Project Report: Simple Number Guessing Game

BY

B.Tech First Year Student

Alok Kumar Sharma

(Reg. No: 25BAI10346)

November 25, 2025

Submitted in Partial Fulfilment for the Course
Introduction to Programming (CSE1021)

VIT BHOPAL UNIVERSITY

Contents

1 Introduction

This project is a simple, command-line based number guessing game implemented in Python. The goal of the game is for the user to guess a secret number chosen randomly by the computer within a defined range (1 to 100) and a limited number of attempts (7 tries). The system provides hints (higher or lower) after each guess to help the user narrow down the possibilities. The project demonstrates fundamental programming concepts like input validation, loops, conditional statements, and using basic Python libraries.

2 Problem Statement

To develop an interactive and engaging console application where a user can play a classic number guessing game. The application must accurately select a random number, handle user input robustly, provide feedback, and determine if the user wins or loses within a fixed number of attempts. Furthermore, the application should provide a simple analytical review of the user's guessing strategy (probability and optimal tries needed) at the end.

3 Functional Requirements

1. **Random Number Generation:** The game must generate a random secret number between 1 and 100.
2. **Input Validation:** The system must ensure the user's guess is a valid integer and falls within the allowed range (1-100).
3. **Uniqueness Check:** The system must reject a guess that the user has already made.
4. **Hint Mechanism:** After each incorrect guess, the system must provide feedback: "Try a higher number" or "Try a lower number."
5. **Attempt Tracking:** The game must limit the user to a maximum of 7 attempts.
6. **Game End:** The game must clearly announce a win or a loss and display the secret number upon termination.

4 Non-functional Requirements

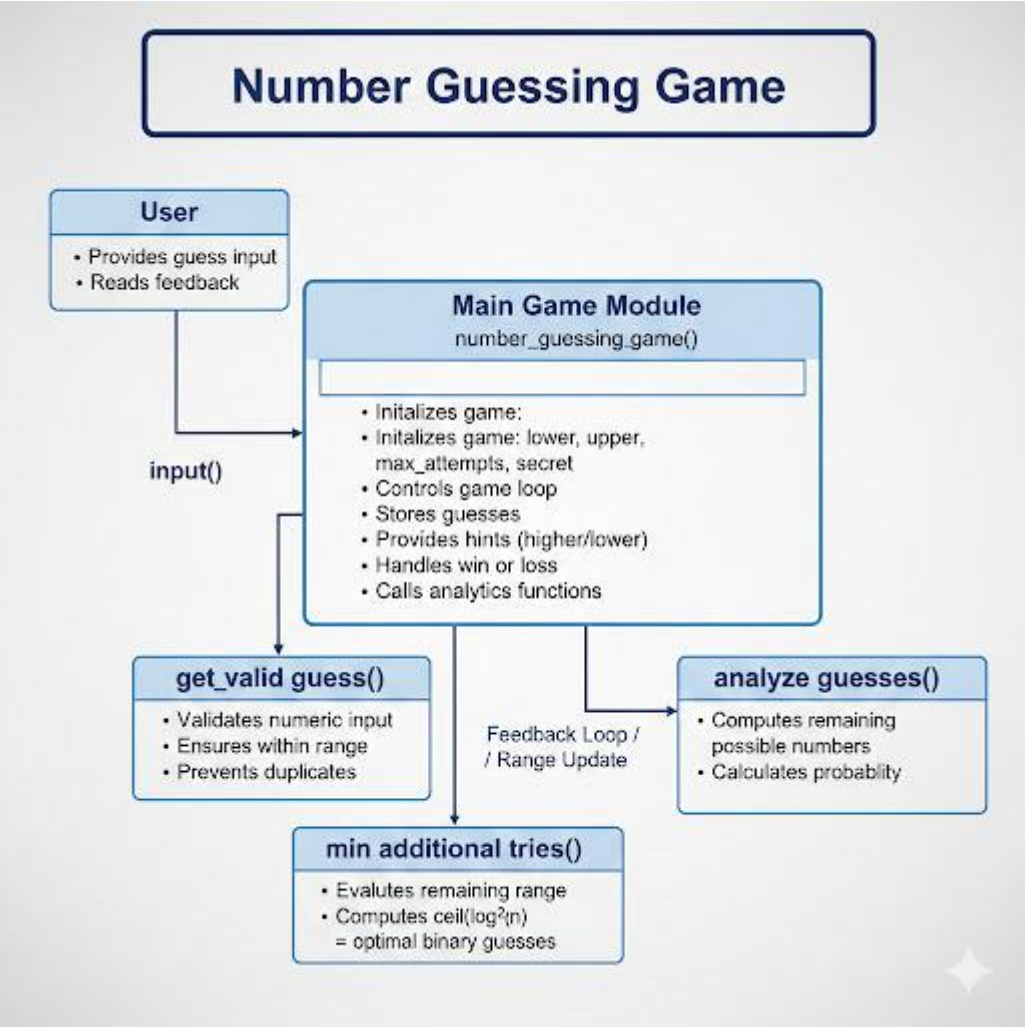
- **Usability:** The console interface should be clear and easy to understand.
- **Reliability:** The program must handle invalid input (e.g., text instead of numbers) without crashing.
- **Simplicity:** The code must be well-structured and easy to maintain.

- **Responsiveness:** The game should respond quickly to user input (a small delay is added using time.sleep for better user experience).

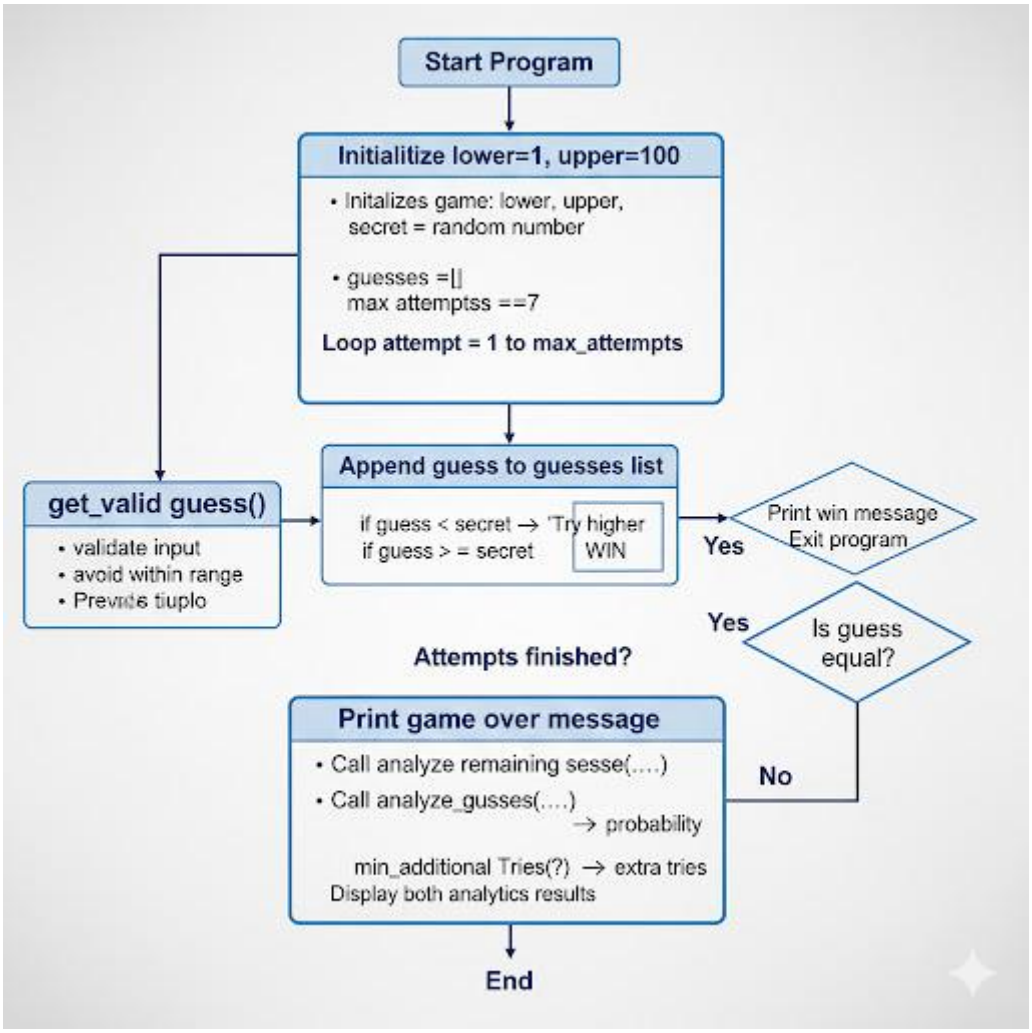
5 System Architecture

The system architecture is a simple, single-tier, monolithic structure, implemented as a standalone Python script. All application logic, user interface (console I/O), and data handling (local variables) are contained within this single file.

- **Input Layer:** Handled by the input() function.
- **Processing Layer:** Handled by the core Python functions (get_valid_guess, number_guessing_game).
- **Output Layer:** Handled by the print() function.



EXECUTION FLOWCHART



6 Design Diagrams

Due to the constraints of a console application and a single-file implementation, the diagrams are conceptually represented or simplified.

```python
import random
import sys
import time


def get_valid_guess(lower, upper, previous_guesses):  1 usage
    while True:
        try:
            guess = input(f"Enter your guess ({lower}-{upper}): ")
            guess = int(guess)
            if guess in previous_guesses:
                print(f"You already guessed {guess}. Try a different number.")
                time.sleep(1)
                continue
            if guess < lower or guess > upper:
                print(f"Please enter a number between {lower} and {upper}.")
                time.sleep(1)
                continue
            return guess
        except ValueError:
            print("Invalid input. Please enter a valid integer.")
            time.sleep(1)


def analyze_guesses(guesses, secret, lower, upper):  1 usage
    possible = set(range(lower, upper + 1))
    for g in guesses:
        if g < secret:
            possible = set(x for x in possible if x > g)
        elif g > secret:
            possible = set(x for x in possible if x < g)
    remaining = len(possible)
    total = upper - lower + 1
    prob = 1 - remaining / total
    return max(0, min(1, prob))


def min_additional_tries(guesses, secret, lower, upper):  1 usage
    # Calculate how many more tries would be needed assuming an optimal binary search on the reduced range
    possible = set(range(lower, upper + 1))
    for g in guesses:
        if g < secret:
            possible = set(x for x in possible if x > g)
        elif g > secret:
            possible = set(x for x in possible if x < g)
    remaining = len(possible)
    # Number of tries in worst-case binary search = ceil(log2(remaining))
    import math
    if remaining <= 1:
        return 0
    return math.ceil(math.log2(remaining))

```

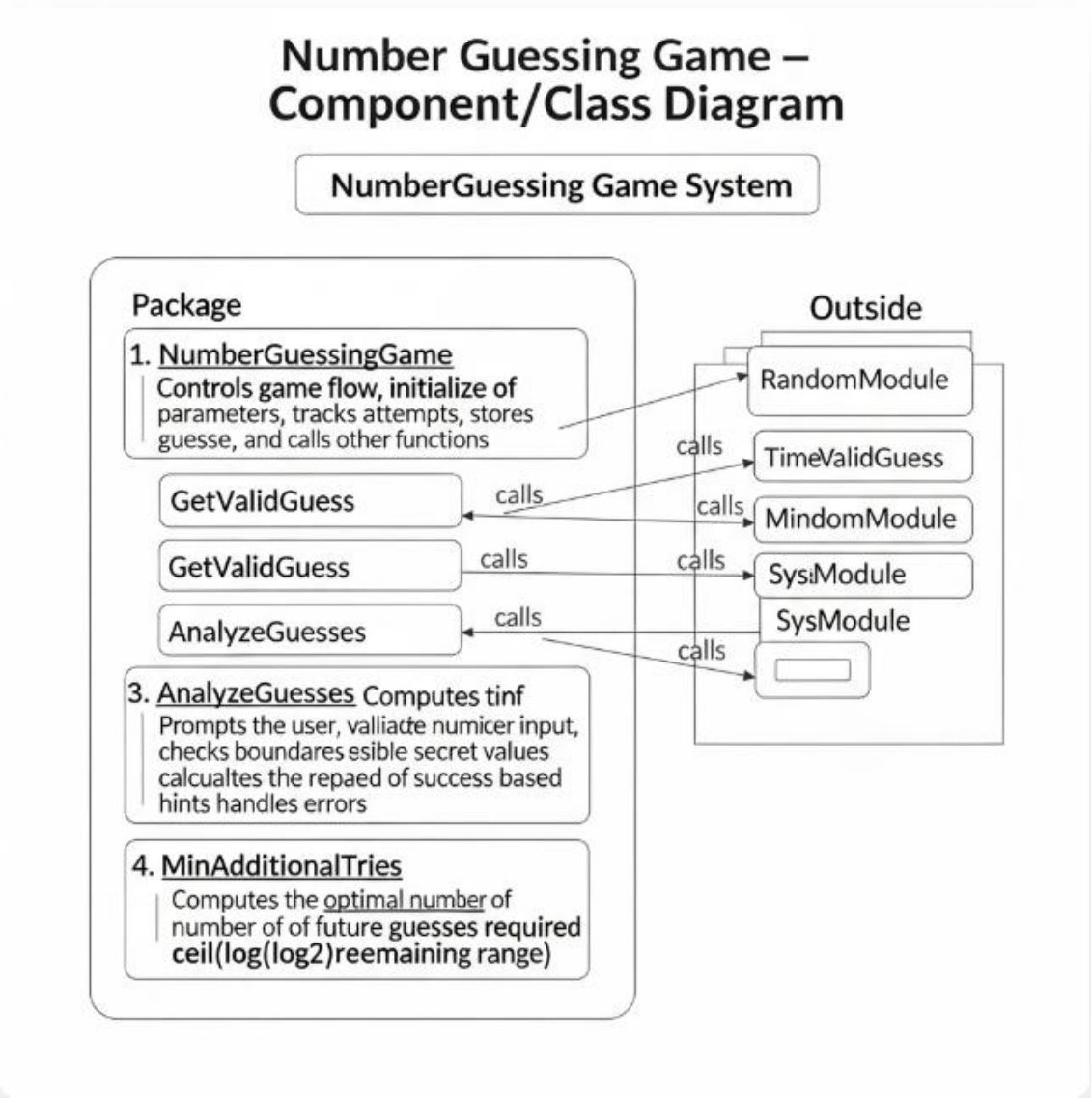
```
51      return math.ceil(math.log2(remaining))
52
53
54  def number_guessing_game():  1 usage
55      lower = 1
56      upper = 100
57      max_attempts = 7
58      secret = random.randint(lower, upper)
59      guesses = []
60
61      print(f"Welcome to the Number Guessing Game!")
62      time.sleep(1)
63      print(f"Guess the number between {lower} and {upper}. You have {max_attempts} tries.")
64
65      for attempt in range(1, max_attempts + 1):
66          guess = get_valid_guess(lower, upper, guesses)
67          guesses.append(guess)
68
69          if guess < secret:
70              print("Try a higher number.")
71          elif guess > secret:
72              print("Try a lower number.")
73          else:
74              print(f"Congratulations! You guessed the number {secret} in {attempt} tries.")
75              sys.exit(0)
76          time.sleep(1)
77      else:
```

```
54  def number_guessing_game():  1 usage
71          elif guess > secret:
72              print("Try a lower number.")
73          else:
74              print(f"Congratulations! You guessed the number {secret} in {attempt} tries.")
75              sys.exit(0)
76          time.sleep(1)
77      else:
78          print(f"Sorry, you've used all your tries. The number was {secret}. Try again!")
79
80      probability = analyze_guesses(guesses, secret, lower, upper)
81      print(f"Based on your answers, your probability of winning was approximately {probability:.2%}.")
82
83      more_tries_needed = min_additional_tries(guesses, secret, lower, upper)
84      print(f"Based on your guesses so far, "
85            f"you could have found the secret number in about {more_tries_needed} more "
86            f"optimally chosen guess{'es' if more_tries_needed != 1 else ''}.")
87
88
89  if __name__ == "__main__":
90      number_guessing_game()
91
```

Component/Class Diagram

The system consists of three main components (functions):

- number_guessing_game(): The main component controlling the game loop.
- get_valid_guess(): Component for handling and validating user input.
- analyze_guesses() / min_additional_tries(): Components for analytical calculations.



ER Diagram

An Entity-Relationship (ER) Diagram is not applicable to this project as no persistent database storage is used. All game data (secret number, guesses, attempts) is stored in memory for the duration of the program execution.

7 Design Decisions & Rationale

- **Python as Language:** Python was chosen for its simplicity, readability, and quick prototyping capabilities, making it ideal for a first-year project.
- **Separate Input Function (`get_valid_guess`):** Isolating input handling into its own function ensures that the main game logic remains clean. It uses a `while True` loop and `try-except` block to repeatedly prompt the user until a valid and new integer guess is provided, ensuring robust error handling.
- **Fixed Range (1-100) and Attempts (7):** These standard settings are used because $\log_2(100) \approx 6.64$, meaning 7 attempts is the mathematically optimal worst-case number needed to guarantee a win using a binary search strategy.
- **Analysis Functions:** The `analyze_guesses` and `min_additional_tries` functions were added to introduce a basic level of statistical analysis, calculating the remaining uncertainty and the minimum optimal future moves.

8 Implementation Details

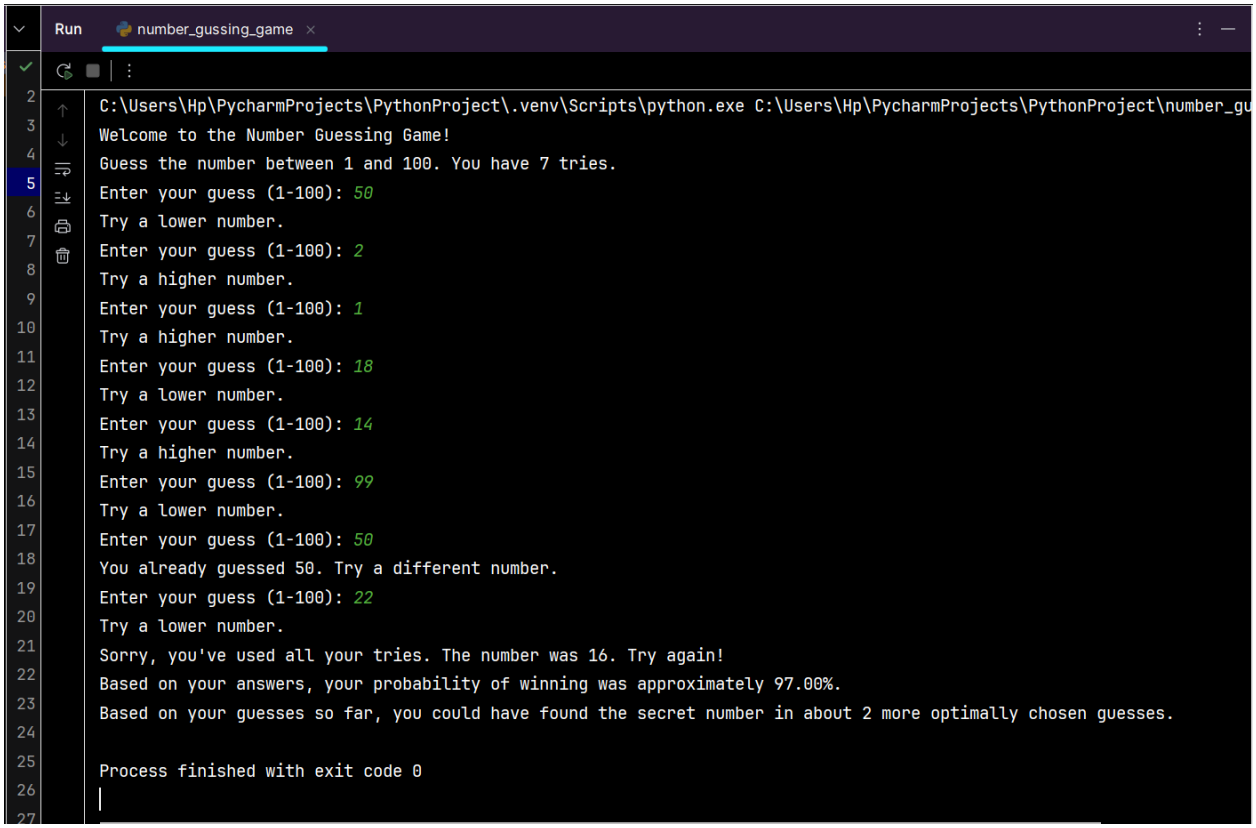
The entire application is implemented in a single Python file.

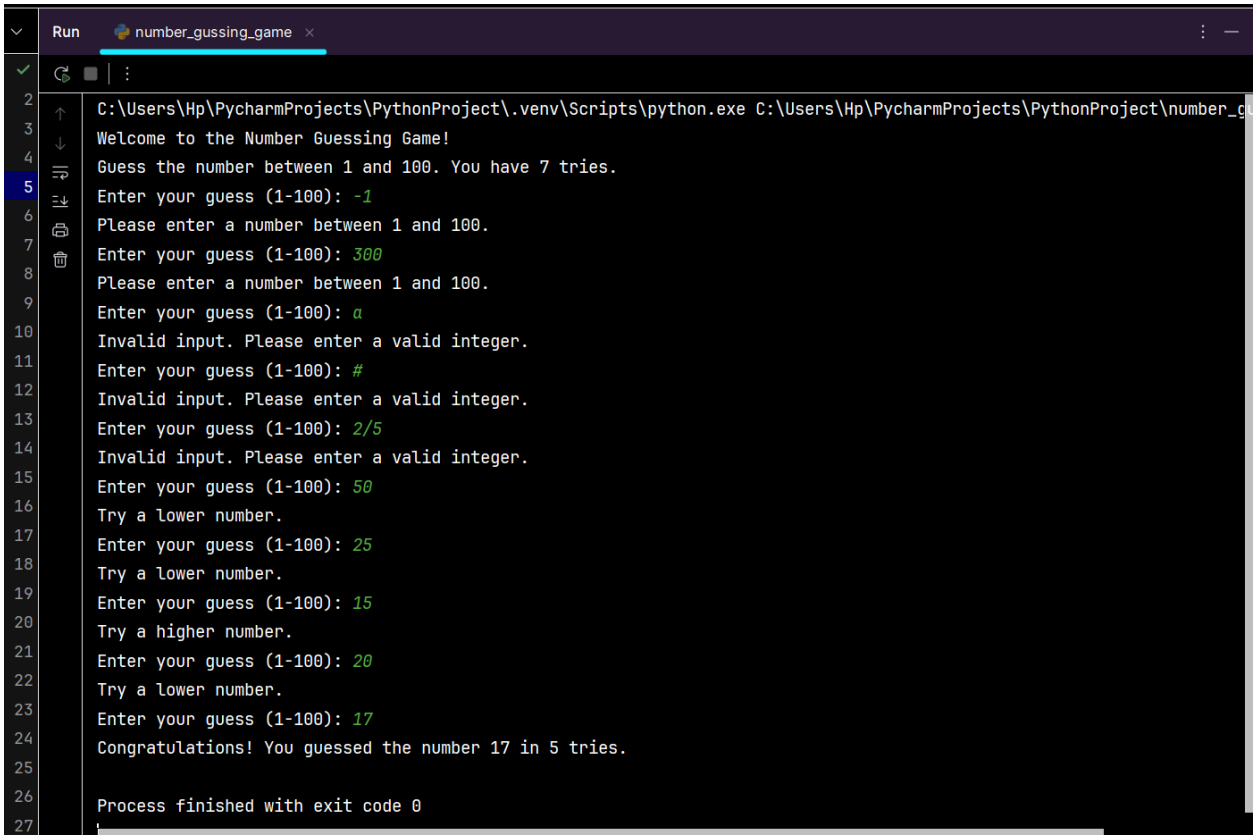
8.1 Key Functions

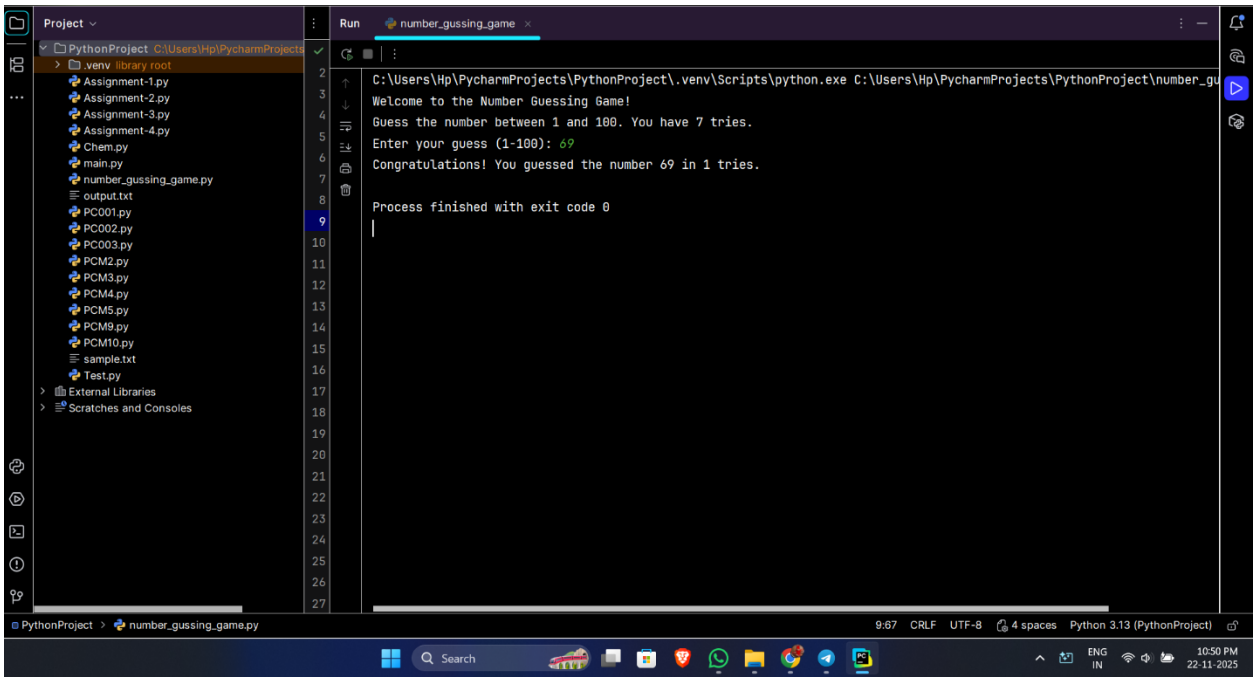
- `get_valid_guess(lower, upper, previous_guesses)`: This function manages user interaction. It uses a `try-except ValueError` to catch non-integer inputs and checks if the number is within the bounds and not a duplicate guess.
- `analyze_guesses(...)`: This function calculates the set of all remaining possible secret numbers based on the user's past hints, and determines the probability of winning if the game were to end at that point.
- `min_additional_tries(...)`: This function computes the number of optimal binary search steps required to find the secret number within the currently known reduced range.
- `number_guessing_game()`: This is the main function. It initializes the game, sets the secret number, runs the `for` loop for the maximum attempts, and calls `get_valid_guess` in each iteration.

9 Screenshots / Results

The following is a conceptual representation of the console output.







10 Testing Approach

A manual black-box testing approach was used.

- **Normal Flow Test:** Guessing correctly within 7 attempts.
- **Boundary Testing:** Checking inputs like 1, 100, 0, and 101.
- **Invalid Input Testing:** Entering text, floating-point numbers, and duplicate guesses to ensure `get_valid_guess` handles them gracefully.
- **Win/Loss Condition Test:** Ensuring the game correctly terminates when the guess limit is reached or the number is found.

11 Challenges Faced

- **Input Robustness:** Ensuring the code correctly handles unexpected input types (e.g., `ValueError` when converting non-integer text to `int`). This was solved by using a `try-except` block inside the validation function.
- **Tracking Guesses:** Making sure the user doesn't repeat a guess, which was solved by storing all previous guesses in a list and checking for membership.
- **Analytical Logic:** Correctly implementing the set logic in `analyze_guesses` to mathematically narrow down the possible numbers based on all past hints.

12 Learnings & Key Takeaways

- Mastery of the fundamental control flow structures: `while` loops, `for` loops, and `if-elif-else` blocks.
- The importance of **Input Validation** for creating robust software.
- Practical application of the **Binary Search** algorithm concept, reflected in the game's design and the `min_additional_tries` function.
- Using data structures (lists and sets) to track state (previous guesses and possible numbers).

13 Future Enhancements

1. **Difficulty Levels:** Allowing the user to choose the range (e.g., 1-50, 1-1000) which would adjust the number of allowed attempts.
2. **Graphical User Interface (GUI):** Converting the command-line interface to a simple graphical application using a library like Tkinter or PyQt.
3. **Computer Player Mode:** Implementing an AI opponent that uses the binary search strategy to guess the user's number.
4. **Leaderboard:** Integrating a database (like SQLite or Firestore) to store high scores (lowest number of attempts).

14 References

- Python Documentation (Input/Output, Random module).
- Code Logic: Self-developed based on the standard number guessing game structure.