VITYARTHI PROJECT REPORT

Name – DEEPAK JAIN

Rego. No – 25BCE10066

Faculty name – Dr. PAVITHRA R MA'AM

Slot – A11-A12-A13

Problem Statement – Lack Of General knowledge among youth

COURSE CODE – CSE1021

SUBJECT – Problem Solving In Programming

TITLE – Vityarthi Project Report

OBJECTIVE

To create an engaging and interactive game that encourages youth to learn and explore general knowledge in a funnier, interactive and modern way.

Functional Requirements

In my project I used various functions to make it look and run properly and efficiently. I have used various types of

1) Input and Output statements
2) A logical workflow of how the user interacts with the system
3) The system displays GK questions to the user as per their age category.
4) The system validates answers and show correct/incorrect feedback.
5) The system shall maintain and display the user's score.
6) The system shall handle invalid inputs without crashing.

Non Functional Requirements

Not just have used functional requirements, In my program I have also taken care of all the non – functional requirements such as :-

1) The system provide fast response time, displaying questions and results without noticeable delay.
2) The system is made easy to use with clear instructions and simple text-based interaction with the user.
3) The game shall maintain reliability by running without crashes during continuous play.
4) The program shall be portable and run on any system with Python installed.
5) In the whole program, the security of the user's data is kept at priority and is safe to use.

SYSTEM ARCHITECTURE

1) User Interface Layer

- Displays questions, options, instructions, and results.
- Takes user input for answers and menu choices.
- Shows feedback (correct/incorrect) and total score.

2) Game Logic Layer

- Controls game flow (start, play, next question, exit).
- Validates user answers.
- Updates score and tracks progress.
- Manages difficulty levels or rounds (if included).

3) Question Management Module

- Stores GK questions in a list/dictionary/file.
- Randomly selects and delivers questions to the game logic.
- Allows easy addition or modification of questions.

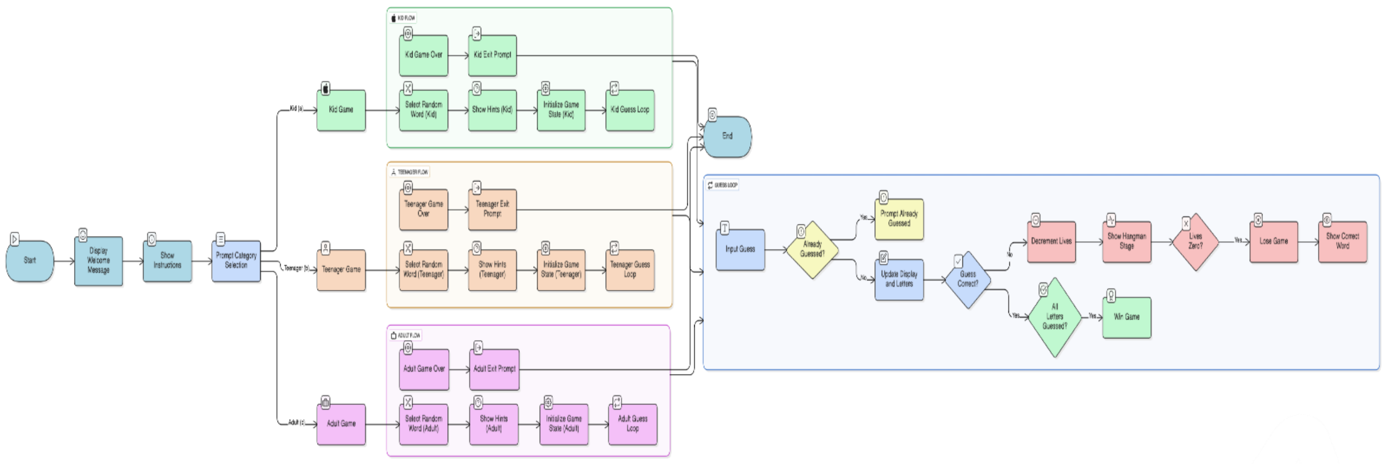
4) Data Handling Layer

- Reads/writes question data (if external file is used).
- Stores user score or high scores (optional).
- Performs input validation.

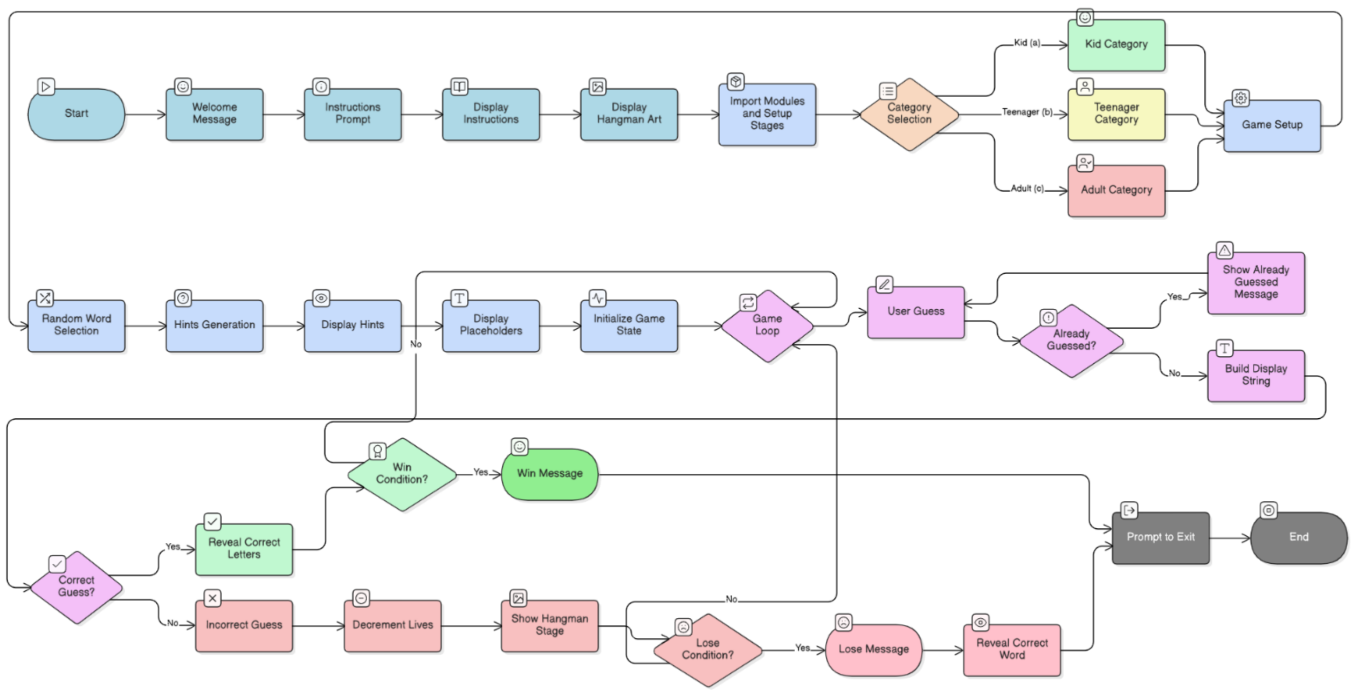
5) System Environment

- Python interpreter.
- Runs in terminal/command line.
- Uses built-in Python modules (random, time, etc.).

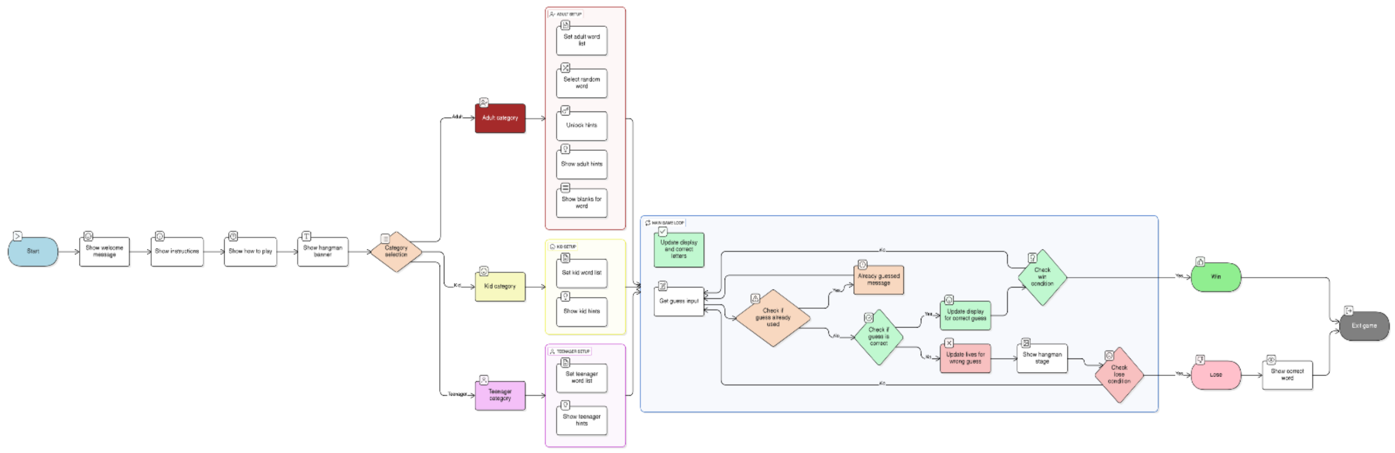
USER FLOW DIAGRAM



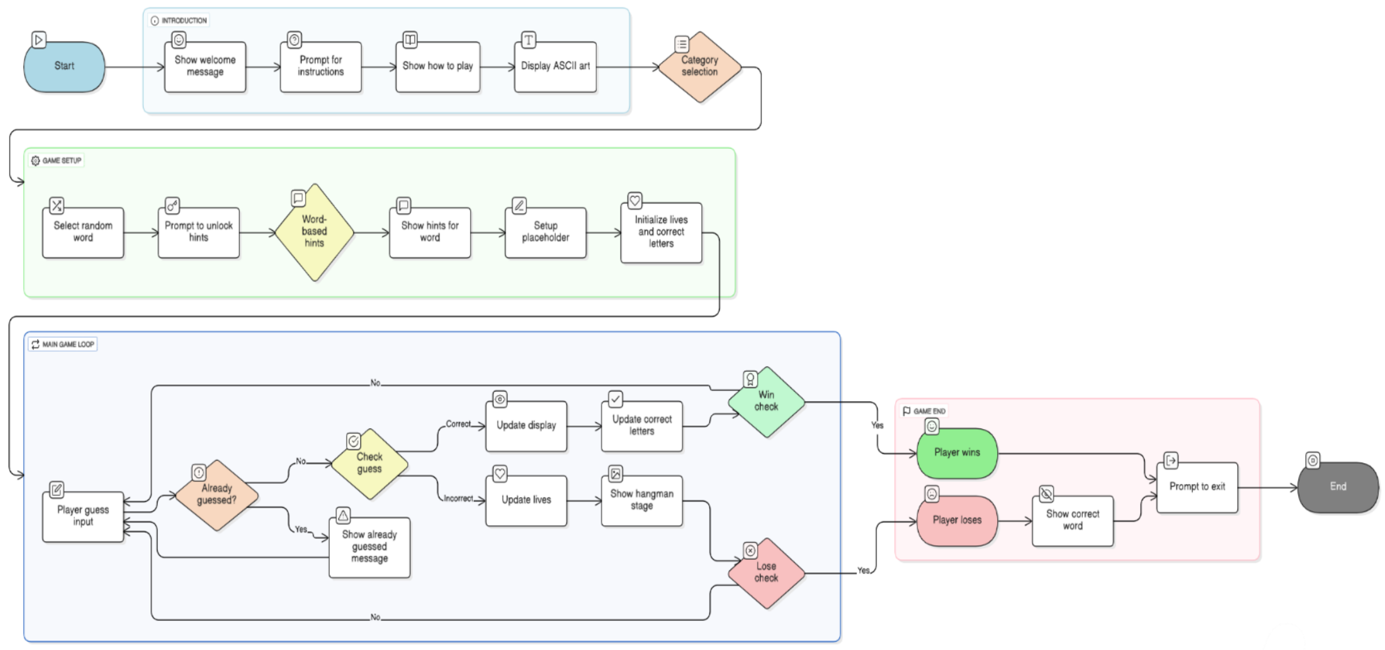
WORKFLOW DIAGRAM



Sequence Diagram



Class/Component Diagram



Implementation Details

The Word challenge arena Game is implemented using Python and follows a structured, category-based approach to enhance user engagement.
The program begins by welcoming the player and displaying instructions, followed by an ASCII-art banner for visual appeal.

---

1. Category Selection
Users choose one of three categories:

- Category A – Kids
- Category B – Teenagers
- Category C – Adults

Based on the user's choice, a word list is loaded and a random word is selected using the random.choice() function.

---

2. Hint System
Before starting the game, players can unlock hints using input().
Each word has 3–4 predefined hints that help the user guess the hidden word.
Hints are displayed through conditional if/elif blocks based on the chosen word.

---

3. Word Placeholder Generation
A placeholder containing underscores is created to represent the hidden word

---

4. Game Loop & Guess Handling

The game runs inside a while loop until:

- the user guesses all letters, or
- user's lives reach zero.

Inside the loop:

- The user enters a guessed letter.
- Previously guessed letters are tracked using a correct_letters list.
- For every guess, the program constructs a display string that shows correctly guessed letters and underscores for the rest.

---

5. Life Management

The user starts with **6 lives**.

If a guessed letter is not in the chosen word:

- A life is deducted.
- The corresponding ASCII stages is displayed.
- If lives reach 0, the game ends and the correct word is shown.

---

6. Win Condition

If the updated display string contains **no underscores**, the user has guessed the full word.

The program prints **"You won!"** and exits the loop.

---

7. ASCII Stages

A list named stages[] contains 7 ASCII art drawings of the Hangman.
The stage is selected dynamically using:
print(stages[6 - lives])
This visually reflects how close the player is to losing.

---

8. Input Validation

- .strip().lower() ensures that the user's guess is processed in a consistent format.
- Repeated letters are handled by checking if they already exist in correct_letters.

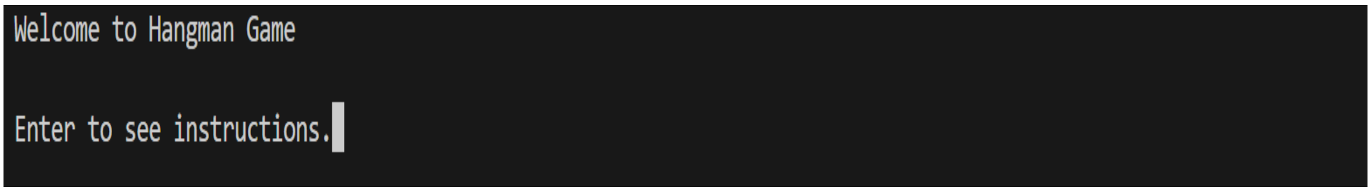
---

9. Program Termination

After winning or losing, the user is prompted:
input("Enter to exit.")
This prevents the program from closing immediately.

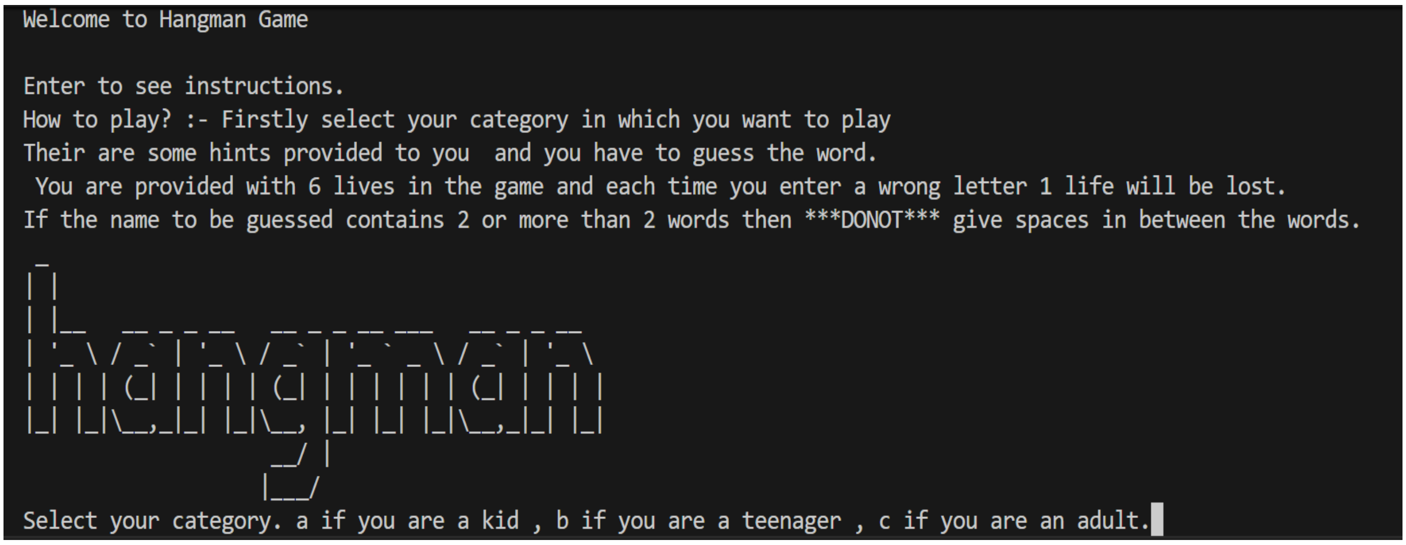
SCREENSHOT OF RESULTS

Program Starts:-



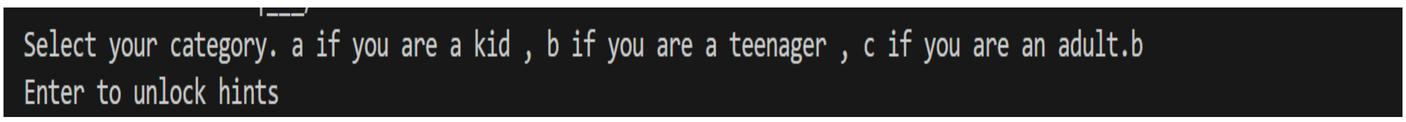
```
Welcome to Hangman Game

Enter to see instructions.
```

All the instructions are shown and category selection window appears :-



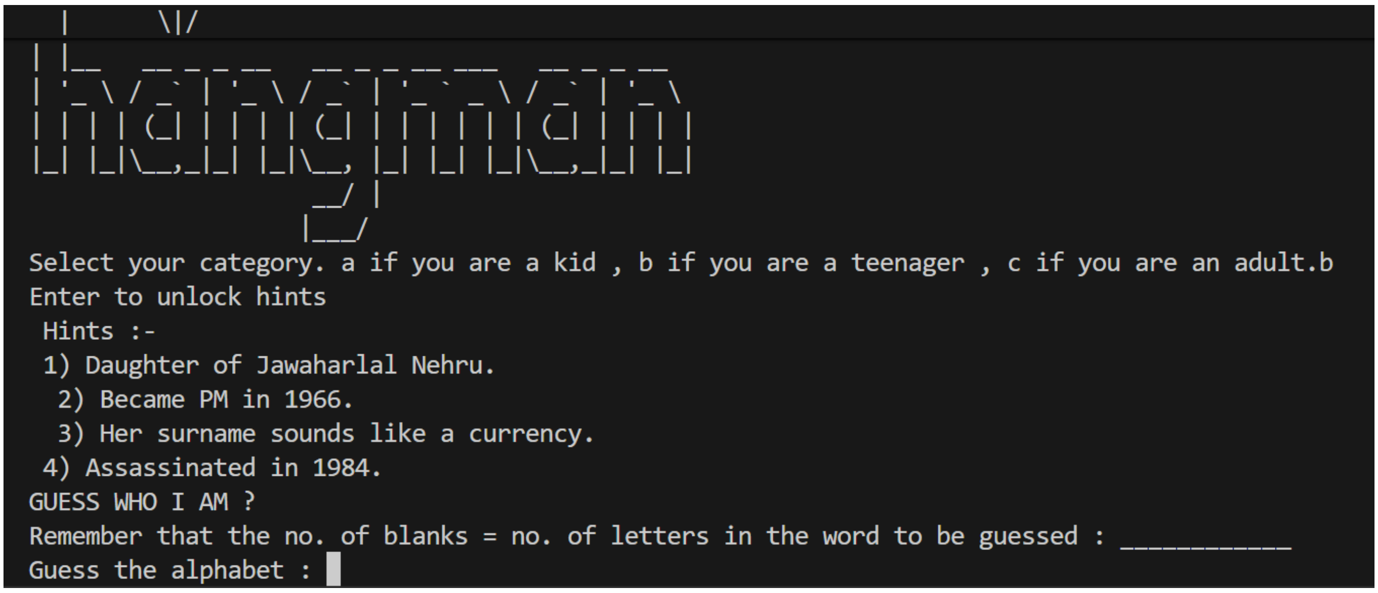
```
Welcome to Hangman Game

Enter to see instructions.
How to play? :- Firstly select your category in which you want to play
Their are some hints provided to you  and you have to guess the word.
 You are provided with 6 lives in the game and each time you enter a wrong letter 1 life will be lost.
If the name to be guessed contains 2 or more than 2 words then ***DONOT*** give spaces in between the words.
 _
| |
| |__   __ _ _ __   __ _ _ __ ___   __ _ _ __
| '_ \ / _` | '_ \ / _` | '_ ` _ \ / _` | '_ \
| | | | (_| | | | | (_| | | | | | | (_| | | | |
|_| |_|\__,_|_| |_|\__, |_| |_| |_|\__,_|_| |_|
                    __/ |
                   |___/
Select your category. a if you are a kid , b if you are a teenager , c if you are an adult.
```

That’s how category is seleted and user is asked to unlock hints if he want to do so:-



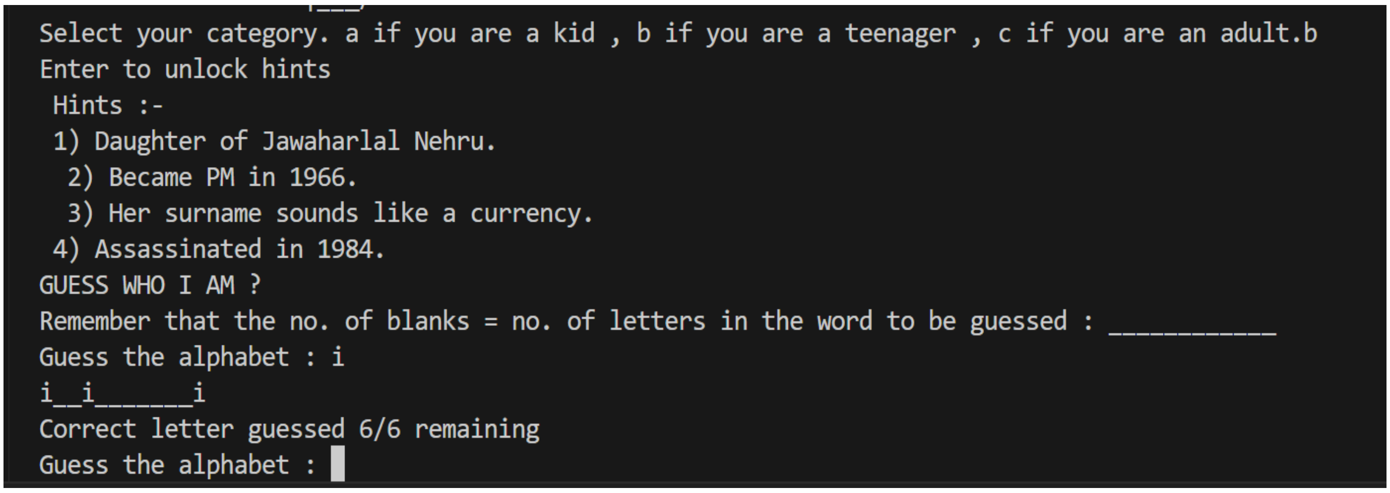
```
Select your category. a if you are a kid , b if you are a teenager , c if you are an adult.b
Enter to unlock hints
```

This is how the question appears and user has to guess the alphabet :-

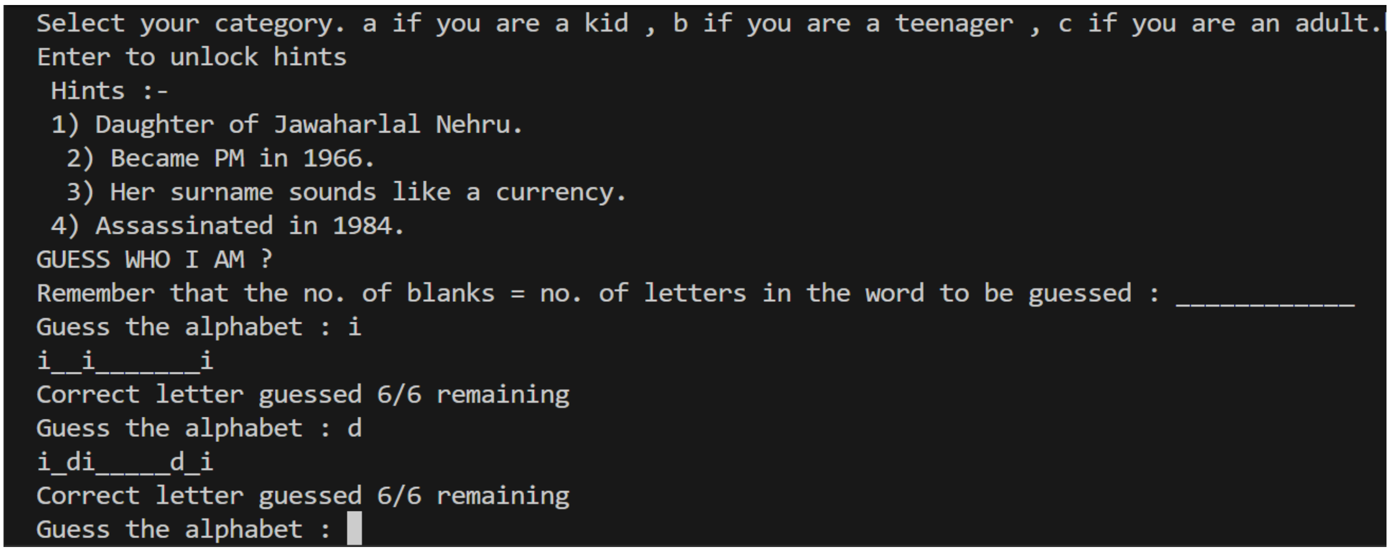


```
   |       \|/
 | |__   __ _ _ __   __ _ _ __ ___   __ _ _ __
 | '_ \ / _` | '_ \ / _` | '_ ` _ \ / _` | '_ \
 | | | | (_| | | | | (_| | | | | | | (_| | | | |
 |_| |_|\__,_|_| |_|\__, |_| |_| |_|\__,_|_| |_|
                     __/ |
                    |___/
Select your category. a if you are a kid , b if you are a teenager , c if you are an adult.b
Enter to unlock hints
 Hints :-
 1) Daughter of Jawaharlal Nehru.
  2) Became PM in 1966.
  3) Her surname sounds like a currency.
 4) Assassinated in 1984.
GUESS WHO I AM ?
Remember that the no. of blanks = no. of letters in the word to be guessed : ____________
Guess the alphabet :
```

If the alphabet guessed is correct :-

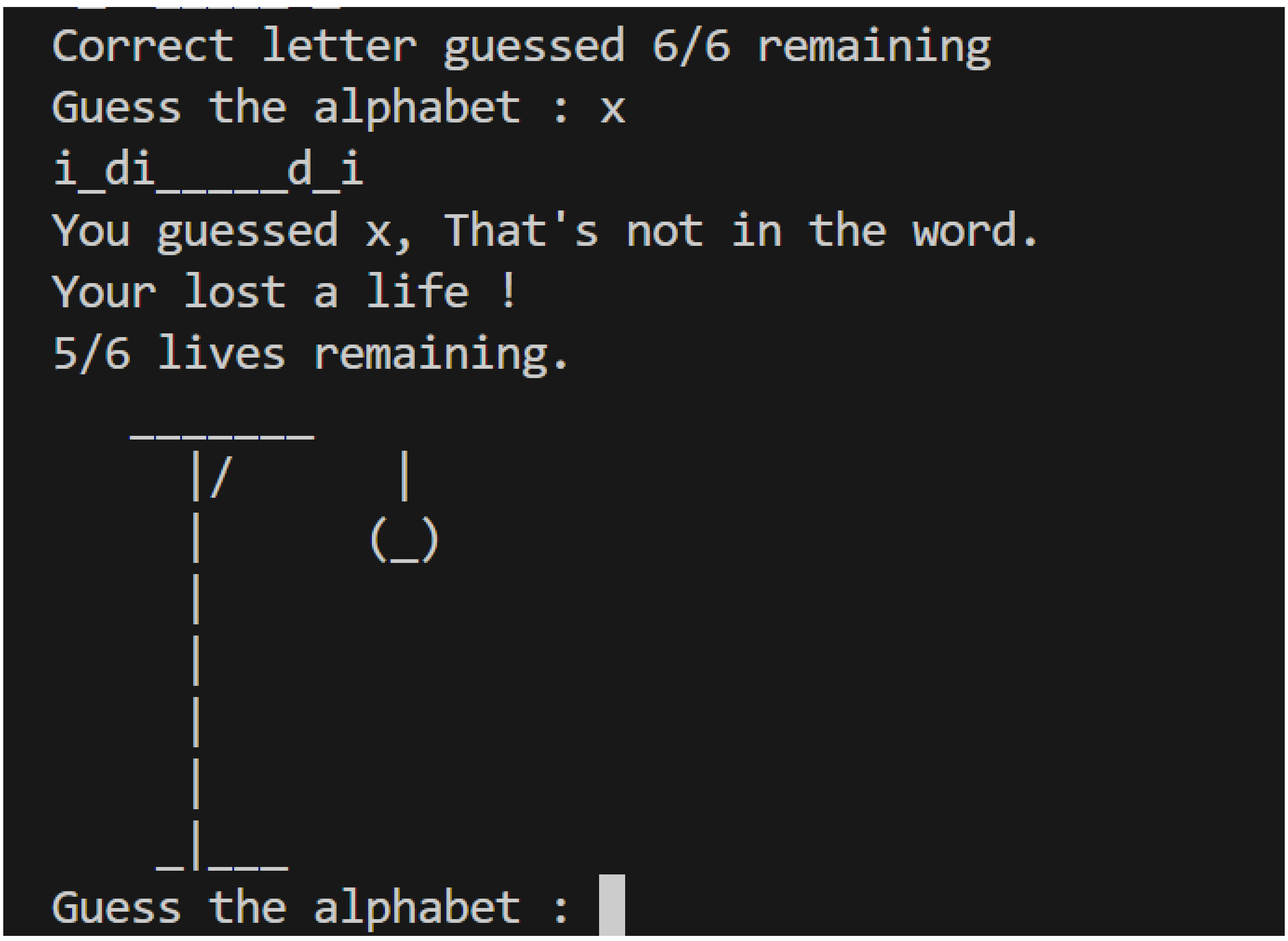


```
Select your category. a if you are a kid , b if you are a teenager , c if you are an adult.b
Enter to unlock hints
 Hints :-
 1) Daughter of Jawaharlal Nehru.
  2) Became PM in 1966.
  3) Her surname sounds like a currency.
 4) Assassinated in 1984.
GUESS WHO I AM ?
Remember that the no. of blanks = no. of letters in the word to be guessed : ____________
Guess the alphabet : i
i__i_______i
Correct letter guessed 6/6 remaining
Guess the alphabet :
```



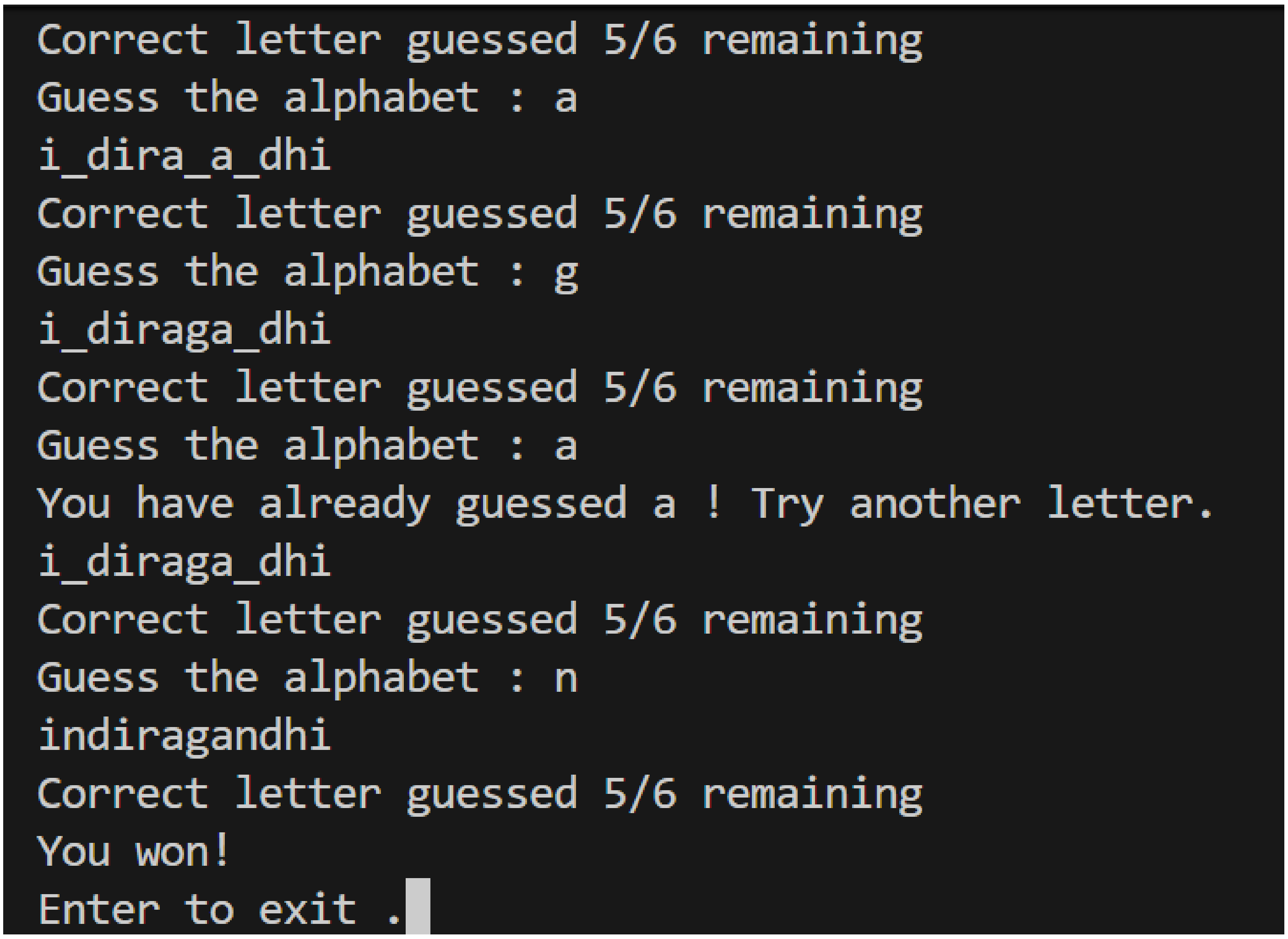
```
Select your category. a if you are a kid , b if you are a teenager , c if you are an adult.
Enter to unlock hints
 Hints :-
 1) Daughter of Jawaharlal Nehru.
  2) Became PM in 1966.
  3) Her surname sounds like a currency.
 4) Assassinated in 1984.
GUESS WHO I AM ?
Remember that the no. of blanks = no. of letters in the word to be guessed : ____________
Guess the alphabet : i
i__i_______i
Correct letter guessed 6/6 remaining
Guess the alphabet : d
i_di_____d_i
Correct letter guessed 6/6 remaining
Guess the alphabet :
```

If user guesses a wrong alphabet :-



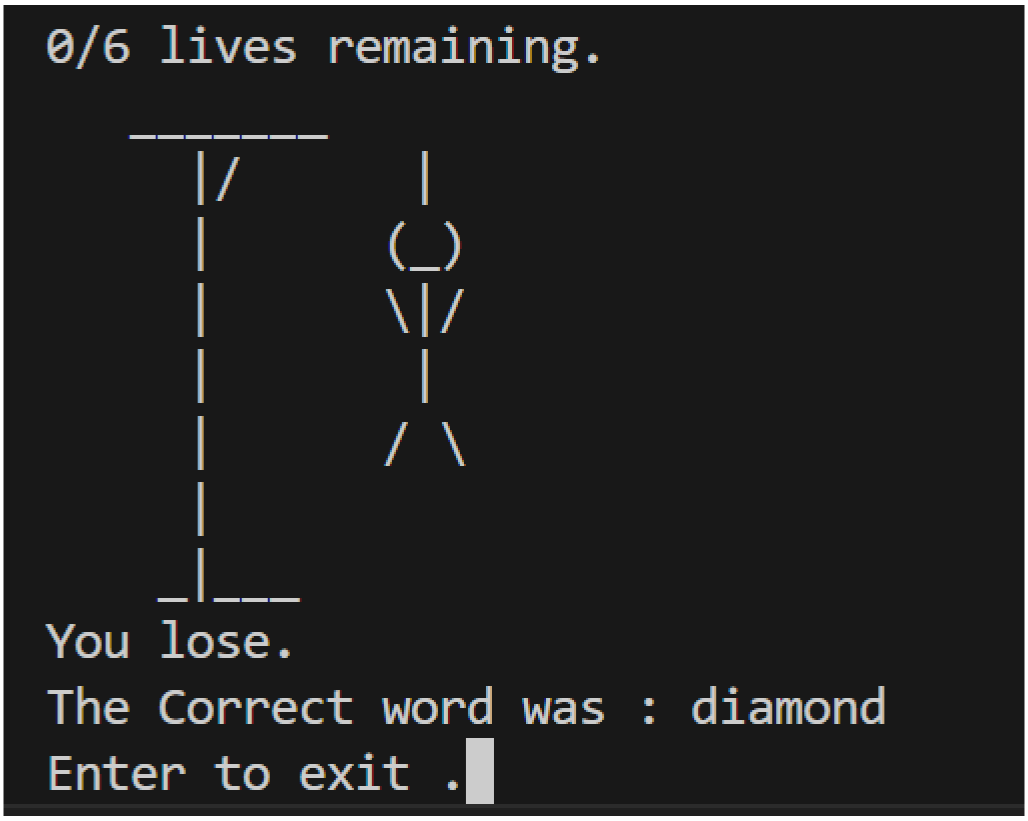
```
Correct letter guessed 6/6 remaining
Guess the alphabet : x
i_di_____d_i
You guessed x, That's not in the word.
Your lost a life !
5/6 lives remaining.

   _______
     |/      |
     |      (_)
     |
     |
     |
     |
    _|___
Guess the alphabet : 
```

if at the end user guesses the correct word :-



```
Correct letter guessed 5/6 remaining
Guess the alphabet : a
i_dira_a_dhi
Correct letter guessed 5/6 remaining
Guess the alphabet : g
i_diraga_dhi
Correct letter guessed 5/6 remaining
Guess the alphabet : a
You have already guessed a ! Try another letter.
i_diraga_dhi
Correct letter guessed 5/6 remaining
Guess the alphabet : n
indiragandhi
Correct letter guessed 5/6 remaining
You won!
Enter to exit .
```

If the user is unable to guess the word :-



```
0/6 lives remaining.

   _______
     |/      |
     |      (_)
     |      \|/
     |       |
     |      / \
     |
    _|___
You lose.
The Correct word was : diamond
Enter to exit .
```

TESTING APPROACH

The “Word Challenge arena” game will be tested using a systematic and incremental approach to ensure that all features work correctly across different categories and gameplay scenarios. The testing covers functional behavior, input handling, and edge cases.

---

1. Unit Testing

Individual components of the game are tested separately:

- Category selection (a, b, c)
- Random word selection
- Hint display logic
- Placeholder generation
- Life deduction mechanism
- Letter-matching functionality
- Detection of win/loss conditions

Each function or code block is verified whether it performs its expected role.

---

2. Functional Testing

All game functions are tested end-to-end:

- Game starts and instructions display correctly.
- Hints appear based on the selected word.
- Correct letters update the placeholder.
- Incorrect guesses reduce lives and show correct ASCII stage.
- Victory occurs when all letters are guessed.
- Defeat occurs when lives reach zero.
- Exit prompt appears properly.

3. Input Validation Testing

The system is tested with various kinds of user input:

- Lowercase, uppercase letters
- Repeated letters
- Numeric or special characters
- Blank inputs
- Multi-character inputs

The game should handle invalid inputs gracefully without crashing.

4. Category-Based Scenario Testing

Each category (Kids, Teens, Adults) is tested separately:

- Confirm the word list loads properly
- Verify unique hints for each word
- Ensure long words (like "objectorientedprogramming") display correctly
- Check if multi-word names without spaces work as expected

5. Randomness Testing

Multiple runs are performed to ensure:

- Different words are selected randomly
- Hints correctly match the chosen word
- Game behaves consistently regardless of which word appears

6. Edge Case Testing

Special scenarios are tested:

- Guessing the same letter repeatedly
- Guessing all incorrect letters
- Guessing the correct word on the final life
- Breaking the game loop intentionally
- Words with repeated letters (like "banana") display correctly

---

7. User Experience Testing

The game interface is checked for:

- Readability of messages
- Correct spacing and formatting
- Proper display of ASCII graphics
- Smooth flow of the game

---

8. Performance & Stability Testing

Even though this is a console game, it is tested for:

- Fast response time for all inputs
- No crashes during extended play
- Smooth functioning across Python versions and OS environments

CHALLENGES FACED

1) **Managing Multiple Categories:**
   Designing separate word lists, hints, and game logic for kids, teenagers, and adults required careful structuring to avoid repetition and ensure accuracy for each category.

2) **Handling Long and Complex Words:**
   Words like *“objectorientedprogramming”* and*“sardarvallabhbhaipatel”* created challenges in placeholder generation, display formatting, and readability during gameplay.

3) **Ensuring Correct Letter Matching:**
   Maintaining a list of correct letters and updating the displayed word dynamically without errors required careful conditional checks within loops.

4) **Avoiding Repeated Input Issues:**
   Preventing users from guessing the same letter repeatedly and handling such cases gracefully was a challenge, requiring extra validation.

5) **Life Deduction and Hangman Stage Accuracy:**
   Synchronizing lives with the correct ASCII hangman stage demanded precise indexing. Mistakes could easily lead to misaligned visuals.

6) **Ensuring Hint Accuracy and Mapping:**
Each chosen word needed correctly matched hints. Managing many if/elif cases without mixing hints required extra attention.

7) **Input Validation Problems:**
Users entering uppercase letters, special characters, spaces, or multiple characters at once caused unexpected behavior and required sanitization using .strip().lower().

8) **Loop Control & Game Flow:**
Implementing a smooth game loop where the game ends exactly when needed (win or lose) required careful handling of game_over conditions.

9) **Avoiding Code Repetition:**
Large repeated blocks for each category (Kids, Teenagers, Adults) made the code harder to maintain and update, presenting structural and readability challenges.

10) **Maintaining User Experience:**
Printing hints, updating placeholders, and showing ASCII stages while keeping the output neat and readable required thoughtful formatting and spacing.

Learnings And Take Aways

❖ **Key Learnings**

- User Input and Interaction
  The game uses multiple input() prompts to interact with the player, including showing instructions, choosing categories, unlocking hints, and receiving guesses. This reinforces the use of Python's input/output functions for interactive command-line applications.

- Control Flow & Loops
  This informative game progression relies on while loops to repeatedly prompt for guesses until the player wins or loses. Conditional statements (if, elif, else) manage game branching, hint display, loss of lives, and win/loss notifications. This improves logical thinking and flow control management in Python.

- Randomization and Word Selection
  The use of Python's random.choice() selects a word from a predefined list, which teaches how to incorporate randomness in games and applications, ensuring varied gameplay experiences.

- Data Structures
  Lists are used for the word bank per category, a list of Hangman stages in ASCII art, and for tracking correctly guessed letters. String manipulation and list traversal are demonstrated when revealing correct guesses and updating the display.

- Game State and Logic
  The program tracks game state through variables like lives, a placeholder string for the word display, and a list of already-guessed letters. Life decrement and stage update logic are tied to incorrect guesses, which enforces state management in game programming.

- Modularization & Readability
  Code segmentation by category with dedicated word banks and hints offers an introduction to modular design. Consistent formatting and messaging (win/lose, hints, display updates) enhance readability and maintainability.

❖Key Takeaways

- Handling user input and validating it is critical to maintaining game integrity.
- Random selection and updates to display create a dynamic experience.
- Life-based gameplay mechanics motivate the player and emphasize the impact of correct/incorrect actions.
- Category-based structuring can help tailor the difficulty and content for different audiences, broadening engagement.
- ASCII art can be used to make terminal games visually engaging.
- Proper use of Python's data structures aids in tracking progress, state, and user choices efficiently.
- Implementing hints and guidance supports educational value and helps newcomers participate enjoyably.

These aspects make the "Word challenge arena" game an excellent foundational project for mastering interactive console applications, game logic, state management, and user-centric design in Python.

Future Enhancements

Some meaningful future enhancements for this “Word challenge arena “game are:

1. Code Structure & Reusability

- Extract repeated logic (game loop, hint display, word selection) into separate functions like play_game(category), show_hints(word), and update_display().
- Store categories, words, and hints in dictionaries or external JSON/CSV files so new words and categories can be added without changing core logic.

2. Input Validation & UX Improvements

- Validate category and guess inputs (ensure single alphabetic character, handle invalid options, allow replay without restarting the script).
- Add clear messages for invalid entries, show remaining unguessed letters, and optionally show already-guessed wrong letters separately.

3. Difficulty Levels & Scoring

- Add difficulty modes (easy/medium/hard) that change number of lives, word length, or hint availability.
- Implement a scoring system based on remaining lives, number of wrong attempts, and time taken, and maintain a high-score table.

4. Better Word and Hint Management

- Expand word lists for each category, and make hints more structured.
- Let users choose a subcategory (e.g., fruits, countries, tech) before starting.

5. UI / UX Enhancements

- Improve ASCII art with clearer stages, and show lives visually (like hearts or bars).
- Add a "play again" menu, and maybe a small intro menu (1: Play, 2: Instructions, 3: Exit).

6. Persistence & Analytics

- Log results to a file (wins, losses, word guessed, number of attempts) to analyze difficulty and player performance.
- Use this data later to adjust difficulty automatically (adaptive difficulty based on player success rate).

7. Advanced Extensions

- Build a GUI version using tkinter or a web version using Flask/React to make it more user-friendly and visually rich.
- Add multiplayer or versus-computer mode where one player enters the word and another guesses, or words are fetched from an online dictionary/API.

REFERENCES

1) Vityarthi course modules
2) Python documentation
3) https://ascii.co.uk/art used for inserting figures
4) Help from slot faculty
5) www.google.com
6) Object-Oriented Python book by IRV KALB
7) Module ppt’s shared in class