategy tables—Basic, Ace, and Pair—covering key moves like hitting (H), standing (S), doubling down (D), and splitting (P). Additionally, the class incorporates logic for automatic hitting when the player's hand value is less than 16, prioritising optimal decision-making based on the game's rules and probabilities. This process is illustrated in Figure 2.

Dynamic Betting

The betting system is designed to adapt dynamically based on the true count, optimising wagering strategies to align with favourable or unfavourable odds. The base bet starts at $10, providing a stable foundation. When the true count exceeds +2, indicating favourable odds, the bet is doubled to capitalise on the player's advantage. Conversely, when the true count drops below -2, signaling unfavorable conditions, the bet is halved to minimise potential losses. To ensure control and adaptability, the system includes adjustable boundaries, with a minimum bet of $0 and a maximum bet of $20. This structure balances risk and reward, enhancing the effectiveness of the betting strategy. This process is illustrated in Figure 3.

Simulation Workflow

The simulation follows these steps:

1. Initialisation: Set up the game environment by importing the necessary libraries and initialising key components, including the deck, player, and strategy tables. These tables are used to guide the player's decisions during gameplay. There are also two functions introduced the

2. Dealing: Cards are dealt to both the player and the dealer. A check is performed for blackjack (Ace + 10). If either party achieves blackjack, the round concludes immediately and the results are recorded.

3. Player Decision: The player evaluates their hand based on its value and refers to the predefined strategy tables to decide whether to hit, stand, double down, or split. Using the true count derived from card values, the player also adjusts their bets accordingly to maximise profitability. This decision-making process combines strategy tables with card counting to optimise outcomes.

4. Dealer Play: The dealer adheres to fixed rules, hitting until their hand value reaches or exceeds 16 ensuring consistency.

5. Outcome Determination: The results are calculated based on the final hand values:

- **Win**: The player's hand value is higher than the dealer's or the dealer busts.
- **Loss**: The player busts or the dealer's hand value is higher.
- **Push**: Both the player and dealer have equal hand values.

6. Result Tracking: Outcomes are recorded and analysed, including wins, losses, bets placed, and changes to the player's bankroll. These results are used to identify trends, evaluate the effectiveness of the strategy, and highlight opportunities for improvement.

Results and Analysis

Simulation Outcomes: Over 300 games, the player achieves a win rate of approximately 46% and incurs an average loss of $0.0667 per game. Sample simulations of the game are illustrated in Figure 5.

5.3 The Monte Carlo

Monte Carlo Reinforcement Learning (MCRL) could be applied to investigate either decision focused strategies and/ or betting sizing strategies. We have focused our use of Monte Carlo Reinforcement Learning methods to explore only the decision focused strategies. While other MCRL methods were simulated, MCFS remains the focus.

Defining the Blackjack Environment and Helper Functions

BlackJack Environment

The Blackjack environment models both the player and dealer behavior while enabling interaction via discrete state and action spaces. States are represented by a 3-dimensional tuple: the presence of a usable ace, the player’s hand sum (12–21), and the dealer’s visible card (Ace–10). Actions are binary: 0 for Stand, 1 for Hit.

The game logic ensures players keep drawing until a minimum hand sum of 16 (as any value below necessitates hitting) and incorporates realistic Ace valuation. To support prioritized reward structures, hands are labeled using a helper function into one of five categories: *Pair of Aces*, *Blackjack (Natural)*, *5-Card Charlie*, *Regular*, and *Bust*. These labels allow nuanced reward assignment based on hand strength and game outcome, which is essential for evaluating and training Monte Carlo agents. The full environment logic, including reward assignment and state transitions, is included in **Figure 6.1.a**.

Helper Functions

To support the Testing and Visualization of the trained Monte Carlo agents, two key helper functions were developed: play_test() and plotter().

Helper Function	Purpose	Usage
play_test()	1. Simulates a specified number of games using a trained agent and logs the resulting win, tie, and loss rates. 2. Encourages the agent to learn to **win > tie > lose** by assigning rewards accordingly. 3. Computes the expected return using a fixed $10 bet size assumption.	This function is utilised in **Section 6.4**, where we quantitatively assess agent performance through aggregated win rates and returns.
plotter()	1. Visualizes the Q-values learned by the agent over the state space. 2. For both usable and non-usable ace cases, it plots 3D bar charts showing the maximum expected reward across state-action pairs, and indicates whether Stand or Hit is the optimal action through color coding.	This is used in **Section 6.5.2**, which analyzes the structure and policy learned by the agent via Q-value surfaces.

It is important to distinguish that while **win priority rules** (e.g., *Pair of Aces > Blackjack > Regular win*) are used to settle ties between winning hands, the **reward incentive structure** (i.e., *win > tie > loss*) is what drives the agent's learning process and policy formation.

The full implementation of these helper functions is included in **Figure 6.1.b**.

Overview of the Monte Carlo Agents
Defining the Monte Carlo Methods
In Table below, we outlined 4 MCRL methods that were explored: MCFS, MCTS, MC-On Policy, MC- Off Policy.

#	MCRL Method	Description
1	Monte Carlo First Search (MCFS)	Updates Q(s,a) only on the first occurrence of a state-action pair per episode. Also known as *Monte Carlo First Visit.*
2	Monte Carlo Tree Search (MCTS)	Updates Q(s,a) every time a pair is seen. Also known as *Monte Carlo Every Visit.*
3	Monte Carlo On Policy (MC-On Policy)	Uses an ε-greedy policy for both training and evaluation.
4	Monte Carlo Off Policy (MC-Off Policy)	Samples using a behavior policy and corrects using importance sampling.

Implementation Overview of the Monte Carlo Methods

The table below maps the four MCRL types to their respective method implementations within the AgentMC class (**Figure 6.2.a**), clarifying how each strategy is instantiated and trained.

AgentMC method	Purpose	Usage
predict_q()	Implements the core update logic for MCFS and MCTS strategies. 1. Samples episodes from the environment using ε-greedy behavior. 2. Supports First Visit updates (MCFS) and Every Visit updates (MCTS) via a first_Search toggle. 3. Tracks convergence of Q-values over time (early stopping logic	Used to train **MCFS and MCTS agents** by setting first_Search = True or False respectively. Refer to **Section 6.4.1** and **Figure 6.2.d**.

	commented out for consistency).	
GPI()	Implements Generalized Policy Iteration for MC-On and MC-Off Policy strategies. 1. Allow agents to be trained using either its own policy (on-policy) or a behavioral policy (off-policy). 2. Uses **importance sampling** for off-policy updates.	Used to train **MC-On** and **MC-Off Policy** agents by setting on_polic = True or False respectively. Refer to **Section 6.4.1** and **Figure 6.2.e**.
sample_episode()	Generates a complete episode from environment interaction. Records the sequence of state-action-reward triplets from start to terminal state.	Used by both predict_q() and GPI() to sample agent-environment trajectories. Refer to **Figure 6.2.c**.
act()	Performs greedy action selection using the current Q-values.	Used during testing (e.g., in play_test() in **Section 6.4.1**).
target_policy() / behavioral_policy()	Define the agent's ε-greedy policies. target_policy is used for on-policy actions; behavioral_policy is used for off-policy exploration.	Supports exploration exploitation in both on- and off-policy training. Refer to **Figure 6.2.b**.

The full implementation of these agent methods is included in **Figures 6.2.a to 6.2.e**.

Model training

To ensure consistency in experimental results and enable reproducibility, a fixed random seed (**SEED = 42**) was set prior to training. All four MCRL agents were trained independently for 500,000 episodes each, using the AgentMC class defined earlier.

A **discount factor of γ = 1.0** was used throughout, reflecting the short-horizon nature of Blackjack where rewards are received almost immediately—making discounting unnecessary. (**Figure 6.2.a**)

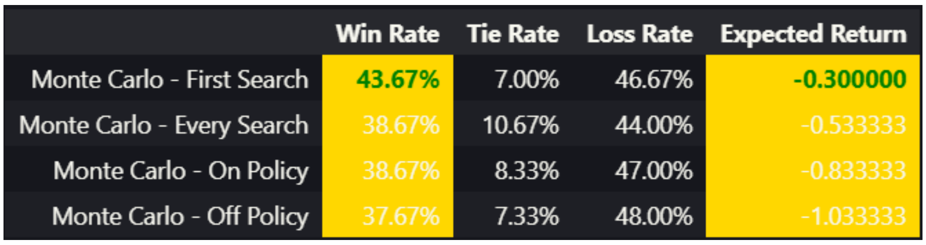
The training process was segmented according to each MCRL method. A summary of the agent methods and configurations is shown below.

MCRL Method	AgentMC Method	Key Configuration
MCFS	predict_q()	first_Search = True: updates Q-values **once per (s,a)** per episode
MCTS	predict_q()	first_Search = False: updates Q-values **every time (s,a)** appears in episode
MC- On Policy	GPI()	Uses target_policy() (ε = 0.1) for both training and evaluation. **No importance sampling**.
MC- Off Policy	GPI()	Uses behavioral_policy() (ε = 0.5) for sampling and target_policy() (ε = 0.1) for evaluation. **Importance sampling** adjusts Q-values.

All trained agent instances were stored in a centralized list: trained_Blackjack_agents for downstream evaluation, win-rate analysis, and policy visualization in **Section 6.4.1**. The full training code can be found in **Figure 6.3.a**

Model Testing

To evaluate agent performance after training, each Monte Carlo agent was tested over exactly 300 simulated Blackjack games using the play_test() function. During testing, each agent followed its learned policy, and outcomes were recorded in terms of Win Rate, Tie Rate, Loss Rate, and Expected Return.



	Win Rate	Tie Rate	Loss Rate	Expected Return
Monte Carlo - First Search	43.67%	7.00%	46.67%	-0.300000
Monte Carlo - Every Search	38.67%	10.67%	44.00%	-0.533333
Monte Carlo - On Policy	38.67%	8.33%	47.00%	-0.833333
Monte Carlo - Off Policy	37.67%	7.33%	48.00%	-1.033333

The MCFS agent achieved the **highest win rate at 43.67%** and the best expected return of -0.30 hence incurs an **average loss of $0.30 per game**. However, it is important to note that all Monte Carlo models exhibit non-deterministic behavior due to random sampling and policy exploration. As a result, the best-performing model in one training cycle may differ in future runs.

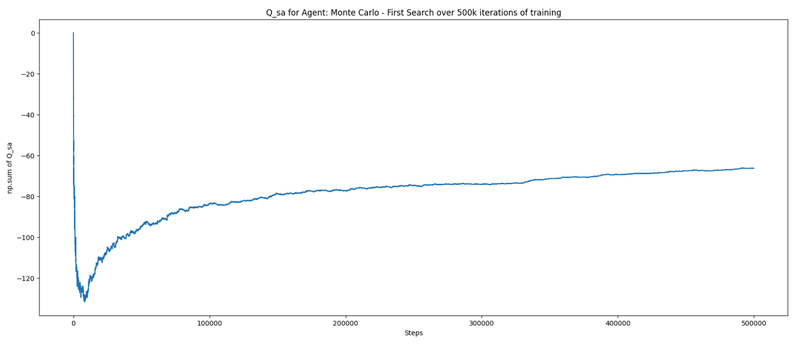
Analysis of Monte Carlo First Search

Given that the MCFS agent demonstrated the best performance in both Win rate and Expected return, we conduct a deeper inspection of its learning behavior and final policy to ascertain its reliability. We evaluate two aspects first the convergence of its Q-values across training and next the structure of its learned policy through 3D Q_sa surface plots, split by usable vs. non-usable Ace states.

These analyses allow us to assess both the agent's training stability and the rationality of its final strategy.

Visualise Q_sa convergence of MCFS

We visualized the convergence of the MCFS agent's Q-values by plotting the sum of Q(s,a) across training iterations. These Q-values, which represent the expected return of taking action a in state s, were stored internally by the agent's AgentMC class and logged via the Q_sa_history attribute after each episode.



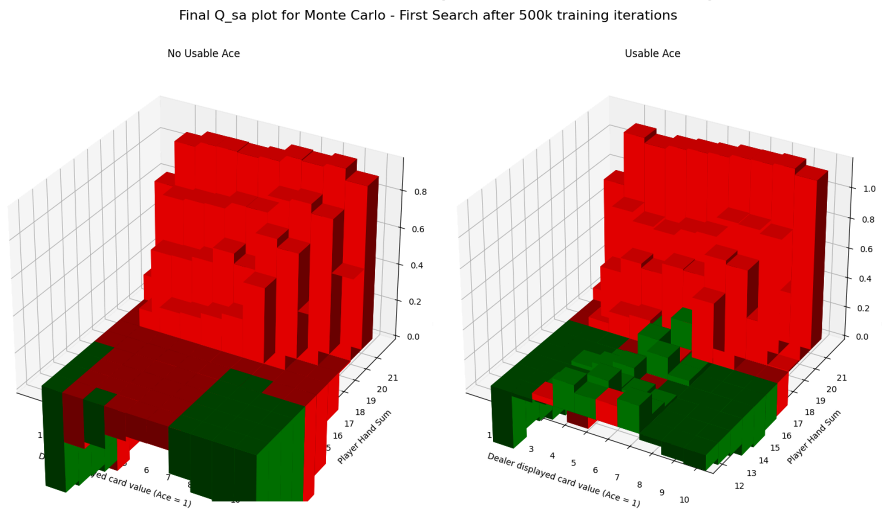
The plot shows that while the initial portion of training is highly volatile — due to the agent's exploration of suboptimal actions — the Q-values begin to stabilize by around 300,000 episodes, and plateau thereafter. **(Refer to code for higher resolution image)**

This indicates that the agent has effectively converged to a relatively stable policy by the 500,000th episode, despite no explicit convergence termination. This outcome supports our earlier decision to disable convergence stopping for fairness across agent types.

The steady increase in total Q-value reflects increasingly accurate action-value estimates as the agent collects more data. Indicative of a key property of Monte Carlo learning without bootstrapping or function approximation.

Visualise MCFS's learned state-action values (Usable vs non-Usable Ace)
We visualized the final Q-values learned by the MCFS agent using the plotter() function, as shown below.

Each 3D bar represents the maximum expected reward in a given state, color-coded by the optimal action: Green for *Hit* and Red for *Stand*. **(Refer to code for higher resolution image)**



To highlight key behavioral differences, we summarize the characteristics of both policies below:

Characteristic	Non- Usable Ace	Usable Ace
Policy Risk Profile	Conservative – stands more often	Aggressive – hits more often at moderate hand values
Decision Boundaries	Less distinct, flatter surface	Sharper, clearer transitions between Hit and Stand
Expected Reward Trend	Mostly negative across all states	Slightly positive in some mid-value hand ranges
Greedy Behaviour	Red-dominant (Stand) at high player sums	Green-dominant (Hit) at lower sums with high dealer cards
Notable Pattern	“Wall of red” when near 17–21	Similar wall, but delayed due to Ace flexibility
Potential Weaknesses	Lacks flexibility in low-mid hand regions	Some early Stand choices in low ranges may suggest undertraining or cautious policy artifacts

Despite the sharper policy in the Usable Ace case, the expected reward remains mostly negative or only marginally positive. This aligns with the nature of Blackjack, a game inherently biased toward the house. The MCFS agent’s average expected return of −0.30 (as shown in Section 6.4.1) reflects this — it confirms that the agent, while competent, cannot systematically outperform the house in the long run.

Interestingly, these plots closely resemble a 2D decision boundary, reminiscent of the simplified Blackjack strategy chart, where the optimal action toggles between *Hit* and *Stand* based on thresholds of player and dealer card values. The Usable Ace policy in particular exhibits a sharper frontier, reflecting the MCFS agent’s ability to confidently exploit favorable soft-hand states — even without implementing *Double* or *Split* actions. An extra analysis on win type distribution was performed in the code (**Part 5.3**).

Suggested Improvement

An improvement is to incorporate reinforcement learning methods beyond pure Monte Carlo, such as Temporal-Difference (TD) learning or hybrid methods like TD(λ). MC methods suffer from high variance and require full episodes before learning updates. TD methods learn more efficiently by updating Q-values at every time step, and hybrids can balance long-term planning with sample efficiency.

VI Discussions (Comparative Analysis, Conclusion)

Comparative Analysis

Summary Performance

Among the six simulated strategies, clear performance distinctions emerged. **The Card Counter** achieved the **highest win rate (46.00%)**. In contrast, **The Betting God** delivered the **highest expected returns (+0.075)**. At the other extreme, the **Overly Aggressive** player proved least effective, posting both the **lowest win rate (18.60%)** and the **worst expected return (-3.967)**.

Player Strategy Profile	Win Rate	Expected Returns
1. Average Rational Player	42.10%	+0.002
2. Overly Conservative Player	42.00%	-0.158
3. Overly Aggressive Player	18.60%	-3.967
4. The Betting God	42.40%	+0.075
5. The Card Counter	46.00%	-0.0667
6. The Monte Carlo	43.67%	-0.300

Performance Reflections Against the Control

Overly Conservative and Overly Aggressive vs. Average Rational Player

Both extreme strategies underperformed relative to the Average Rational Player. The Overly Conservative Player matched the control in win rate (42.00% vs. 42.10%) but posted a lower expected return 0f –0.158, reflecting missed opportunities from overly safe decisions. The Overly Aggressive Player fared worst overall, with a win rate of 18.60% and an expected return of –3.967, driven by frequent busts and reckless play. These results highlight that deviating too far in either direction — overly cautious or overly bold — leads to inefficiency, reinforcing the strength of a balanced, rule-based approach.

MCFS vs. Average Rational Player

The MCFS agent marginally outperformed the Average Rational Player in win rate (43.67% vs. 42.10%), showing that reinforcement learning can match and slightly improve upon rule-based strategies. This narrow gap reflects the strength of basic strategy in Blackjack and the limited advantage MCFS could exploit under fixed bet sizes and simplified rules.

The Betting God vs Average Rational Player

Despite being modelled after the exact playing strategy and risk profile, The Betting God achieved a higher expected return (+0.075 vs. +0.002) by adjusting bet sizes after losses. This return edge stems from betting behavior, not better decision-making. It highlights how profitability can diverge from strategy quality when capital is unconstrained.

Reconciling Deviations

While the Card Counter achieved the highest win rate (46.00%), it did not top the expected returns, raising questions about the effectiveness of its advanced strategy in our setup. A key factor lies in the single-deck configuration of our simulation. In real-world casinos, card counting strategies are optimized for multi-deck environments (typically 6 to 8 decks) where players can exploit card composition imbalances as the game progresses.

With just one deck, the effectiveness of card counting diminishes significantly. This reflects a limitation of our choice of abstraction: it mirrors at-home casual play, not casino conditions. At home, single-deck games are the norm, and the edge gained from tracking remaining cards is minimal. Thus, although the Card Counter's logic may be sound, its advantage was constrained by the simulation environment, suggesting that its high win rate may stem more from sound base strategy execution than from counting-based adaptations.

Overall Optimal Strategy

Across the six simulated profiles, no single strategy dominated both win rate and expected returns. This reflects the fundamental trade-off between decision quality and risk behavior: Decision-focused strategies prioritize accuracy and risk control, leading to stable performance and low variance outcomes. Bet-sizing strategies sacrifice decision consistency for higher variance in payouts, achieving better returns under idealised conditions like unlimited funds and no table limits.

While certain strategies performed better on specific metrics, our current data does not support a definitive conclusion on which approach is superior overall. Several key limitations include the lack of volatility data for understanding risk-adjusted performance and our 1v1 setup prevents head-to-head simulations where strategies compete with each other. While the isolated setting of our game simplifies comparison, it omits interaction effects present in real-world multiplayer blackjack. Additionally, the simulation's current abstraction of a single-deck setup has likely disadvantaged strategies like card counting, which rely on memory and deck composition.

<u>Conclusion</u>

This project set out to explore and simulate six Blackjack strategies to uncover what "optimal" looks like not just in theory, but in the festive, human, and occasionally chaotic setting of Chinese New Year blackjack.

Through our simulations, we observed distinct strengths and trade-offs across decision-making and bet-sizing strategies. Decision-focused models, like the Average Rational Player and MCFS, emphasized consistency and low variance. In contrast, The Betting God leveraged riskier bet sizing to amplify returns, a reminder that profitability can stem as much from money management as from sound card play.

Importantly, while Monte Carlo First Search did well in both win rate and strategic behavior, we recognize that real-life application diverges given that it is not intuitive to replicate this strategy in a quick round with family. Moreover, real players often don't stick rigidly to one policy as we are emotional decision-makers. Our strategy can shift depending on whether we're riding a hot streak or recovering from losses, introducing human factors like momentum bias and risk-seeking after setbacks. This makes static strategies useful as guides, but never fully representative of human play.

We also acknowledge that limitations are an inevitable part of any modeling abstraction. Our 1v1 setup simplified comparisons but omitted multiplayer dynamics, and our single-deck game constrained strategies like card counting. Still, these were deliberate trade-offs to keep the model interpretable and accessible. For the purpose of exploration and learning, this abstraction was perfectly adequate.

After all, creating memories and enjoying a fun, supportive environment with our red packet money matters more than winning every single round.

VII Appendix and references

Process Description

Player Profile	Strategy Type	Decision Making (Hit or Stand)	Bet Sizing
1.Average Rational Player	Basic Strategy	Yes	No
2.Overly Conservative	Heuristic	Yes	No
3.Overly Aggressive	Heuristic	Yes	No
4.Betting God	Basic + Martingale Strategy	No	Yes
5.Card Counter	Basic Strategy + Adjustments from Count	Yes	Yes
6.Monte Carlo	Monte Carlo (Decision-Making Focus)	Yes	No

Figure 1: Overview of Player Profiles, Strategies, and Decision Framework

First 4 Strategies

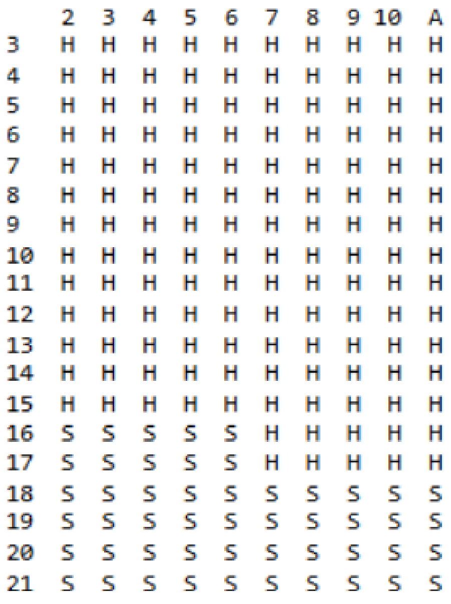
Pseudocode of the first 4 strategies:

```
BEGIN
    IMPORT necessary libraries
    DEFINE three blackjack strategies as DataFrames:
        - Basic (standard strategy)
        - Conservative (stands more)
        - Aggressive (hits more)
    CLASS Player:
        PROPERTIES: hand, points, money, bet, stats
        METHODS: update hand value
    CLASS Deck:
        PROPERTIES: cards
        METHODS: shuffle, deal card
    FUNCTION is_blackjack(hand)
        RETURN true if Ace + 10-value card
    FUNCTION evaluate_hand_quality(hand)
        RETURN quality score based on hand composition
    FUNCTION play_hand(player, dealer, strategy, deck):
        IF either has blackjack:
            Wins instantly
        WHILE player wants to hit:
            DRAW card based on strategy
        DEALER plays according to house rules
        COMPARE hands with tiebreakers
        UPDATE player stats and money
    FUNCTION run_simulation(strategy):
        INITIALIZE player and deck
        PLAY sample hands for display
        RUN multiple simulations
        CALCULATE statistics
        PLOT results
    MAIN:
        SET random seed
        RUN simulations for all strategies
        DISPLAY comparison results
END
```

For the **average rational player**, the **pseudocode** is as follows:

```
FUNCTION play_hand_basic_strategy(player_hand, dealer_upcard):
    WHILE player_hand.total < 21 AND player_hand.cards < 5:
        IF player_hand.total <= 15:
            HIT
        ELSE IF player_hand.total == 16 or 17:
            IF dealer_upcard in [2,3,4,5,6]:
                STAND
            ELSE:
                HIT
        ELSE IF player_hand.total >= 18:
            STAND
        // Dealer plays according to house rules
    dealer_hand = play_dealer_hand(dealer_hand)
    RETURN compare_hands(player_hand, dealer_hand)
```

As such, the following **data frame**, where the left column represents the total points of the player's hands and the top row represents the dealer's upcard, summarises the **average rational player's** decision making on whether to hit (H) or stand (S):

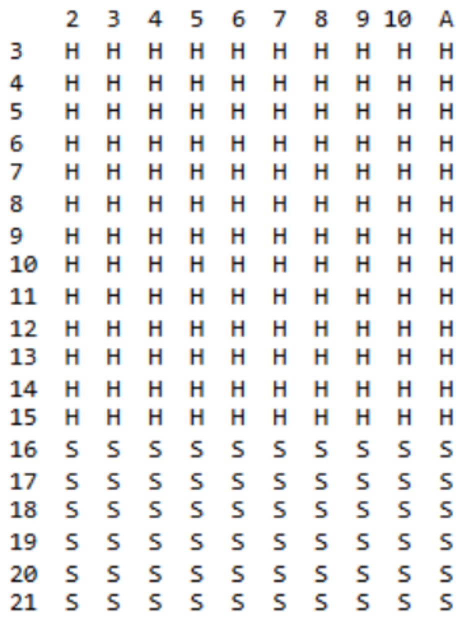


```
    2  3  4  5  6  7  8  9 10  A
3   H  H  H  H  H  H  H  H  H  H
4   H  H  H  H  H  H  H  H  H  H
5   H  H  H  H  H  H  H  H  H  H
6   H  H  H  H  H  H  H  H  H  H
7   H  H  H  H  H  H  H  H  H  H
8   H  H  H  H  H  H  H  H  H  H
9   H  H  H  H  H  H  H  H  H  H
10  H  H  H  H  H  H  H  H  H  H
11  H  H  H  H  H  H  H  H  H  H
12  H  H  H  H  H  H  H  H  H  H
13  H  H  H  H  H  H  H  H  H  H
14  H  H  H  H  H  H  H  H  H  H
15  H  H  H  H  H  H  H  H  H  H
16  S  S  S  S  S  H  H  H  H  H
17  S  S  S  S  S  H  H  H  H  H
18  S  S  S  S  S  S  S  S  S  S
19  S  S  S  S  S  S  S  S  S  S
20  S  S  S  S  S  S  S  S  S  S
21  S  S  S  S  S  S  S  S  S  S
```

For the **overly conservative player**, the **pseudocode** is as follows:

```
FUNCTION play_hand_conservative_strategy(player_hand, dealer_upcard):
    WHILE player_hand.total < 21 AND player_hand.cards < 5:
        IF player_hand.total <= 16:
            HIT
        ELSE:
            STAND
    // Dealer plays according to house rules
    dealer_hand = play_dealer_hand(dealer_hand)
    RETURN compare_hands(player_hand, dealer_hand)
```

As such, the following **data frame**, where the left column represents the total points of the player's hands and the top row represents the dealer's upcard, summarises the **overly conservative player's** decision making on whether to hit (H) or stand (S):

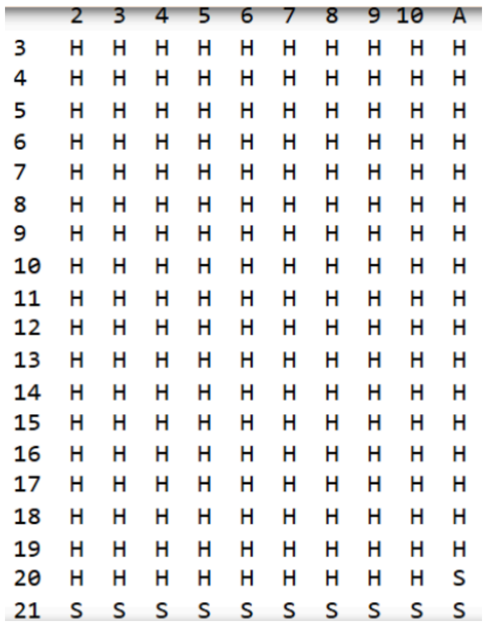


	2	3	4	5	6	7	8	9	10	A
3	H	H	H	H	H	H	H	H	H	H
4	H	H	H	H	H	H	H	H	H	H
5	H	H	H	H	H	H	H	H	H	H
6	H	H	H	H	H	H	H	H	H	H
7	H	H	H	H	H	H	H	H	H	H
8	H	H	H	H	H	H	H	H	H	H
9	H	H	H	H	H	H	H	H	H	H
10	H	H	H	H	H	H	H	H	H	H
11	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
16	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
20	S	S	S	S	S	S	S	S	S	S
21	S	S	S	S	S	S	S	S	S	S

For the **overly aggressive player,** the **pseudocode** is as follows:

```
FUNCTION play_hand_aggressive_strategy(player_hand, dealer_upcard):
    WHILE player_hand.total < 21 AND player_hand.cards < 5:
        IF player_hand.total <= 19:
            HIT
        ELSE IF player_hand.total == 20:
            IF dealer_upcard in [A]:
                STAND
            ELSE:
                HIT
        ELSE:
            STAND
    // Dealer plays according to house rules
    dealer_hand = play_dealer_hand(dealer_hand)
    RETURN compare_hands(player_hand, dealer_hand)
```

As such, the following **data frame**, where the left column represents the total points of the player's hands and the top row represents the dealer's upcard, summarises the **overly aggressive player's** decision making on whether to hit (H) or stand (S):

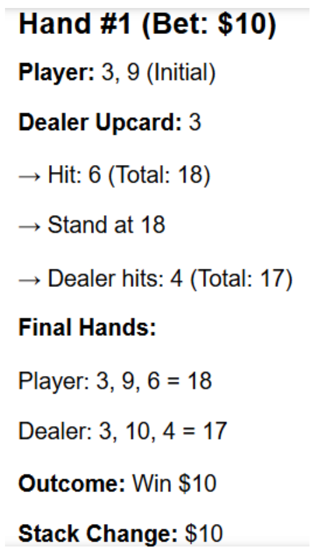


	2	3	4	5	6	7	8	9	10	A
3	H	H	H	H	H	H	H	H	H	H
4	H	H	H	H	H	H	H	H	H	H
5	H	H	H	H	H	H	H	H	H	H
6	H	H	H	H	H	H	H	H	H	H
7	H	H	H	H	H	H	H	H	H	H
8	H	H	H	H	H	H	H	H	H	H
9	H	H	H	H	H	H	H	H	H	H
10	H	H	H	H	H	H	H	H	H	H
11	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
17	H	H	H	H	H	H	H	H	H	H
18	H	H	H	H	H	H	H	H	H	H
19	H	H	H	H	H	H	H	H	H	H
20	H	H	H	H	H	H	H	H	H	S
21	S	S	S	S	S	S	S	S	S	S

For the **betting god**, the **pseudocode** is as follows:

```
FUNCTION play_hand_betting_god(player_hand, dealer_upcard):
    // Uses Basic Strategy for gameplay
    outcome = play_hand_basic_strategy(player_hand, dealer_upcard)
     // Special betting rules
    IF outcome == WIN:
        current_bet = initial_bet
        consecutive_losses = 0
    ELSE IF outcome == LOSE:
        consecutive_losses += 1
        current_bet = 2 * initial_bet
    ELSE: // Push
        consecutive_losses = 0
    RETURN outcome
```

Below are some sample **simulations for Average Rational Player**:



Hand #1 (Bet: $10)

Player: 3, 9 (Initial)

Dealer Upcard: 3

→ Hit: 6 (Total: 18)

→ Stand at 18

→ Dealer hits: 4 (Total: 17)

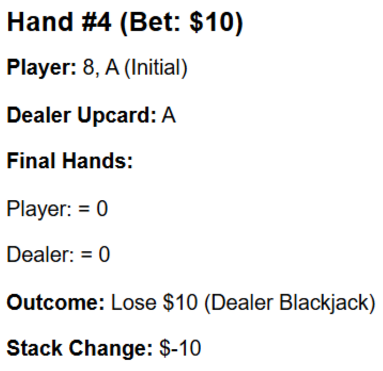
Final Hands:

Player: 3, 9, 6 = 18

Dealer: 3, 10, 4 = 17

Outcome: Win $10

Stack Change: $10



Hand #4 (Bet: $10)

Player: 8, A (Initial)

Dealer Upcard: A

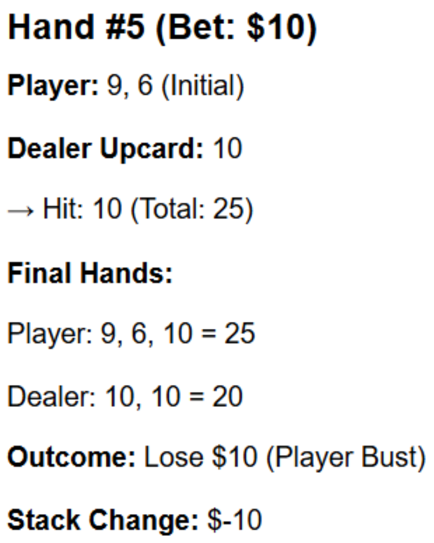
Final Hands:

Player: = 0

Dealer: = 0

Outcome: Lose $10 (Dealer Blackjack)

Stack Change: $-10



Hand #5 (Bet: $10)

Player: 9, 6 (Initial)

Dealer Upcard: 10

→ Hit: 10 (Total: 25)

Final Hands:

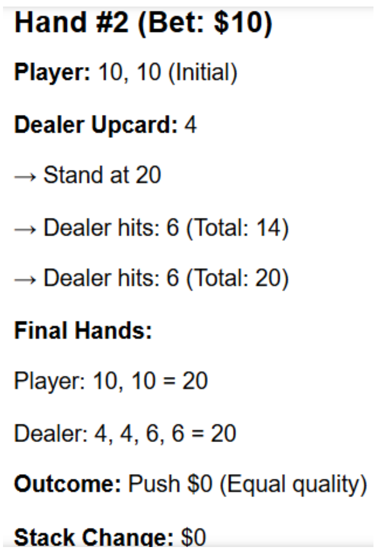
Player: 9, 6, 10 = 25

Dealer: 10, 10 = 20

Outcome: Lose $10 (Player Bust)

Stack Change: $-10

Below are some sample **simulations for the Overly Conservative player**:


Hand #2 (Bet: $10)

Player: 10, 10 (Initial)

Dealer Upcard: 4

→ Stand at 20

→ Dealer hits: 6 (Total: 14)

→ Dealer hits: 6 (Total: 20)

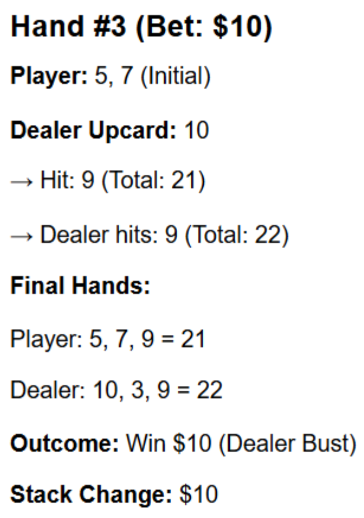
Final Hands:

Player: 10, 10 = 20

Dealer: 4, 4, 6, 6 = 20

Outcome: Push $0 (Equal quality)

Stack Change: $0


Hand #3 (Bet: $10)

Player: 5, 7 (Initial)

Dealer Upcard: 10

→ Hit: 9 (Total: 21)

→ Dealer hits: 9 (Total: 22)

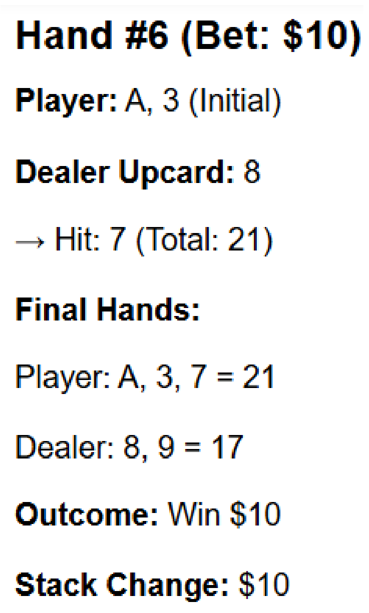
Final Hands:

Player: 5, 7, 9 = 21

Dealer: 10, 3, 9 = 22

Outcome: Win $10 (Dealer Bust)

Stack Change: $10


Hand #6 (Bet: $10)

Player: A, 3 (Initial)

Dealer Upcard: 8

→ Hit: 7 (Total: 21)

Final Hands:

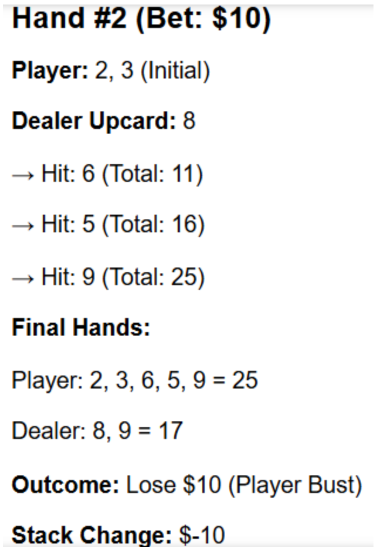
Player: A, 3, 7 = 21

Dealer: 8, 9 = 17

Outcome: Win $10

Stack Change: $10

Below are some sample **simulations for the Overly Aggressive player**:


Hand #2 (Bet: $10)

Player: 2, 3 (Initial)

Dealer Upcard: 8

→ Hit: 6 (Total: 11)

→ Hit: 5 (Total: 16)

→ Hit: 9 (Total: 25)

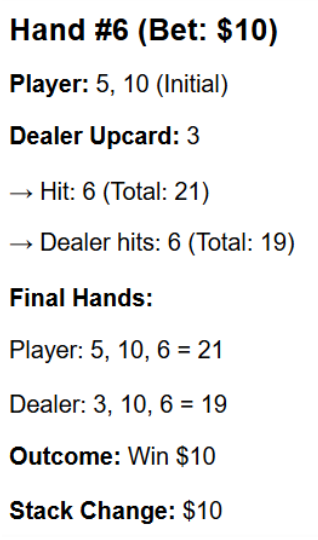
Final Hands:

Player: 2, 3, 6, 5, 9 = 25

Dealer: 8, 9 = 17

Outcome: Lose $10 (Player Bust)

Stack Change: $-10


Hand #6 (Bet: $10)

Player: 5, 10 (Initial)

Dealer Upcard: 3

→ Hit: 6 (Total: 21)

→ Dealer hits: 6 (Total: 19)

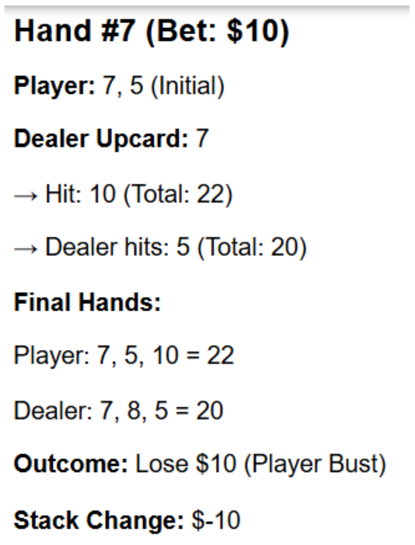
Final Hands:

Player: 5, 10, 6 = 21

Dealer: 3, 10, 6 = 19

Outcome: Win $10

Stack Change: $10


Hand #7 (Bet: $10)

Player: 7, 5 (Initial)

Dealer Upcard: 7

→ Hit: 10 (Total: 22)

→ Dealer hits: 5 (Total: 20)

Final Hands:

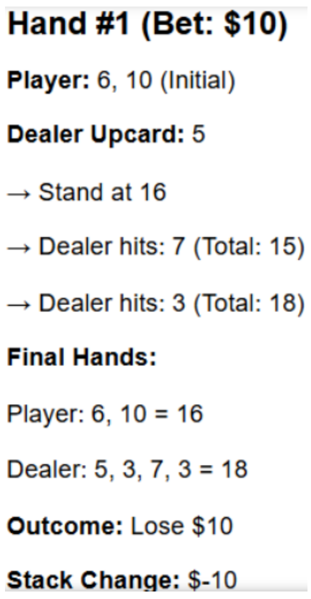
Player: 7, 5, 10 = 22

Dealer: 7, 8, 5 = 20

Outcome: Lose $10 (Player Bust)

Stack Change: $-10

Below are some sample **simulations for the Betting God Strategy**:


Hand #1 (Bet: $10)

Player: 6, 10 (Initial)

Dealer Upcard: 5

→ Stand at 16

→ Dealer hits: 7 (Total: 15)

→ Dealer hits: 3 (Total: 18)

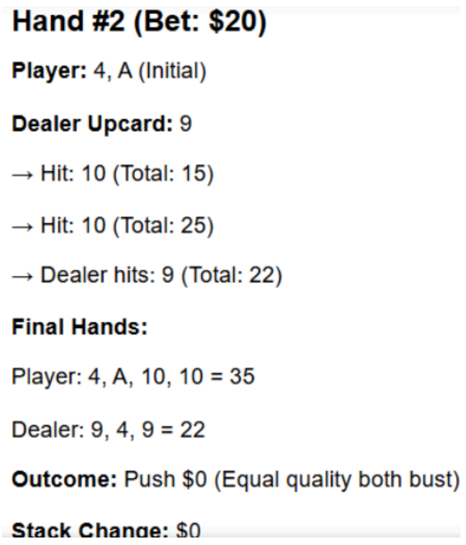
Final Hands:

Player: 6, 10 = 16

Dealer: 5, 3, 7, 3 = 18

Outcome: Lose $10

Stack Change: $-10


Hand #2 (Bet: $20)

Player: 4, A (Initial)

Dealer Upcard: 9

→ Hit: 10 (Total: 15)

→ Hit: 10 (Total: 25)

→ Dealer hits: 9 (Total: 22)

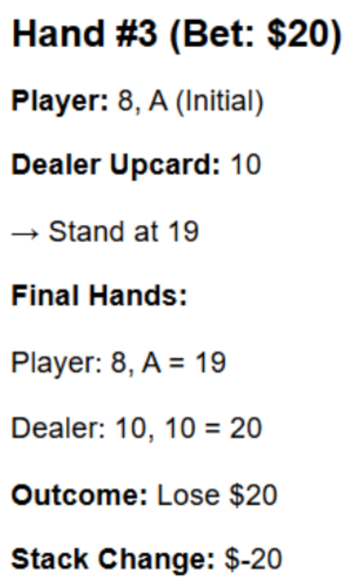
Final Hands:

Player: 4, A, 10, 10 = 35

Dealer: 9, 4, 9 = 22

Outcome: Push $0 (Equal quality both bust)

Stack Change: $0


Hand #3 (Bet: $20)

Player: 8, A (Initial)

Dealer Upcard: 10

→ Stand at 19

Final Hands:

Player: 8, A = 19

Dealer: 10, 10 = 20

Outcome: Lose $20

Stack Change: $-20

Card Counting

```python
# Deck Class with Running Count and True Count Integration
'''
Represents the shuffled deck(s) used in blackjack. It supports:
1. Maintaining and updating the Hi-Lo running count as cards are dealt.
2. Calculating the true count by adjusting the running count based on the number of remaining decks.
3. Initialises a shuffled deck and converts face cards to a value of 10 for consistency in blackjack.
4. The true count adjusts the running count based on the number of remaining decks,
providing a more accurate measure of the odds.
'''
class Deck:
    def __init__(self, num_decks=1,penetration = 0.75):
        """Creates a shuffled deck and initializes Hi-Lo count."""
        self.num_decks = num_decks
        self.num_decks = num_decks
        self.penetration = penetration
        self.reshuffle_point = int(num_decks * 52 * penetration)
        # represents the number of cards in the Deck
        self.cards = ([i for i in range(2, 11)] + ["J", "Q", "K", "A"]) * 4 * num_decks
        self.cards = [10 if i in ('J', 'Q', 'K') else i for i in self.cards]
        random.shuffle(self.cards)
        self.running_count = 0  # initialises the Hi-Lo Running Count

    def update_count(self, card):
        """Updates the running count based on dealt cards."""
        # calls the external function get_card_value() to determine the Hi-Lo value of the card
        self.running_count += get_card_value(card)

    def deal_card(self):
        if len(self.cards) <= (self.num_decks * 52) - self.reshuffle_point:
            self.__init__(self.num_decks)  # Reshuffle
        """Deals a card and updates the running count."""
        # checks if the deck is empty and reshuffles the deck by reinitialising it with the __init__
method
        card = self.cards.pop()
        self.update_count(card)
        return card

    def true_count(self):
        remaining_decks = max(len(self.cards) / 52, 0.5)  # Never less than 0.5 decks
        return self.running_count / remaining_decks
```

Figure 1: Deck class with running count and true count integration

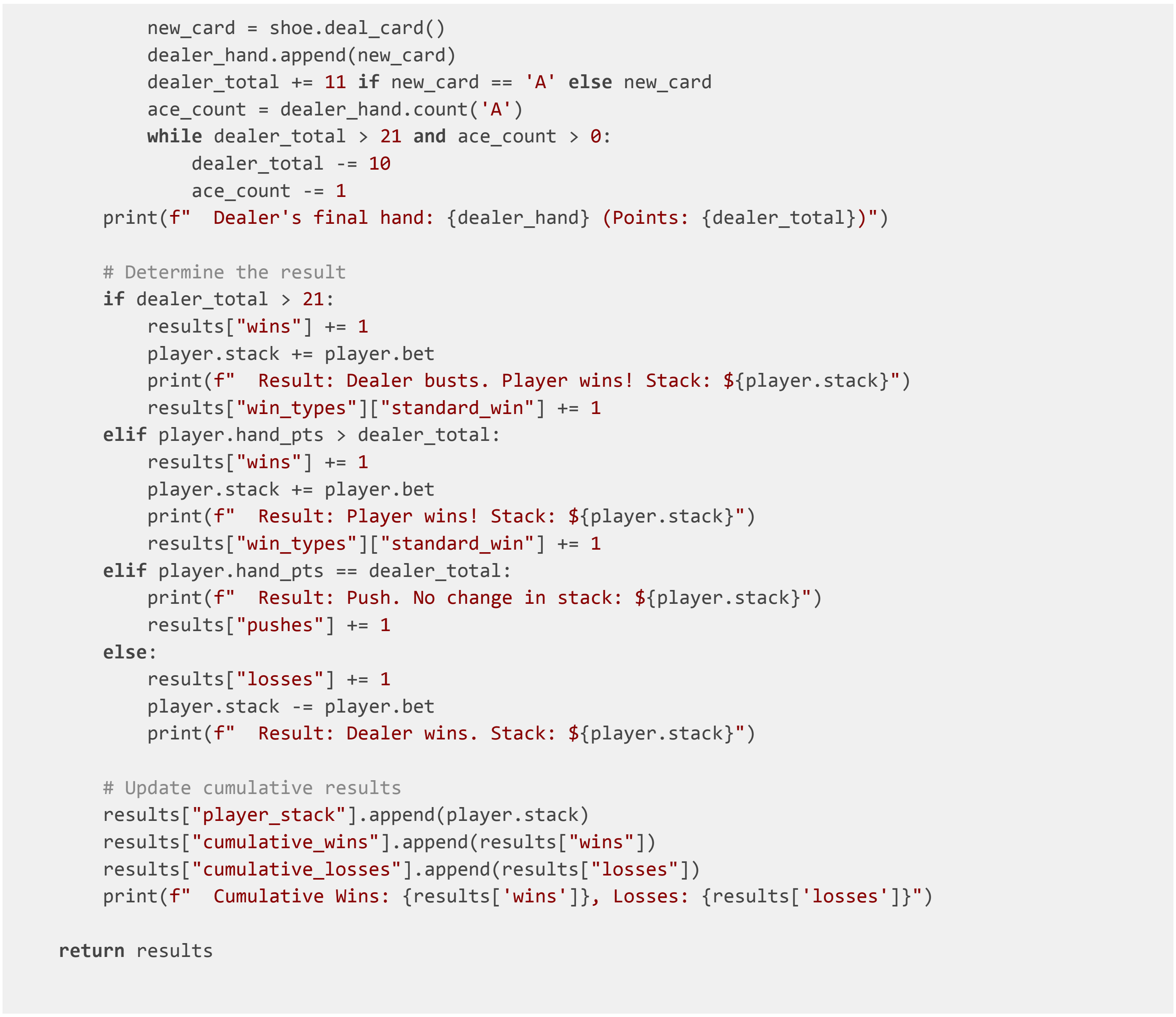
```python
class Player:
    def __init__(self, stack):
        """Initializes player attributes."""
        self.hand = []
        self.hand_pts = 0
        self.action = '' # stores the decision of the player to "hit","stand", "double down"
        self.stack = stack # players total chips, representing their current balance
        self.bet = 0  # amount the player bets in a round calculated dynamically based on strategy

    def update_hand_points(self):
        """Updates player's hand point total."""
        # calculates the initial points, treating Aces as 11 by default
        self.hand_pts = sum([11 if card == 'A' else card for card in self.hand])
        # counts the number of Aces in the hand
        ace_count = self.hand.count('A')
        # Special case: If there are exactly two aces, set hand points to 21
        if self.hand.count('A') == 2 and len(self.hand) == 2:
            self.hand_pts = 21
        else:
            #Adjust for aces if the hand has more than 2 cards and points exceed 21
            while self.hand_pts > 21 and ace_count > 0:
                self.hand_pts -= 10 #Reduce the value of each Ace from 11 to 1
                ace_count -= 1

    def decide_action(self, dealer_upcard, basic_df, ace_df, pair_df):
        """
        Decides the player's action based on the dealer's upcard and the current hand.
        """
        # Automatically force a hit if the player's hand points are less than 16
        while self.hand_pts < 16:
            new_card = shoe.deal_card()  # Deal a new card from the shoe
            self.hand.append(new_card)  # Add the new card to the player's hand
            self.update_hand_points()  # Update the player's hand points
            print(f"Player draws a card: {new_card}. New hand: {self.hand} (Points: {self.hand_pts})")

        # If player busts, no action needed
        if self.hand_pts > 21:
            self.action = 'B'  # Bust
            print(f"Player busts with {self.hand_pts} points")
            return

        # Convert hand cards to numerical values
        hand_values = [convert_card_to_value(card) for card in self.hand]

        if len(self.hand) == 2 and hand_values[0] == hand_values[1]:  # Pair strategy
            strategy_df = pair_df
            pair_value = hand_values[0] * 2
            if pair_value in strategy_df.index:
                self.action = strategy_df.loc[pair_value, dealer_upcard]
            else:
                self.action = 'H'  # Default to hit if pair value not in strategy
        elif 'A' in self.hand and self.hand_pts <= 21:  # Ace strategy (soft hand)
            strategy_df = ace_df
            if self.hand_pts in strategy_df.index:
                self.action = strategy_df.loc[self.hand_pts, dealer_upcard]
            else:
                self.action = 'H'  # Default to hit if soft hand value not in strategy
        else:  # Basic strategy (hard hand)
            strategy_df = basic_df
            if self.hand_pts in strategy_df.index:
                self.action = strategy_df.loc[self.hand_pts, dealer_upcard]
            else:
                self.action = 'H'  # Default to hit if hard hand value not in strategy

        print(f"Player's final hand: {self.hand} (Points: {self.hand_pts}). Decision: {self.action}")
```

Figure 2: Player Class

```
# Variable Betting Based on True Count
'''
Implementation of dynamic betting based on the true count and player stack
1. Higher bets and bets adjustments are placed when the true count indicates favourable odds
'''
def decide_variable_betting(true_count, player_stack, base_bet, min_bet = 0,max_bet = 20):
    max_bet = 1000
    # Adjust bet based on the true count
    if true_count > 2:
        bet = base_bet * 2  # Higher betting for favourable conditions
    elif true_count < -2:
        bet = base_bet / 2  # Minimise loss in unfavourable conditions
    else:
        bet = base_bet  # Default betting for neutral conditions

    # Ensure the bet is within valid boundaries and does not exceed player stack
    bet = round(bet)
    bet = max(min_bet, min(bet, max_bet, player_stack))
    return bet
```

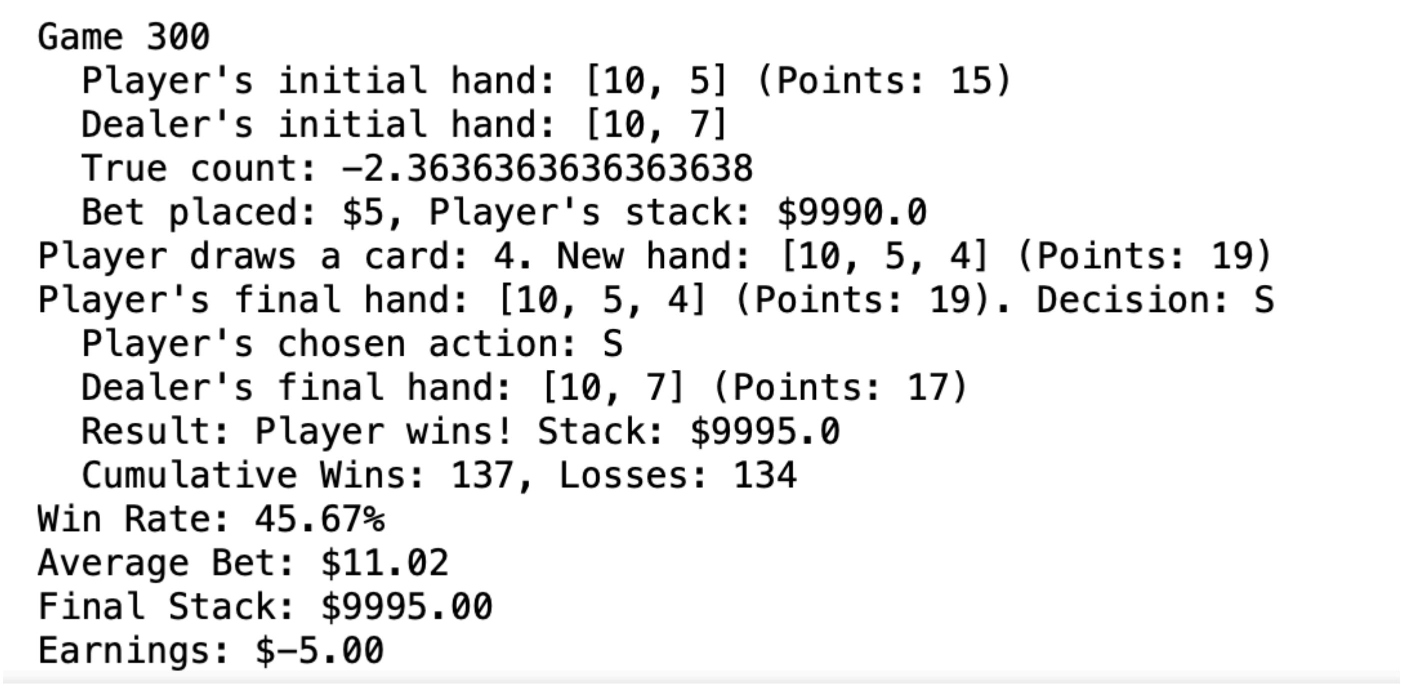
Figure 3: Dynamic Betting

```
# Simulation Logic
'''
Simulates multiple blackjack games, tracking:
Player wins/losses, betting patterns, Hi-Lo counts, and stack changes.
Incorporates logic for dealing cards, betting adjustments, and determining game outcomes.
'''
def is_blackjack(hand):
    """Check if hand is a natural blackjack (Ace + 10-value card)"""
    return len(hand) == 2 and (
        (hand[0] == 'A' and hand[1] in [10, 'J', 'Q', 'K']) or
        (hand[1] == 'A' and hand[0] in [10, 'J', 'Q', 'K'])
    )

def simulate_strategy(shoe, player, dealer_hand, strategy_basic, strategy_ace, strategy_pair, base_bet,
num_games=300):
    """Simulates 300 games and tracks performance."""
    results = {
        'wins': 0,
        'losses': 0,
        'pushes': 0,
        'player_stack': [player.stack],
        'bets': [],
        'cumulative_wins': [],
        'cumulative_losses': [],
        'win_types': {
            'pair_aces': 0,
            'natural_blackjack': 0,
            'five_card_charlie': 0,
            'standard_win': 0
        },
        'running_counts':[],
        'true_counts':[]
    }

    for game in range(num_games):
        # Reset hands and deal cards
        player.hand = [shoe.deal_card(), shoe.deal_card()]
        dealer_hand = [shoe.deal_card(), shoe.deal_card()]
        player.update_hand_points()
        true_count_value = shoe.true_count()

        print(f"\nGame {game + 1}")
        print(f"  Player's initial hand: {player.hand} (Points: {player.hand_pts})")
        print(f"  Dealer's initial hand: {dealer_hand}")
        print(f"  True count: {true_count_value}")

        # Dynamic bet calculation
        player.bet = decide_variable_betting(true_count_value, player.stack, base_bet)
        results["bets"].append(player.bet)
        print(f"  Bet placed: ${player.bet}, Player's stack: ${player.stack}")

        # Check for natural blackjacks
        player_blackjack = is_blackjack(player.hand)
        dealer_blackjack = is_blackjack(dealer_hand)

        if player_blackjack or dealer_blackjack:
            if player_blackjack and dealer_blackjack:
                # Both have blackjack - push
                print("  Both player and dealer have blackjack - Push")
                results["pushes"] += 1
            elif player_blackjack:
                # Player wins with blackjack (3:2 payout)
                win_amount = player.bet * 1.5
                player.stack += win_amount
                results["wins"] += 1
                results["win_types"]["natural_blackjack"] += 1
                print(f"  Player has blackjack! Wins ${win_amount:.2f}")
            else:
                # Dealer has blackjack - player loses
                player.stack -= player.bet
                results["losses"] += 1
                print("  Dealer has blackjack - Player loses")

            # Update results and continue to next game
            results["player_stack"].append(player.stack)
            results["cumulative_wins"].append(results["wins"])
            results["cumulative_losses"].append(results["losses"])
            continue

        # No blackjacks - proceed with normal play
        player.decide_action(dealer_hand[1], strategy_basic, strategy_ace, strategy_pair)
        print(f"  Player's chosen action: {player.action}")

        # Dealer auto-win logic
        if player.hand_pts > 21: #Player busts
            results["losses"] += 1
            player.stack -= player.bet
            print(f"  Result: Player busts. Stack: ${player.stack}")
            results["player_stack"].append(player.stack)
            results["cumulative_wins"].append(results["wins"])
            results["cumulative_losses"].append(results["losses"])
            continue

        # Dealer logic
        dealer_total = sum([11 if card == 'A' else card for card in dealer_hand])
        ace_count = dealer_hand.count('A')
        while dealer_total > 21 and ace_count > 0:
            dealer_total -= 10
            ace_count -= 1
        while dealer_total < 16:
```

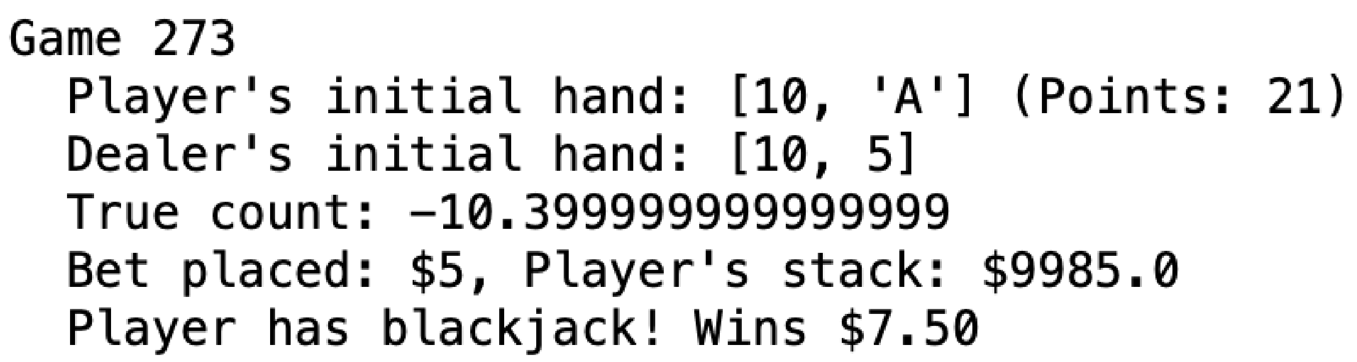


```
            new_card = shoe.deal_card()
            dealer_hand.append(new_card)
            dealer_total += 11 if new_card == 'A' else new_card
            ace_count = dealer_hand.count('A')
            while dealer_total > 21 and ace_count > 0:
                dealer_total -= 10
                ace_count -= 1
        print(f"  Dealer's final hand: {dealer_hand} (Points: {dealer_total})")

        # Determine the result
        if dealer_total > 21:
            results["wins"] += 1
            player.stack += player.bet
            print(f"  Result: Dealer busts. Player wins! Stack: ${player.stack}")
            results["win_types"]["standard_win"] += 1
        elif player.hand_pts > dealer_total:
            results["wins"] += 1
            player.stack += player.bet
            print(f"  Result: Player wins! Stack: ${player.stack}")
            results["win_types"]["standard_win"] += 1
        elif player.hand_pts == dealer_total:
            print(f"  Result: Push. No change in stack: ${player.stack}")
            results["pushes"] += 1
        else:
            results["losses"] += 1
            player.stack -= player.bet
            print(f"  Result: Dealer wins. Stack: ${player.stack}")

        # Update cumulative results
        results["player_stack"].append(player.stack)
        results["cumulative_wins"].append(results["wins"])
        results["cumulative_losses"].append(results["losses"])
        print(f"  Cumulative Wins: {results['wins']}, Losses: {results['losses']}")

    return results
```

Figure 4: Simulation logic and wining priority



```
Game 300
  Player's initial hand: [10, 5] (Points: 15)
  Dealer's initial hand: [10, 7]
  True count: -2.3636363636363638
  Bet placed: $5, Player's stack: $9990.0
Player draws a card: 4. New hand: [10, 5, 4] (Points: 19)
Player's final hand: [10, 5, 4] (Points: 19). Decision: S
  Player's chosen action: S
  Dealer's final hand: [10, 7] (Points: 17)
  Result: Player wins! Stack: $9995.0
  Cumulative Wins: 137, Losses: 134
Win Rate: 45.67%
Average Bet: $11.02
Final Stack: $9995.00
Earnings: $-5.00
```



```
Game 273
   Player's initial hand: [10, 'A'] (Points: 21)
   Dealer's initial hand: [10, 5]
   True count: -10.399999999999999
   Bet placed: $5, Player's stack: $9985.0
   Player has blackjack! Wins $7.50
```

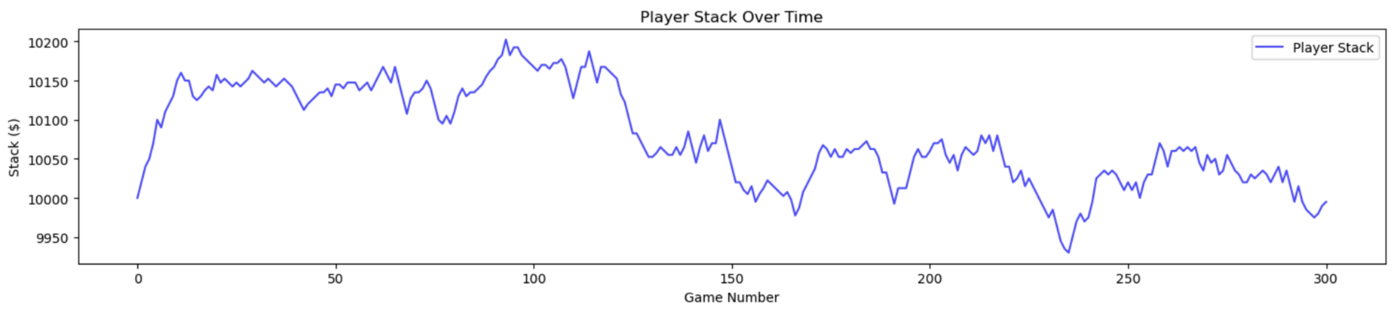
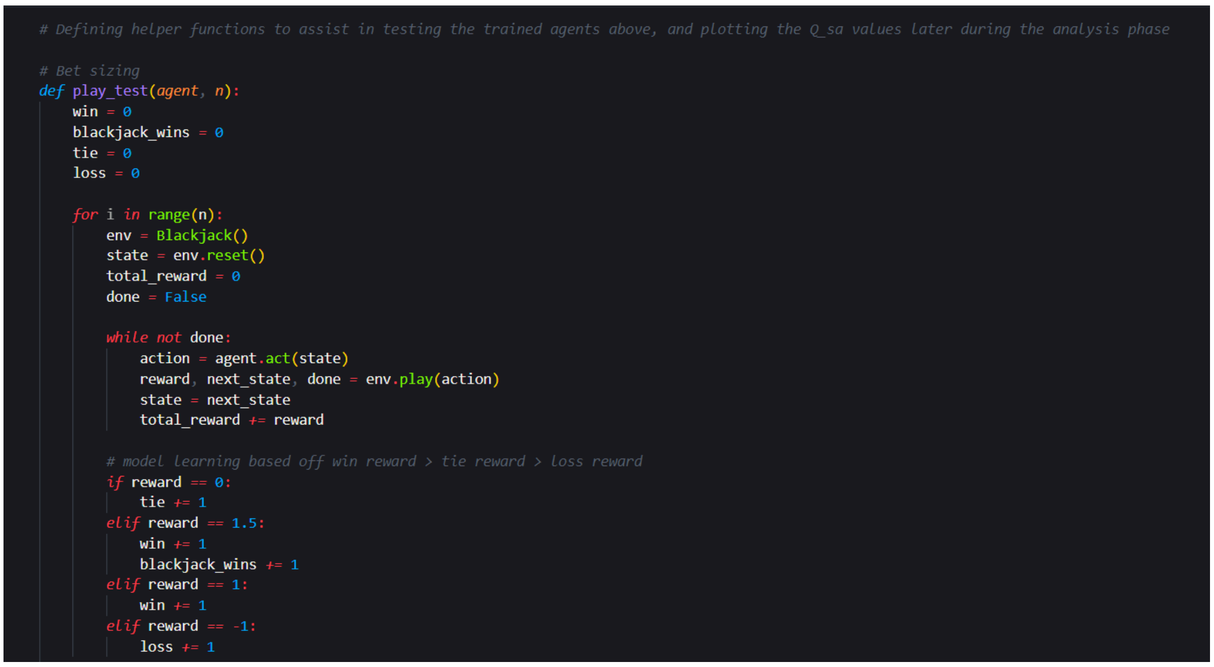


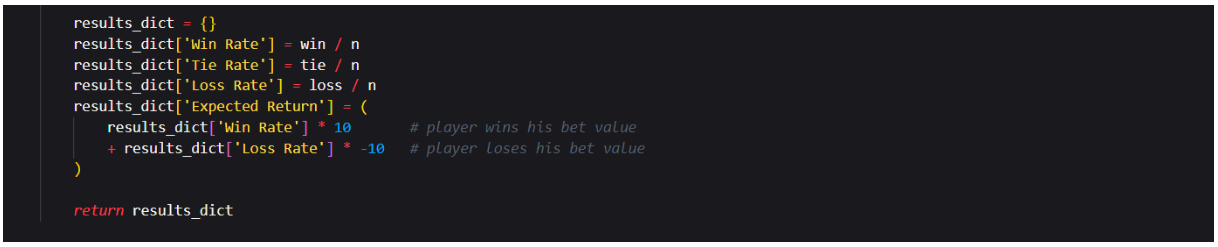
Figure 5: Sample Simulations for card counting

Monte Carlo

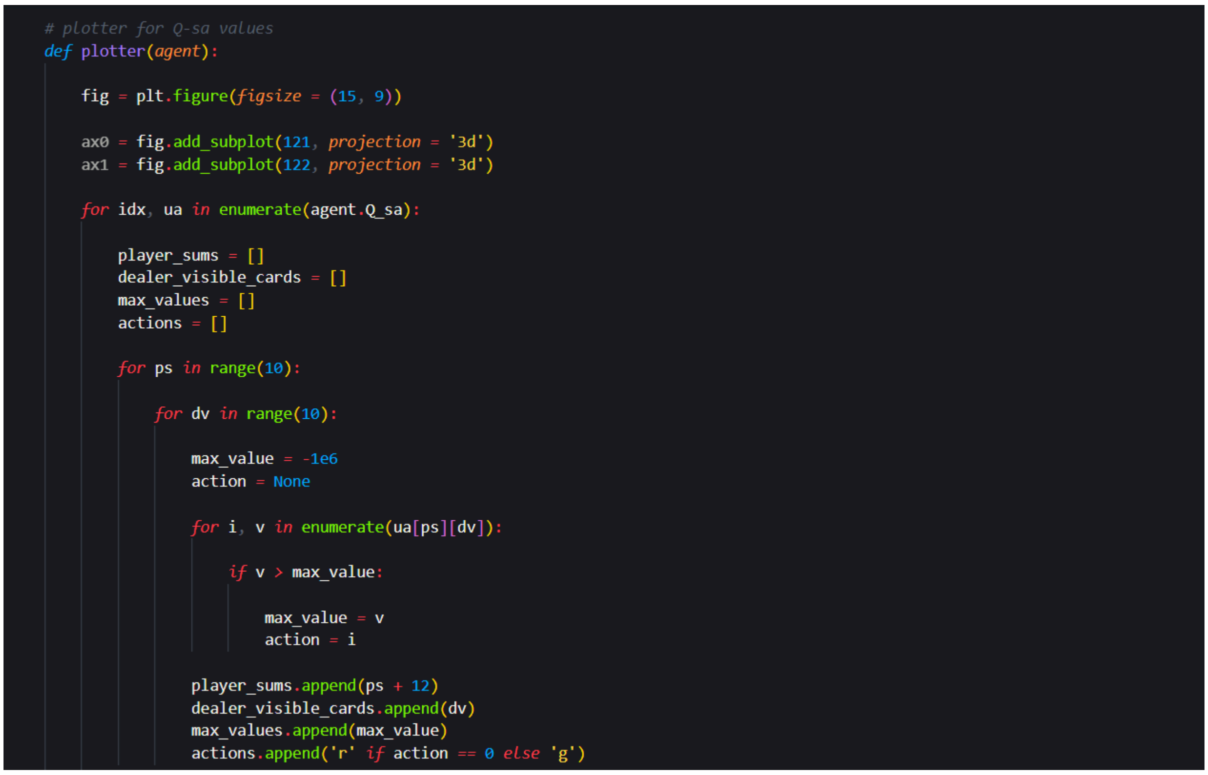
Figure 6.1.a



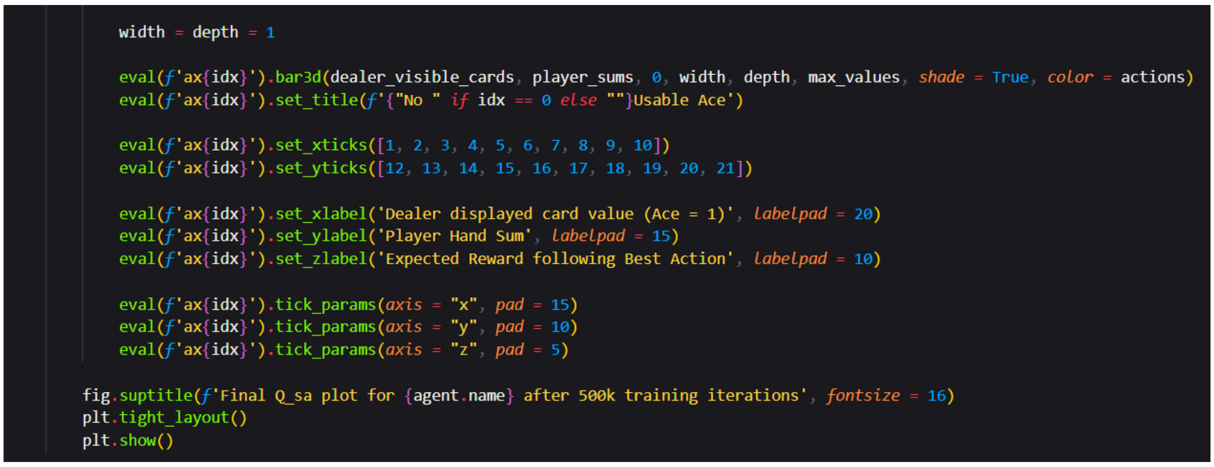
```python
# Defining helper functions to assist in testing the trained agents above, and plotting the Q_sa values later during the analysis phase

# Bet sizing
def play_test(agent, n):
    win = 0
    blackjack_wins = 0
    tie = 0
    loss = 0

    for i in range(n):
        env = Blackjack()
        state = env.reset()
        total_reward = 0
        done = False

        while not done:
            action = agent.act(state)
            reward, next_state, done = env.play(action)
            state = next_state
            total_reward += reward

        # model learning based off win reward > tie reward > loss reward
        if reward == 0:
            tie += 1
        elif reward == 1.5:
            win += 1
            blackjack_wins += 1
        elif reward == 1:
            win += 1
        elif reward == -1:
            loss += 1
```



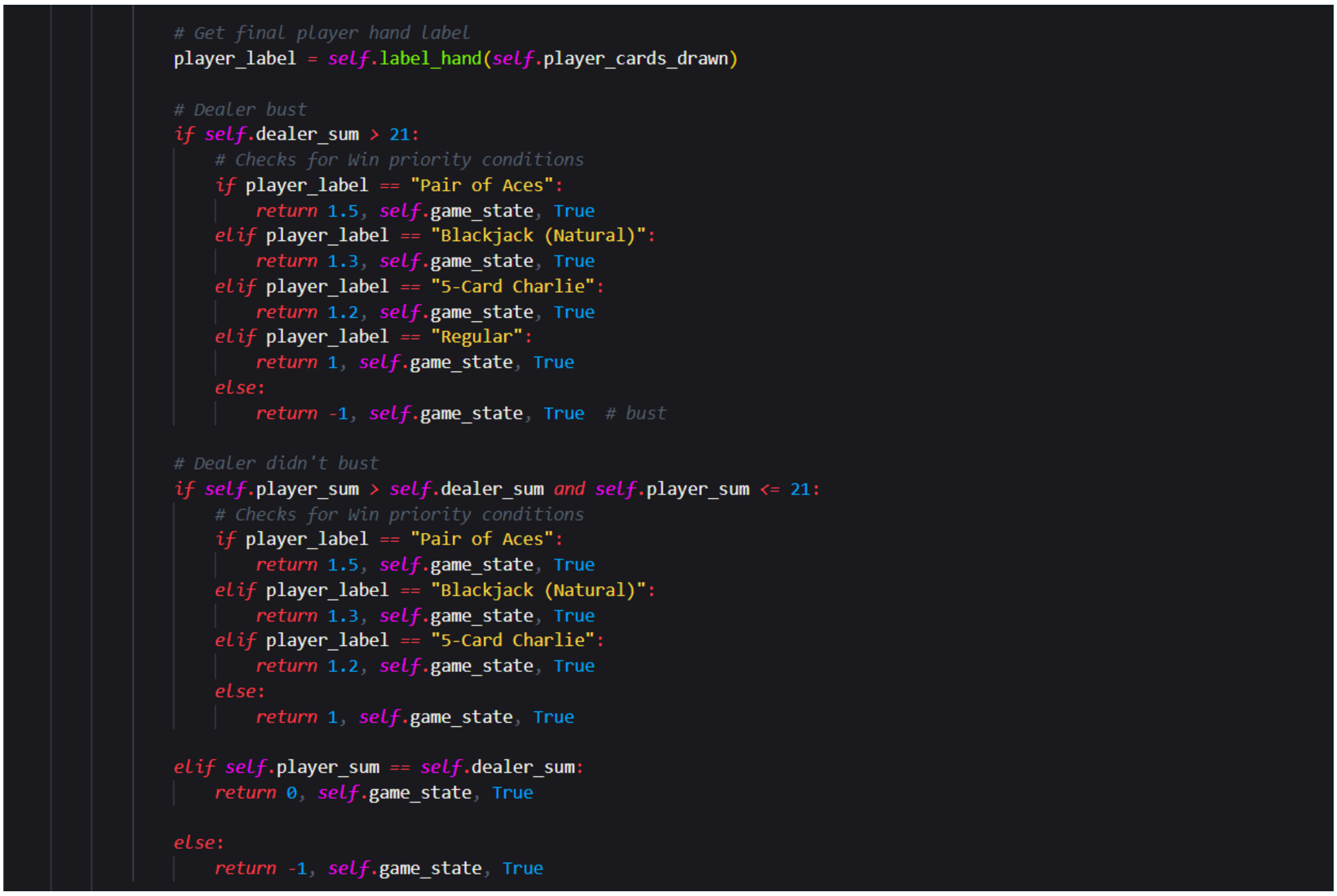
```python
    results_dict = {}
    results_dict['Win Rate'] = win / n
    results_dict['Tie Rate'] = tie / n
    results_dict['Loss Rate'] = loss / n
    results_dict['Expected Return'] = (
        results_dict['Win Rate'] * 10      # player wins his bet value
        + results_dict['Loss Rate'] * -10  # player loses his bet value
    )

    return results_dict
```



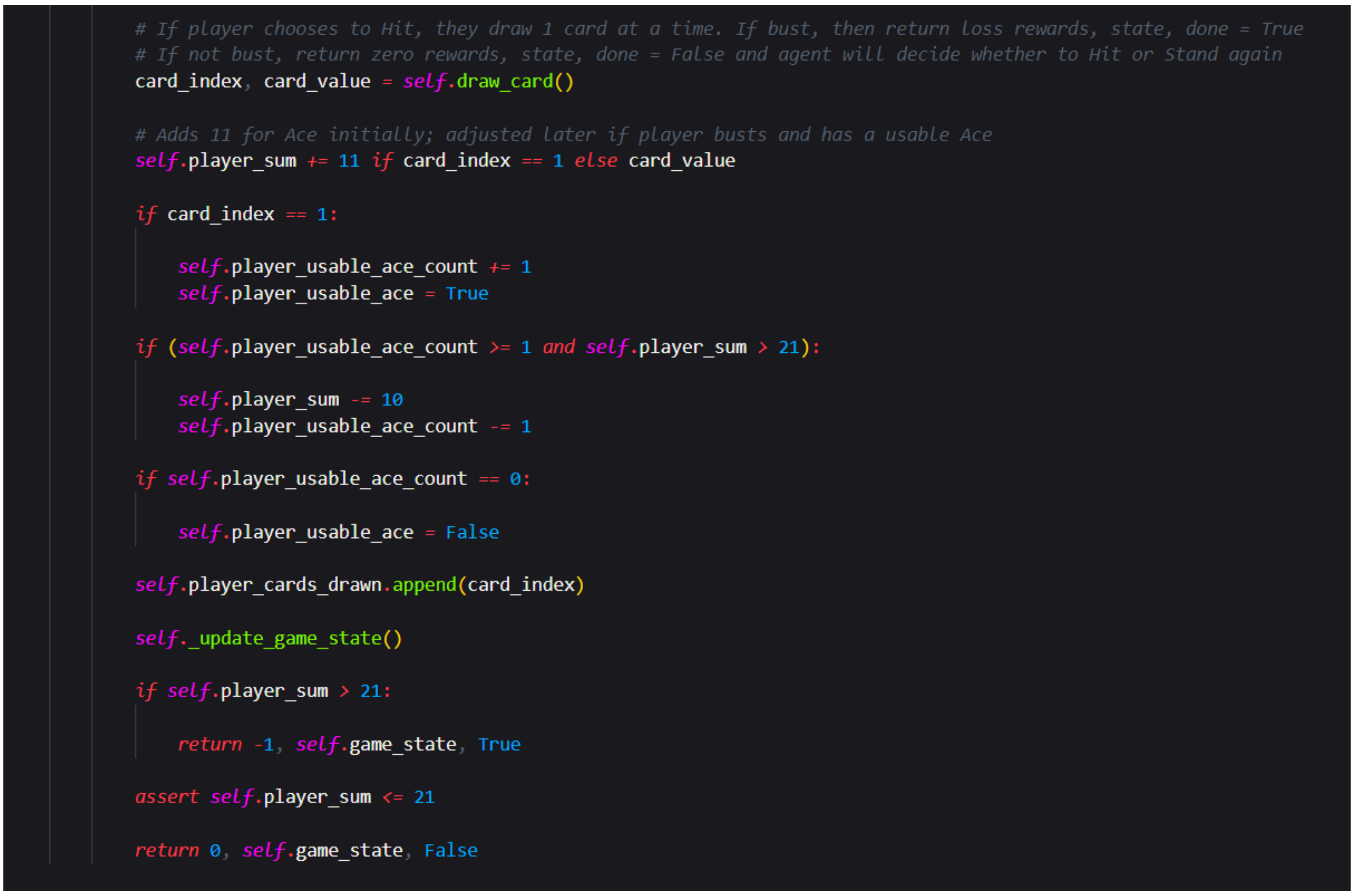
```python
# plotter for Q-sa values
def plotter(agent):

    fig = plt.figure(figsize = (15, 9))

    ax0 = fig.add_subplot(121, projection = '3d')
    ax1 = fig.add_subplot(122, projection = '3d')

    for idx, ua in enumerate(agent.Q_sa):

        player_sums = []
        dealer_visible_cards = []
        max_values = []
        actions = []

        for ps in range(10):

            for dv in range(10):

                max_value = -1e6
                action = None

                for i, v in enumerate(ua[ps][dv]):

                    if v > max_value:

                        max_value = v
                        action = i

                player_sums.append(ps + 12)
                dealer_visible_cards.append(dv)
                max_values.append(max_value)
                actions.append('r' if action == 0 else 'g')
```



```python
        width = depth = 1

        eval(f'ax{idx}').bar3d(dealer_visible_cards, player_sums, 0, width, depth, max_values, shade = True, color = actions)
        eval(f'ax{idx}').set_title(f'{"No " if idx == 0 else ""}Usable Ace')

        eval(f'ax{idx}').set_xticks([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
        eval(f'ax{idx}').set_yticks([12, 13, 14, 15, 16, 17, 18, 19, 20, 21])

        eval(f'ax{idx}').set_xlabel('Dealer displayed card value (Ace = 1)', labelpad = 20)
        eval(f'ax{idx}').set_ylabel('Player Hand Sum', labelpad = 15)
        eval(f'ax{idx}').set_zlabel('Expected Reward following Best Action', labelpad = 10)

        eval(f'ax{idx}').tick_params(axis = "x", pad = 15)
        eval(f'ax{idx}').tick_params(axis = "y", pad = 10)
        eval(f'ax{idx}').tick_params(axis = "z", pad = 5)

    fig.suptitle(f'Final Q_sa plot for {agent.name} after 500k training iterations', fontsize = 16)
    plt.tight_layout()
    plt.show()
```

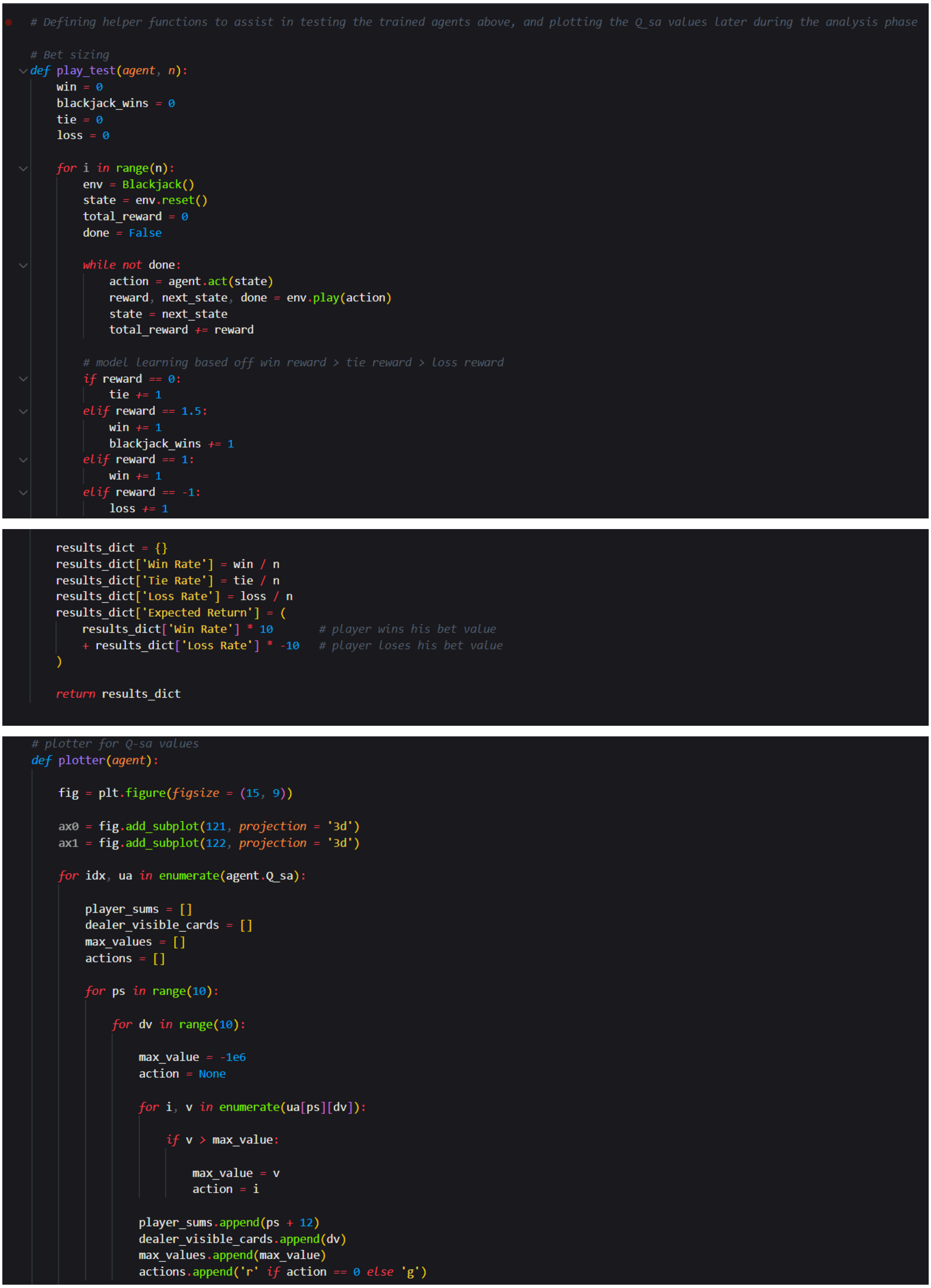


```python
            # Get final player hand label
            player_label = self.label_hand(self.player_cards_drawn)

            # Dealer bust
            if self.dealer_sum > 21:
                # Checks for Win priority conditions
                if player_label == "Pair of Aces":
                    return 1.5, self.game_state, True
                elif player_label == "Blackjack (Natural)":
                    return 1.3, self.game_state, True
                elif player_label == "5-Card Charlie":
                    return 1.2, self.game_state, True
                elif player_label == "Regular":
                    return 1, self.game_state, True
                else:
                    return -1, self.game_state, True  # bust

            # Dealer didn't bust
            if self.player_sum > self.dealer_sum and self.player_sum <= 21:
                # Checks for Win priority conditions
                if player_label == "Pair of Aces":
                    return 1.5, self.game_state, True
                elif player_label == "Blackjack (Natural)":
                    return 1.3, self.game_state, True
                elif player_label == "5-Card Charlie":
                    return 1.2, self.game_state, True
                else:
                    return 1, self.game_state, True

            elif self.player_sum == self.dealer_sum:
                return 0, self.game_state, True

            else:
                return -1, self.game_state, True
```



```python
        # If player chooses to Hit, they draw 1 card at a time. If bust, then return loss rewards, state, done = True
        # If not bust, return zero rewards, state, done = False and agent will decide whether to Hit or Stand again
        card_index, card_value = self.draw_card()

        # Adds 11 for Ace initially; adjusted later if player busts and has a usable Ace
        self.player_sum += 11 if card_index == 1 else card_value

        if card_index == 1:

            self.player_usable_ace_count += 1
            self.player_usable_ace = True

        if (self.player_usable_ace_count >= 1 and self.player_sum > 21):

            self.player_sum -= 10
            self.player_usable_ace_count -= 1

        if self.player_usable_ace_count == 0:

            self.player_usable_ace = False

        self.player_cards_drawn.append(card_index)

        self._update_game_state()

        if self.player_sum > 21:

            return -1, self.game_state, True

        assert self.player_sum <= 21

        return 0, self.game_state, False
```

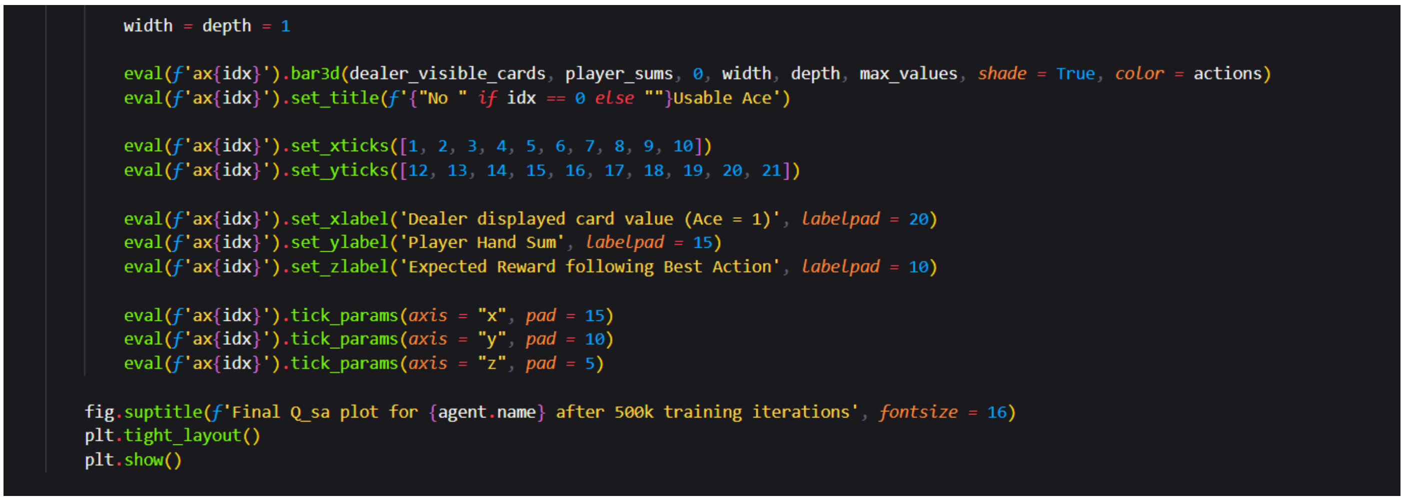
Figure 6.1.b



```python
# Defining helper functions to assist in testing the trained agents above, and plotting the Q_sa values later during the analysis phase

# Bet sizing
def play_test(agent, n):
    win = 0
    blackjack_wins = 0
    tie = 0
    loss = 0

    for i in range(n):
        env = Blackjack()
        state = env.reset()
        total_reward = 0
        done = False

        while not done:
            action = agent.act(state)
            reward, next_state, done = env.play(action)
            state = next_state
            total_reward += reward

        # model learning based off win reward > tie reward > loss reward
        if reward == 0:
            tie += 1
        elif reward == 1.5:
            win += 1
            blackjack_wins += 1
        elif reward == 1:
            win += 1
        elif reward == -1:
            loss += 1
```

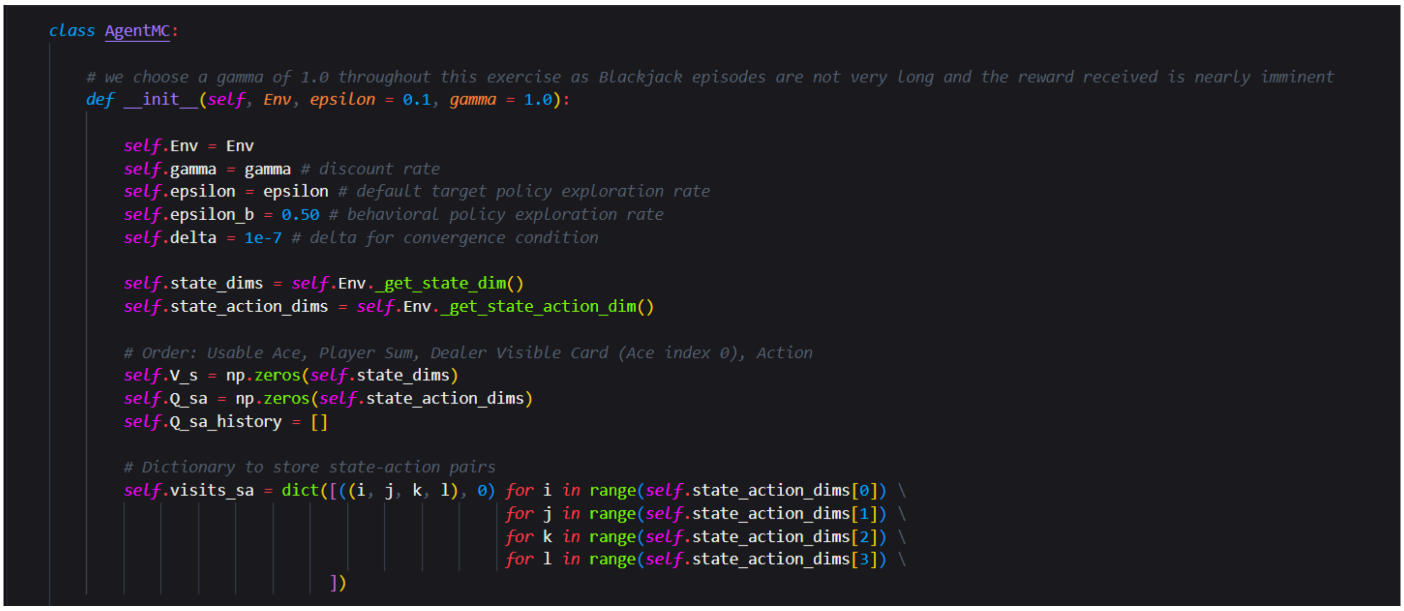
```python
    results_dict = {}
    results_dict['Win Rate'] = win / n
    results_dict['Tie Rate'] = tie / n
    results_dict['Loss Rate'] = loss / n
    results_dict['Expected Return'] = (
        results_dict['Win Rate'] * 10       # player wins his bet value
        + results_dict['Loss Rate'] * -10   # player loses his bet value
    )

    return results_dict
```

```python
# plotter for Q-sa values
def plotter(agent):

    fig = plt.figure(figsize = (15, 9))

    ax0 = fig.add_subplot(121, projection = '3d')
    ax1 = fig.add_subplot(122, projection = '3d')

    for idx, ua in enumerate(agent.Q_sa):

        player_sums = []
        dealer_visible_cards = []
        max_values = []
        actions = []

        for ps in range(10):

            for dv in range(10):

                max_value = -1e6
                action = None

                for i, v in enumerate(ua[ps][dv]):

                    if v > max_value:

                        max_value = v
                        action = i

                player_sums.append(ps + 12)
                dealer_visible_cards.append(dv)
                max_values.append(max_value)
                actions.append('r' if action == 0 else 'g')
```



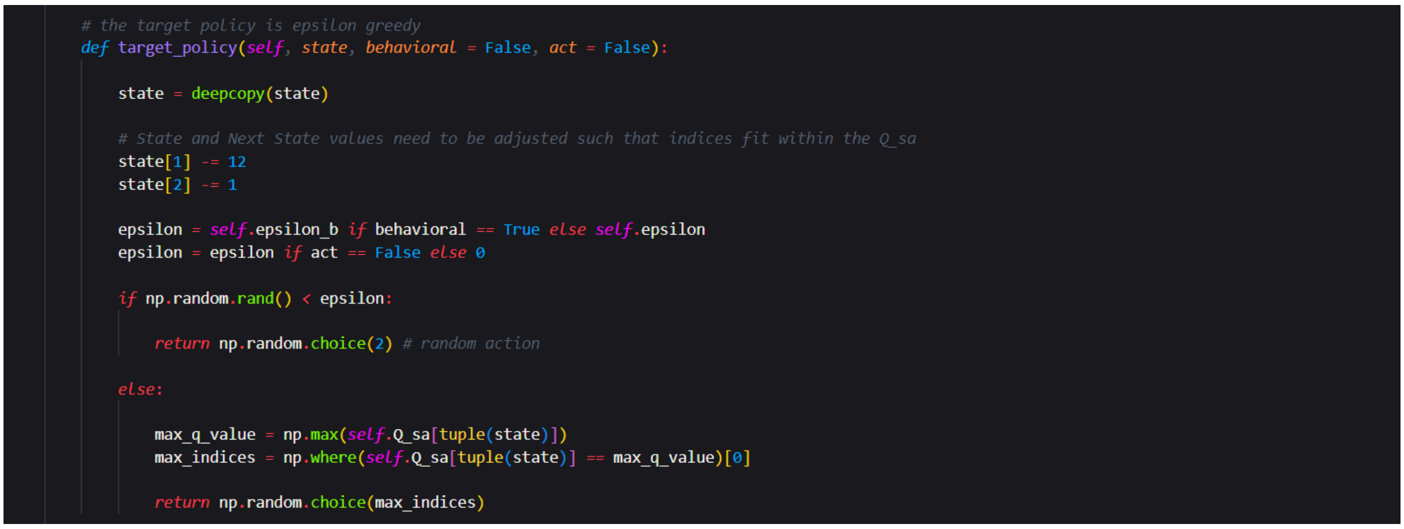
```python
        width = depth = 1

        eval(f'ax{idx}').bar3d(dealer_visible_cards, player_sums, 0, width, depth, max_values, shade = True, color = actions)
        eval(f'ax{idx}').set_title(f'{"No " if idx == 0 else ""}Usable Ace')

        eval(f'ax{idx}').set_xticks([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
        eval(f'ax{idx}').set_yticks([12, 13, 14, 15, 16, 17, 18, 19, 20, 21])

        eval(f'ax{idx}').set_xlabel('Dealer displayed card value (Ace = 1)', labelpad = 20)
        eval(f'ax{idx}').set_ylabel('Player Hand Sum', labelpad = 15)
        eval(f'ax{idx}').set_zlabel('Expected Reward following Best Action', labelpad = 10)

        eval(f'ax{idx}').tick_params(axis = "x", pad = 15)
        eval(f'ax{idx}').tick_params(axis = "y", pad = 10)
        eval(f'ax{idx}').tick_params(axis = "z", pad = 5)

    fig.suptitle(f'Final Q_sa plot for {agent.name} after 500k training iterations', fontsize = 16)
    plt.tight_layout()
    plt.show()
```

Figure 6.2.a

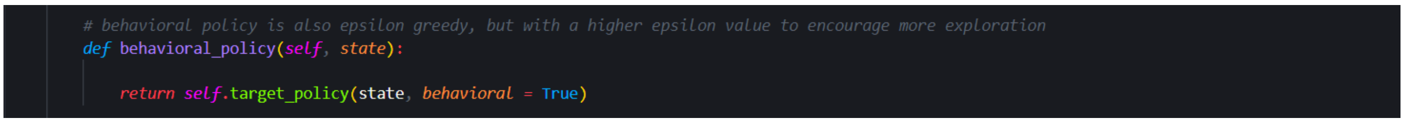


```python
class AgentMC:

    # we choose a gamma of 1.0 throughout this exercise as Blackjack episodes are not very long and the reward received is nearly imminent
    def __init__(self, Env, epsilon = 0.1, gamma = 1.0):

        self.Env = Env
        self.gamma = gamma # discount rate
        self.epsilon = epsilon # default target policy exploration rate
        self.epsilon_b = 0.50 # behavioral policy exploration rate
        self.delta = 1e-7 # delta for convergence condition

        self.state_dims = self.Env._get_state_dim()
        self.state_action_dims = self.Env._get_state_action_dim()

        # Order: Usable Ace, Player Sum, Dealer Visible Card (Ace index 0), Action
        self.V_s = np.zeros(self.state_dims)
        self.Q_sa = np.zeros(self.state_action_dims)
        self.Q_sa_history = []

        # Dictionary to store state-action pairs
        self.visits_sa = dict([((i, j, k, l), 0) for i in range(self.state_action_dims[0]) \
                                                 for j in range(self.state_action_dims[1]) \
                                                 for k in range(self.state_action_dims[2]) \
                                                 for l in range(self.state_action_dims[3]) \
                              ])
```

Figure 6.2.b

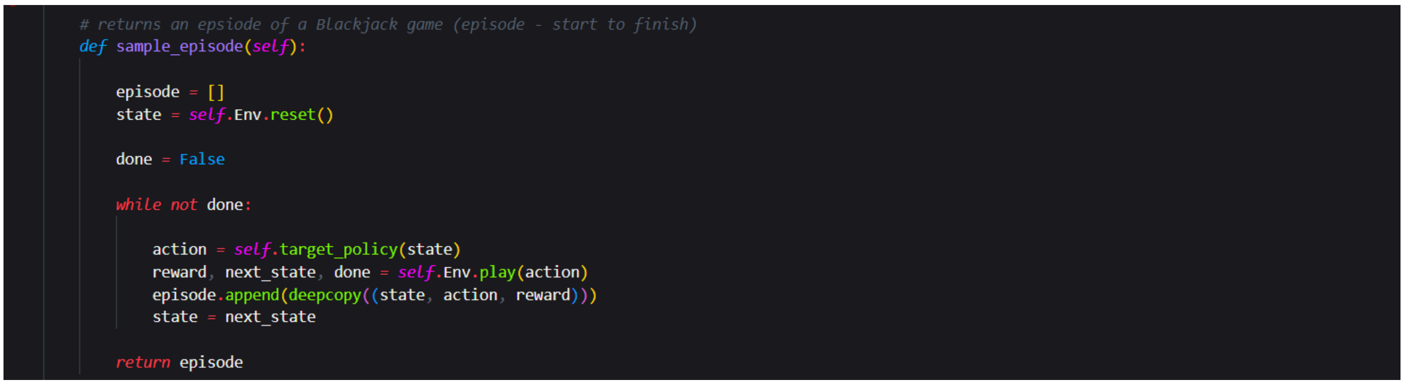


```python
    # the target policy is epsilon greedy
    def target_policy(self, state, behavioral = False, act = False):

        state = deepcopy(state)

        # State and Next State values need to be adjusted such that indices fit within the Q_sa
        state[1] -= 12
        state[2] -= 1

        epsilon = self.epsilon_b if behavioral == True else self.epsilon
        epsilon = epsilon if act == False else 0

        if np.random.rand() < epsilon:

            return np.random.choice(2) # random action

        else:

            max_q_value = np.max(self.Q_sa[tuple(state)])
            max_indices = np.where(self.Q_sa[tuple(state)] == max_q_value)[0]

            return np.random.choice(max_indices)
```

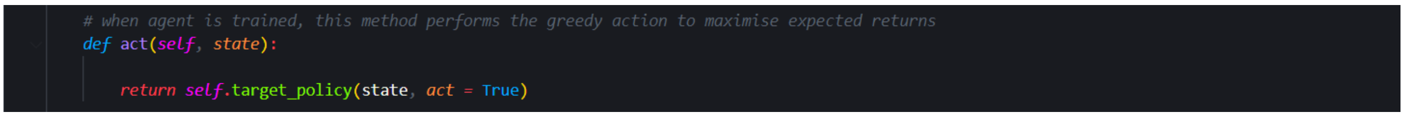


```python
    # behavioral policy is also epsilon greedy, but with a higher epsilon value to encourage more exploration
    def behavioral_policy(self, state):

        return self.target_policy(state, behavioral = True)
```

Figure 6.2.c

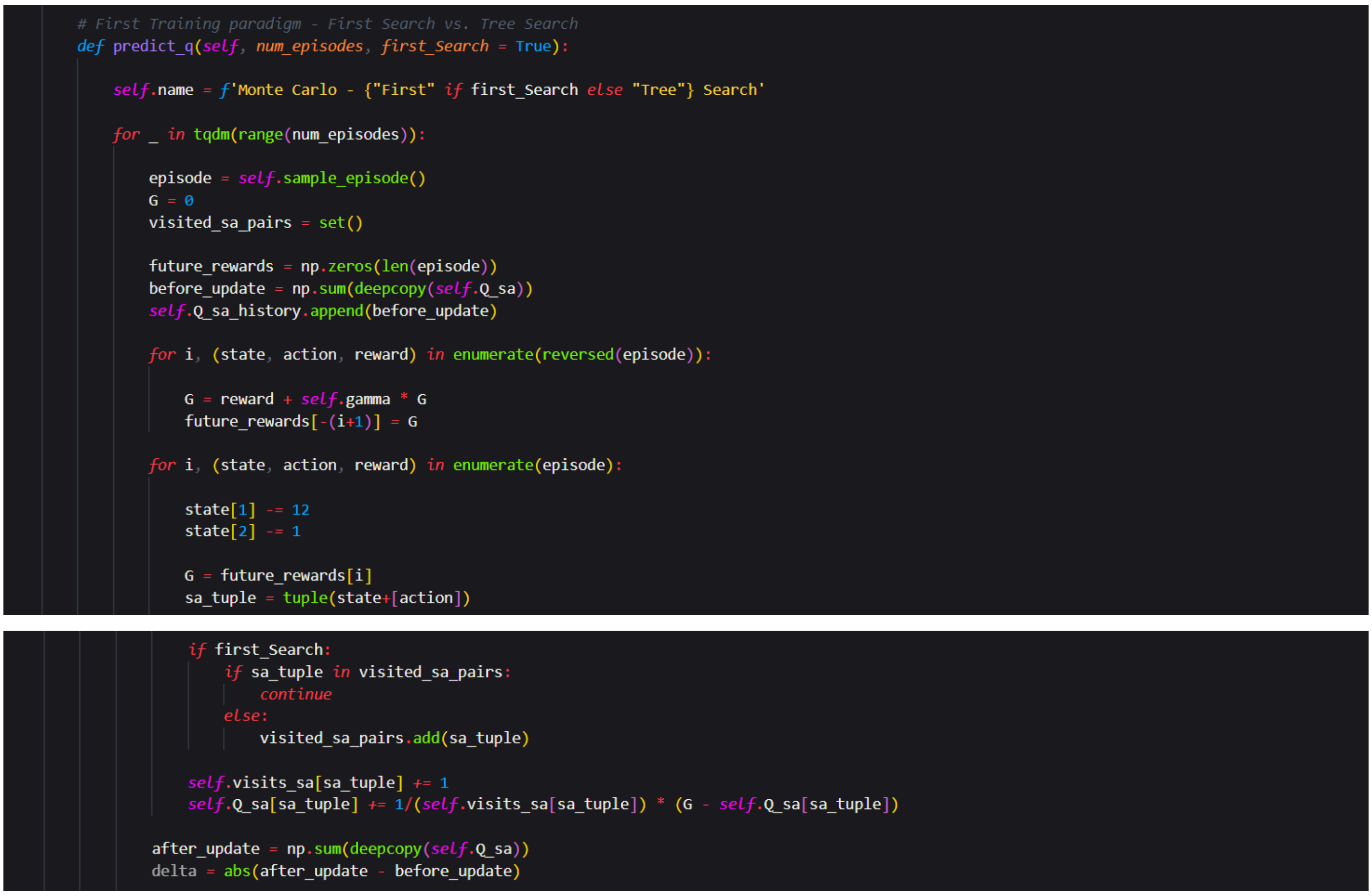


```python
    # returns an epsiode of a Blackjack game (episode - start to finish)
    def sample_episode(self):

        episode = []
        state = self.Env.reset()

        done = False

        while not done:

            action = self.target_policy(state)
            reward, next_state, done = self.Env.play(action)
            episode.append(deepcopy((state, action, reward)))
            state = next_state

        return episode
```



```python
# when agent is trained, this method performs the greedy action to maximise expected returns
def act(self, state):

    return self.target_policy(state, act = True)
```

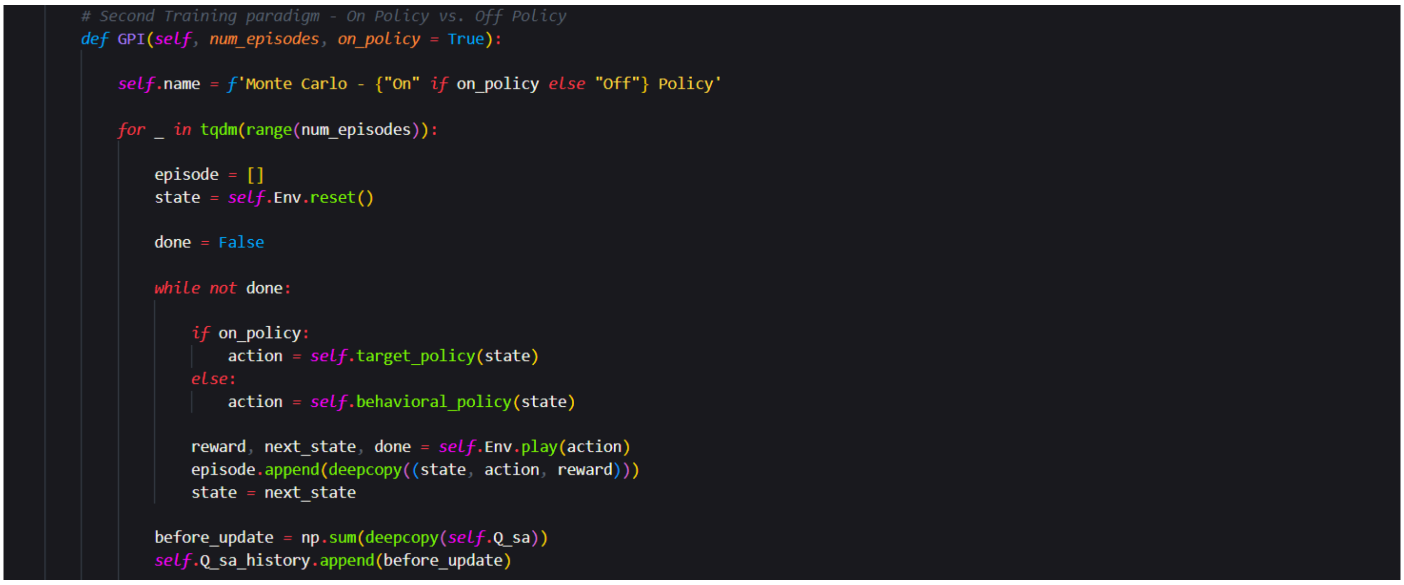
Figure 6.2.d



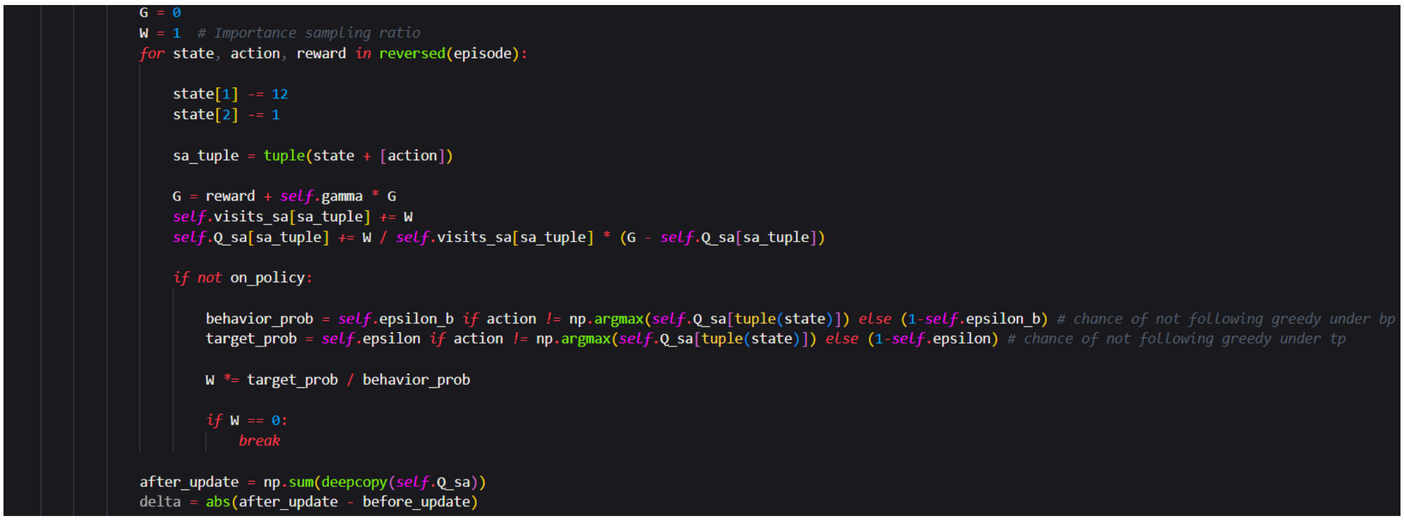
```python
# First Training paradigm - First Search vs. Tree Search
def predict_q(self, num_episodes, first_Search = True):

    self.name = f'Monte Carlo - {"First" if first_Search else "Tree"} Search'

    for _ in tqdm(range(num_episodes)):

        episode = self.sample_episode()
        G = 0
        visited_sa_pairs = set()

        future_rewards = np.zeros(len(episode))
        before_update = np.sum(deepcopy(self.Q_sa))
        self.Q_sa_history.append(before_update)

        for i, (state, action, reward) in enumerate(reversed(episode)):

            G = reward + self.gamma * G
            future_rewards[-(i+1)] = G

        for i, (state, action, reward) in enumerate(episode):

            state[1] -= 12
            state[2] -= 1

            G = future_rewards[i]
            sa_tuple = tuple(state+[action])
```

```python
            if first_Search:
                if sa_tuple in visited_sa_pairs:
                    continue
                else:
                    visited_sa_pairs.add(sa_tuple)

            self.visits_sa[sa_tuple] += 1
            self.Q_sa[sa_tuple] += 1/(self.visits_sa[sa_tuple]) * (G - self.Q_sa[sa_tuple])

        after_update = np.sum(deepcopy(self.Q_sa))
        delta = abs(after_update - before_update)
```

Figure 6.2.e

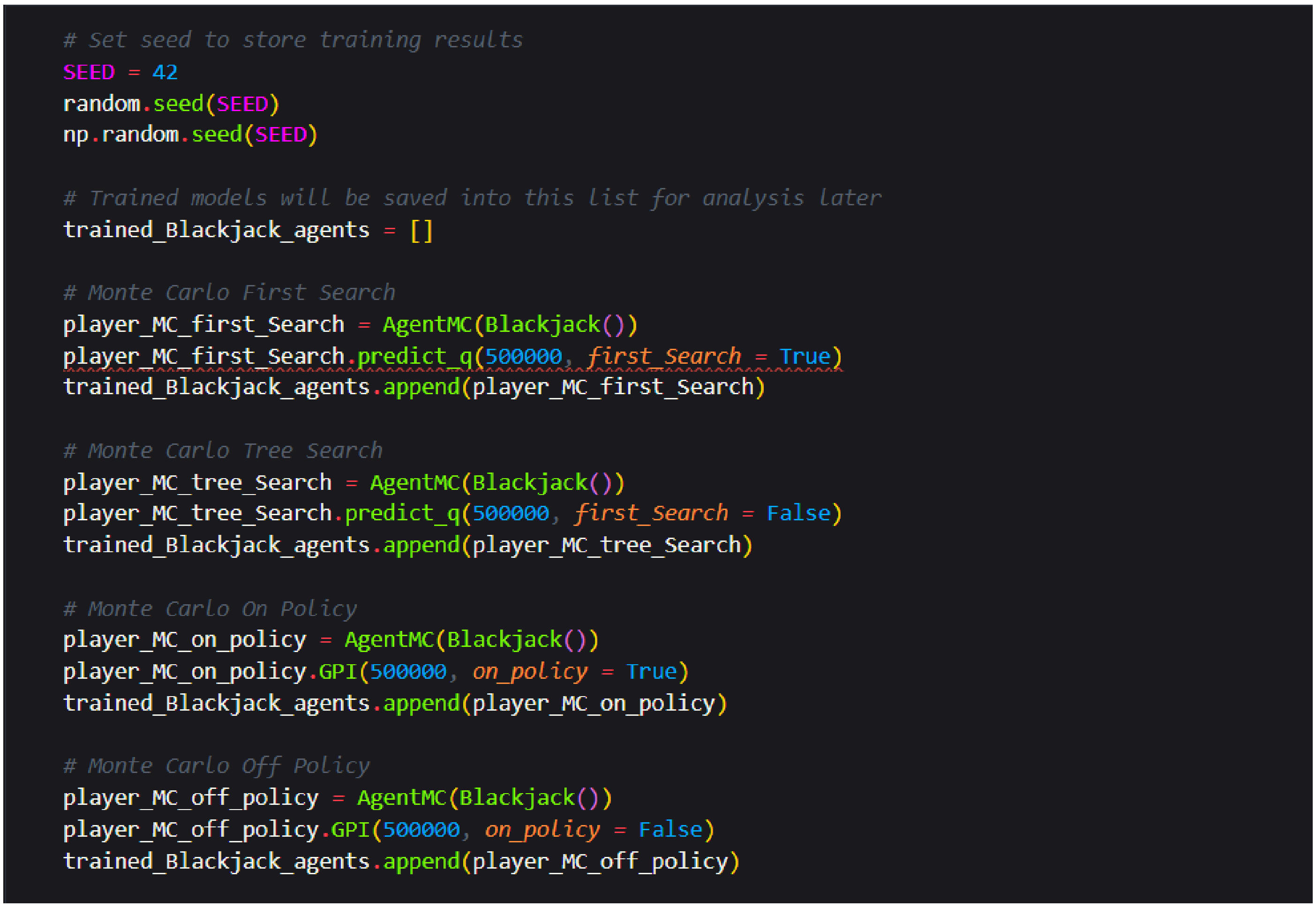


```python
# Second Training paradigm - On Policy vs. Off Policy
def GPI(self, num_episodes, on_policy = True):

    self.name = f'Monte Carlo - {"On" if on_policy else "Off"} Policy'

    for _ in tqdm(range(num_episodes)):

        episode = []
        state = self.Env.reset()

        done = False

        while not done:

            if on_policy:
                action = self.target_policy(state)
            else:
                action = self.behavioral_policy(state)

            reward, next_state, done = self.Env.play(action)
            episode.append(deepcopy((state, action, reward)))
            state = next_state

        before_update = np.sum(deepcopy(self.Q_sa))
        self.Q_sa_history.append(before_update)
```



```
                G = 0
                W = 1  # Importance sampling ratio
                for state, action, reward in reversed(episode):

                    state[1] -= 12
                    state[2] -= 1

                    sa_tuple = tuple(state + [action])

                    G = reward + self.gamma * G
                    self.visits_sa[sa_tuple] += W
                    self.Q_sa[sa_tuple] += W / self.visits_sa[sa_tuple] * (G - self.Q_sa[sa_tuple])

                    if not on_policy:

                        behavior_prob = self.epsilon_b if action != np.argmax(self.Q_sa[tuple(state)]) else (1-self.epsilon_b) # chance of not following greedy under bp
                        target_prob = self.epsilon if action != np.argmax(self.Q_sa[tuple(state)]) else (1-self.epsilon) # chance of not following greedy under tp

                        W *= target_prob / behavior_prob

                        if W == 0:
                            break

                after_update = np.sum(deepcopy(self.Q_sa))
                delta = abs(after_update - before_update)
```

Figure 6.3.a



```
# Set seed to store training results
SEED = 42
random.seed(SEED)
np.random.seed(SEED)

# Trained models will be saved into this list for analysis later
trained_Blackjack_agents = []

# Monte Carlo First Search
player_MC_first_Search = AgentMC(Blackjack())
player_MC_first_Search.predict_q(500000, first_Search = True)
trained_Blackjack_agents.append(player_MC_first_Search)

# Monte Carlo Tree Search
player_MC_tree_Search = AgentMC(Blackjack())
player_MC_tree_Search.predict_q(500000, first_Search = False)
trained_Blackjack_agents.append(player_MC_tree_Search)

# Monte Carlo On Policy
player_MC_on_policy = AgentMC(Blackjack())
player_MC_on_policy.GPI(500000, on_policy = True)
trained_Blackjack_agents.append(player_MC_on_policy)

# Monte Carlo Off Policy
player_MC_off_policy = AgentMC(Blackjack())
player_MC_off_policy.GPI(500000, on_policy = False)
trained_Blackjack_agents.append(player_MC_off_policy)
```