PROJECT REPORT

Project Name: Fun Number Game

A Python CLI Application

By: Shresth Sahu

Topic: Software Development & Game Logic Implementation

Aim: To design and build a text-based strategy game that simulates a simplified version of Blackjack, demonstrating core Python programming concepts such as control flow, state management, and random number generation.

1. Introduction

The Fun Number Game is a quick, logic-based variation of Blackjack that runs right in your terminal. I built this in Python to strip away the complexity of suits and face cards, focusing entirely on the thrill of hitting 21. It's you versus the computer in a race to see who can get the highest score without busting, all powered by simple random number generation.

2. Problem Statement

Modern life is incredibly hectic. Between work, studies, and endless notifications, people are constantly under stress and often need a quick, accessible way to decompress. However, finding a game that provides instant gratification without requiring a long time commitment or complex setup can be difficult.

Additionally, for anyone just starting with Python, jumping straight into graphical games like Flappy Bird or Mario can be intimidating. You have to deal with libraries, frame rates, and assets before writing a single line of game logic.

I wanted to solve this by building a game that:

1. **Acts as a Stress Reliever:** It provides a quick, distraction-free escape from daily pressures, offering a few minutes of fun without a heavy mental load.
2. **Runs anywhere:** No installations, just pure Python code.
3. **Teaches the basics:** It's a perfect playground for loops, conditionals, and functions.
4. **Is actually fun:** It's not just a math problem; there's tension and strategy involved.
5. **Simplifies the rules:** By using digits 1-10, I avoided the headache of coding Ace logic (1 vs 11) or face cards, letting me focus on the core mechanics.

3. Functional Requirements

Here is what the game actually does when you run it:

- **Sets the Stage:** Clears the messy console history and welcomes you with a clean banner and rules.
- **The Deal:** hands out two random numbers (1-10) to you and the computer to start things off.

- **Keeps Score:** Constantly adds up your numbers so you don't have to do the math.
- **Lets You Choose:** You type 'Hit' to risk it for another number, or 'Stand' to play it safe.
- **Smart Opponent:** The computer plays by casino rules—it hits until it reaches 17, then stops. No cheating!
- **Decides the Winner:** It checks for busts (over 21), handles the rare "Instant Win" if the computer hits 21, and declares a winner or a draw at the end.
- **Play Again:** Lets you loop right back into a new game without restarting the program.

4. Non-Functional Requirements

- **Look & Feel:** Even though it's text-based, I used ASCII art (stars and dashes) to make it look organized and readable.
- **Speed:** The game feels snappy. There’s no loading time.
- **User-Proofing:** If you accidentally type "hitt" or "yes" instead of "Hit", the game won't crash; it just asks you to try again.
- **Compatibility:** If you have Python installed, this game works. Windows, Mac, or Linux.

5. System Architecture

I kept the architecture straightforward to match the simplicity of the game.

- **Input:** I use the standard input() function to talk to the player.
- **The Brain (Logic):** I broke the game into small, manageable helpers: rng() for getting numbers and calc() for doing the math. The run() function acts as the main director.
- **Display:** Simple print() statements handle the visuals, and I check the OS to run the right "clear screen" command (cls or clear).
- **Memory:** I didn't need a database here. A simple Python List ([]) acts as temporary storage for the hands during the game.

6. Design Diagrams

6.1 Use Case Diagram

- **Who plays?** Just you (The Player).
- **What can you do?**
    - Launch the game.
    - Check your current hand.
    - Ask for a card (Hit).
    - End your turn (Stand).
    - Win, Lose, or Draw.

6.2 Sequence Diagram (The Flow)

```
sequenceDiagram
    participant Me as Player
    participant Game as The Game
    participant CPU as Computer

    Game->>Me: Here are your first two numbers.
    Game->>Me: Here is your score.
    loop My Turn
        Me->>Game: I want to "Hit"
        Game->>Game: Picks a random number
        Game->>Me: You got a 7! New score: 20.
        Game->>CPU: Are you under 17?
        opt CPU is low
            CPU->>CPU: I'll take a number too.
        end
    end
    Me->>Game: I'll "Stand"
    loop Catch Up
```

```
    opt CPU < 17
      CPU->>CPU: Hits until satisfied
    end
  end
  Game->>Me: Here is the winner!
```

7. Design Decisions & Rationale

- **Short Variable Names:** I used names like p (player) and cpu (computer) instead of long, descriptive names. It gives the code a scrappy, "hacker" vibe and makes it faster to read.
- **No Cards, Just Digits:** I stripped out the concept of "Hearts" or "Spades". It keeps the code focused on the math, which is the core of the game anyway.
- **The "Instant 21" Rule:** I decided that if the computer hits exactly 21, they win immediately (unless you have 21 too). It adds a bit of unfair tension, making the computer feel like a tough "boss" opponent.
- **Clearing the Screen:** I force the screen to clear every round. It stops the terminal from becoming a giant wall of text, making each round feel fresh.

8. Implementation Details

The game is built in **Python**. Here are the cool technical bits:

- **Randomness:** I rely heavily on random.randint(1, 10). It's the engine that powers the unpredictability of the game.
- **OS Detection:** I explicitly check os.name to see if you're on Windows or Mac so I can clear the screen correctly. It’s a small detail that adds polish.
- **Input Cleaning:** I use .strip().lower() on whatever you type. This handles extra spaces or capitalization issues gracefully.
- **The Loop:** I use a while active: loop. This lets me break out of the game instantly if someone wins, or keep it running as long as needed.

9. Simulation Results

Scenario: A Tense Win

```
*****************************************
    WELCOME TO THE FUN NUMBER GAME
       (Try to hit 21!)
*****************************************
... [Rules Omitted] ...

Your Hand:    [5, 8]  (Score: 13)
Computer Hand: [2, 9]  (Score: 11)

Type 'Hit' for another number, or 'Stand' to stop: hit
You drew a 7!
Computer decides to hit... and gets 4

Your Hand:    [5, 8, 7]  (Score: 20)
Computer Hand: [2, 9, 4]  (Score: 15)

Type 'Hit' for another number, or 'Stand' to stop: stand

You chose to Stand.
Computer needs to hit (under 17)...
Computer drew a 8

Computer Hand: [2, 9, 4, 8]  (Score: 23)
Computer went over 21! You win!
```

10. Testing Approach

I didn't just write it and hope for the best. I tested it:

- **The Play Test:** I played about 20 rounds to make sure the numbers felt random and "fair."
- **Breaking It:** I tried typing garbage inputs, hitting until I busted on purpose, and standing immediately with a score of 2.
- **The Edge Cases:** I specifically looked for what happens when *both* players hit 21. Initially, it was buggy, but I fixed it to ensure it calls a "Draw."

11. Challenges I Faced

1. **The "Catch Up" Bug:** At first, the computer would only draw *one* card after I stood. I realized I needed a loop there so the computer keeps hitting until it hits 17.
2. **The Double 21:** Handling the moment where both players hit 21 was tricky. I had to add specific logic to check the player's score *inside* the computer's win condition to make sure I didn't accidentally tell the player they lost when it was actually a tie.
3. **Simultaneous Turns:** Originally, the game felt static because the computer only played after I finished my entire turn. I wanted it to feel like a real-time race, so I adjusted the logic to let the computer hit alongside me if its score was low enough, making the game feel much more alive.

12. Learnings & Key Takeaways

- **State is Everything:** Managing active = True vs False taught me a lot about how game loops work.
- **UX Matters:** Even in a text window, little things like * dividers and clear prompts make the difference between a confusing script and a fun game.
- **Functions are Friends:** Breaking the "calc" and "rng" logic out of the main loop made the code so much easier to read and debug.

13. Future Enhancements

If I had more time, here is what I'd add:

- **Chips & Betting:** Give the player $1000 to start and let them bet on hands.
- **Double Down:** A high-risk move to double the bet for exactly one more card.
- **Difficulty Modes:** An "Easy" mode where the computer makes mistakes, and a "Hard" mode where it plays optimally.
- **Save Files:** Writing your high score to a .txt file so you can brag about it later.

14. References

- Python 3 random library documentation.
- Standard Blackjack rules (simplified for my sanity).