Blackjack Game in Python

Console and GUI implementation using Object-Oriented Programming and Pygame

Alina Akopian

Project Report **May 2025**

Institut für Verkehrsplanung und Transportsysteme
Institute for Transport Planning and Systems

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Blackjack Game in Python

Alina Akopian
ETH Zürich
`alina.akopian@student.ethz.ch`

May 2025

Abstract

This report documents the design and implementation of a Blackjack game in Python, featuring both a console-based and graphical user interface using Pygame. The project explores object-oriented programming principles, state-driven game logic, and user experience design. The final product is a playable and extendable card game that mirrors standard Blackjack rules and includes future extensibility points such as animations, sound, and multi-player support.

Keywords

Python, Blackjack, GUI, Pygame, Object-Oriented Programming, Game Development

Contents

List of Figures

1 Introduction and Motivation

This project was developed as a final assignment for the course "Introduction to Python and Data Science" at ETH Zurich. It involved creating a full-featured card game, Blackjack, in Python. The goal was to implement the entire logic of the game using object-oriented programming and create a user-friendly graphical interface with Pygame.

The project aimed to combine practical software development skills with creative problem-solving by designing, coding, and refining a working game. Specifically, it focused on applying object-oriented principles, managing game state transitions, and building an interactive interface that visually represents gameplay in real-time.

2 Game Rules and Mechanics

The game of Blackjack is structured around comparing the card totals of a player and a dealer. The main objective is to get as close as possible to a total of 21 without exceeding it. Each player begins with two cards. Number cards count as their face value, face cards count as ten, and Aces count as either eleven or one depending on which value is more advantageous to the hand. The player may choose to hit (take another card) or stand (end their turn). The dealer must hit until they reach at least seventeen. A round ends either when the player busts (goes over 21), the dealer busts, or when both have completed their turns and their totals are compared.

3 Console-Based Implementation

The console version of the game, implemented in `blackjack_game.py`, follows a clean object-oriented structure. Four core classes were used: `Card`, `Deck`, `Hand`, and `Chips`. The `Card` class defines individual playing cards, storing their suit, rank, and corresponding value. The `Deck` class creates a standard 52-card deck, supports shuffling, and allows dealing cards one by one. The `Hand` class tracks the cards in a player or dealer's hand, calculates the total value, and includes logic to adjust the value of Aces when necessary. Finally, the `Chips` class manages the player's betting system, including total chips and bet amounts.

The game loop is managed through the `play_game()` function, which coordinates dealing cards, prompting user decisions, evaluating win/loss conditions, and allowing the user to play multiple rounds. Input handling is done via the terminal, using prompts for betting, hitting, or standing. Logic for win conditions, dealer rules, and chip updates is encapsulated in helper functions such as `player_wins()`, `dealer_busts()`, and `push()`.

4 Graphical User Interface (GUI)

To improve user interaction, the game was extended to a GUI using the Pygame library. The interface includes a structured visual layout with labeled buttons, card images, and dynamic text that updates based on the game state. The screen is configured to a resolution of 1024 by 768 pixels, with a consistent visual theme provided by standardized font sizes, button colors, and card dimensions. Card images are loaded from a local folder and scaled appropriately, with fallbacks in case of missing images.

A state-driven approach governs the game flow in the GUI version. Four game states are defined: `STATE_BETTING`, `STATE_PLAYER`, `STATE_DEALER`, and `STATE_ROUND_OVER`. These determine which actions are available and which components are rendered on screen. Buttons are created with a `Button` class that handles rendering and click detection. The main loop listens for mouse events, updates the game state, and triggers redraws accordingly. The rendering pipeline includes drawing the background, player and dealer cards, chip counts, and active messages.

Figure 1: GUI state flow diagram: from betting to round completion

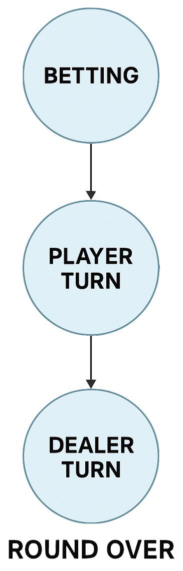

Figure 1 illustrates the core logic of the GUI-based Blackjack implementation. It visually represents the state machine that highlights the game's structure. Starting from the BETTING state, the player sets their wager. Once confirmed, the game transitions to PLAYER TURN, where the player can hit or stand. If the player stands or busts, the game automatically progresses to the DEALER TURN state, in which the dealer draws cards until reaching at least 17 or busting. The game then enters the ROUND OVER state, where the outcome is displayed and the user is prompted to start a new round. This flow ensures that the game remains structured and comprehensible from both a development and user perspective, enabling clean separation of responsibilities and a consistent user experience.

5 Game Logic and Event Handling

The Pygame GUI introduces an event-driven loop that reacts to user input. When the player clicks on a button (e.g., Hit or Stand), the game updates the state accordingly. The `draw_card()` function is used to visually render either the front or back of a card, depending on whether it should be hidden. The dealer's first card, for instance, is hidden until the player ends their turn.

Listing 1: Simplified card drawing logic

```python
def draw_card(card, pos, hidden=False):
    if hidden:
        img = card_images['back']
    else:
        key = f"{card.rank}_of_{card.suit}"
        img = card_images.get(key)
    screen.blit(img, pos)
```

Listing 1 shows the simplified logic for drawing a card on the screen. The function takes a card object and determines whether to show its front (based on rank and suit) or the back image if it is meant to be hidden—this is especially important during the dealer's turn when one card is concealed until the player completes their actions. While the complete GUI implementation includes numerous additional rendering and logic components, this example was chosen to highlight how tightly integrated the data model (the card object) is with the user interface. It demonstrates one of the many bridge points between backend logic and visual feedback in the game.

6 Challenges and Future Extensions

The primary challenge in this project was managing the synchronization between the underlying game logic and the visual representation. Care was taken to ensure that state changes triggered appropriate UI updates and that all edge cases (e.g., busts or ties) were handled correctly. Adjusting for Aces dynamically also introduced complexity in determining optimal hand values. The GUI's modularity and clear state structure, however, helped manage these complexities.

Looking forward, several improvements could be done. Adding animations for card flips and chip movements would improve user experience. Sound effects for different actions such as wins, losses, or card draws would make the game more immersive. Extending the game to support multiple players and advanced Blackjack features like splitting or doubling down would add strategic depth. These features are architecturally supported by the current design and can be added modularly.

7 Conclusion

This project demonstrated the integration of fundamental programming concepts with interactive design. By developing a Blackjack game from scratch, using both terminal-based and graphical interfaces, the full software development cycle was explored: from requirements and structure to implementation and testing. The game provides a solid base for further exploration into more complex user interfaces and game mechanics, and offers a strong example of how Python can be used to create engaging applications with a rich user experience.