VITyarthi- Build Your Own Project

HANGMAN

VIT BHOPAL UNIVERSITY

INTRODUCTION

HANGMAN IS A GUESSING GAME FOR TWO OR MORE PLAYERS. ONE PLAYER THINKS OF A WORD, PHRASE, OR SENTENCE AND THE OTHER(S) TRIES TO GUESS IT BY SUGGESTING LETTERS OR NUMBERS WITHIN A CERTAIN NUMBER OF GUESSES. ORIGINALLY A PAPER AND PENCIL GAME, THERE ARE NOW ELECTRONIC VERSIONS.

THE WORD TO GUESS IS REPRESENTED BY A ROW OF DASHES REPRESENTING EACH LETTER OR NUMBER OF THE WORD. RULES MAY PERMIT OR FORBID PROPER NOUNS (SUCH AS NAMES, PLACES, OR BRANDS) OR OTHER TYPES OF WORDS (SUCH AS SLANG). IF THE GUESSING PLAYER SUGGESTS A LETTER WHICH OCCURS IN THE WORD, THE OTHER PLAYER WRITES IT IN ALL ITS CORRECT POSITIONS. IF THE SUGGESTED LETTER DOES NOT OCCUR IN THE WORD, THE OTHER PLAYER ADDS (OR ALTERNATIVELY, REMOVES) ONE ELEMENT OF A HANGED STICK FIGURE AS A TALLY MARK. GENERALLY, THE GAME ENDS ONCE THE WORD IS GUESSED, OR IF THE STICK FIGURE IS COMPLETE—SIGNIFYING THAT ALL GUESSES HAVE BEEN USED.

THE PLAYER GUESSING THE WORD MAY, AT ANY TIME, ATTEMPT TO GUESS THE WHOLE WORD. IF THE WORD IS CORRECT, THE GAME IS OVER AND THE GUESSER WINS. OTHERWISE, THE OTHER PLAYER MAY CHOOSE TO PENALIZE THE GUESSER BY ADDING AN ELEMENT TO THE DIAGRAM. IF THE GUESSER MAKES ENOUGH INCORRECT GUESSES TO ALLOW THE OTHER PLAYER TO COMPLETE THE DIAGRAM, THE GUESSER LOSES. HOWEVER, THE GUESSER CAN ALSO WIN BY GUESSING ALL THE LETTERS THAT APPEAR IN THE WORD, THEREBY COMPLETING THE WORD, BEFORE THE DIAGRAM IS COMPLETED.

PROBLEM STATEMENT

CREATE A CONSOLE-BASED GAME OF HANGMAN WHERE A PLAYER GUESSES LETTERS TO FIND A HIDDEN WORD. THE PLAYER HAS A LIMITED NUMBER OF INCORRECT GUESSES, AND THE PROGRAM SHOULD TRACK THEIR PROGRESS, REVEAL CORRECT LETTERS, AND DETERMINE IF THE PLAYER WINS OR LOSES.

FUNTIONAL REQUIREMENTS

Functional requirements that this project provides are:-

- Data input and processing
- CRUD operations
- Simulation or visualization

The logic of the Hangman program is to take a secret word and guess it letter by letter. The player has a limited number of wrong guesses before they lose the game. If the player guesses all of the letters in the word correctly before running out of guesses, they win the game.

The program works by keeping track of the following information:

- The secret word
- The letters that have been guessed
- The number of wrong guesses

In each iteration of the game loop, the program displays the current state of the guessed word and asks the player to guess a letter. The program then checks if the guessed letter has already been guessed. If it has, the program prompts the player to guess again. If the guessed letter has not already been guessed, the program checks if it is in the secret word. If it is, the program updates the guessed word to include the letter. If the guessed letter is not in the secret word, the program increments the number of wrong guesses.

The game loop continues until the player either guesses the secret word correctly or runs out of guesses. If the player guesses the secret word correctly, the program displays a message congratulating them and the game ends. If the player runs out of guesses, the program displays a message revealing the secret word and the game ends.

NON FUNCTIONAL REQUIREMENTS

Non funtional requirements offered by this project are as follows:

- Error handling strategy
- Usability
- Performance
- Maintainability

SYSTEM ARCHITECTURE

The system architecture of the Hangman game follows a modular procedural design with clear logical components. Although the implementation uses functions rather than classes, the architecture can be conceptually separated into layers and modules for documentation purposes.

1. Presentation Layer (User Interface Layer)

Responsibilities:

- Interacts directly with the player.
- Displays game state (letters, hangman ASCII art, lives).
- Collects player input.

Key elements in code:

- input() prompts
- Printing display
- Printing ASCII stages
- Win/Lose messages

2. Application Logic Layer (Game Engine Layer)

Responsibilities:

- Runs the core Hangman gameplay loop.
- Validates user guesses.
- Tracks game state (lives, guessed letters, filled positions).
- Determines win/lose conditions.

Key function:

- hangman()

Internal logic handled:

- Initialize word and display
- Manage game loop
- Update display state
- Reduce lives on incorrect guesses
- End-game condition checks

3. Data Layer

Responsibilities:

- Provides the source of playable words.
- Loads words from a file.

Key function:

- get_word()

Data handled:

- words.txt file
- chosen_word variable

4. Resource Layer

Responsibilities:

- Stores pre-defined ASCII art representing hangman states.

Key elements:

- stages list

DESIGN DECISIONS AND RATIONALE

1. Use of ASCII Art (stages list)

Decision

A list of multi-line strings is used to visually represent the hangman at different stages of failure.

Rationale

- Makes the game more engaging and visually clear.
- Keeps graphics self-contained within the program without external files.
- Indexing (stages[6 - lives]) directly maps game progression to visuals.

2. Separation of Concerns Using a get_word() Function

Decision

The logic for retrieving a random word is encapsulated in a dedicated function.

Rationale

- Improves readability and modularity.
- Allows reusability—word selection logic stays isolated.
- Makes it easy to replace with other sources (APIs, databases) later.
- Handles file errors gracefully using a try-except block.

3. Main Game Logic Encapsulated in the hangman() Function

Decision

All game logic runs within a single function, except for word retrieval.

Rationale

- Keeps the code organized and easier to maintain.
- Avoids global variable clutter.
- Simplifies troubleshooting because all game state exists locally within one function.

4. Use of display List to Track Visible Progress

Decision

display = ["_"] * len(chosen_word) stores blanks or correctly guessed letters.

Rationale

- Allows easy updating of only the correct positions.
- Simple to convert to a readable string using ' '.join(display).
- Prevents modifying the original chosen word.

5. Use of a set for guessed_letters

Decision

A set tracks which letters the user already guessed.

Rationale

- Prevents duplicate guesses.
- Constant-time lookup (fast and efficient).
- Cleaner logic for checking repeated input.

6. Controlled Game Loop Using end_of_loop Flag

Decision

A boolean flag determines when the round ends.

Rationale

- Improves clarity compared to multiple break statements.
- Makes the game flow easier to track and extend.

7. Lives System Linked to ASCII Graphics

Decision

lives begins at 6 and decrements for wrong guesses.

Rationale

- Simple, intuitive failure system.
- Direct mapping to ASCII art stages enhances feedback.
- Easy to scale—developer can add or remove stages.

8. Input Normalization

Decision

guess = input(...).lower() converts player input to lowercase.

Rationale

- Ensures case-insensitive gameplay.
- Avoids errors where uppercase guesses fail to match the lowercase word.

9. Win and Lose Conditions Evaluated Every Turn

Decision

The game checks for win ("_" not in display) and lose (lives == 0) at the end of each iteration.

Rationale

- Prevents unnecessary additional loops.
- Provides immediate feedback to the player.
- Easy-to-read, clear game-end logic.

10. Main Program Loop Allowing Replay

Decision

The game runs inside a while True: loop asking for “y/n” input.

Rationale

- Simple and familiar way to allow multiple game sessions.
- Does not require restarting the script to replay.
- Improves user experience.

11. Robust Input Handling

Decision

The program validates:

- repeated guesses
- invalid replay responses
- missing words file

Rationale

- Enhances reliability and user friendliness.
- Prevents unexpected crashes.
- Makes the game safer for beginner users.

RESULTS

```
PS C:\Users\Deepanjali\OneDrive\Desktop\pythoncodes\hangman> python -u "c:\Users\Deep
anjali\OneDrive\Desktop\pythoncodes\hangman\hangman.py"
Do you want to play Hangman? (y/n): y

-------------Welcome to Hangman-------------

Guess the word:-  _ _ _ _ _ _ _ _ _
Lives: 6
Guess a Letter: a
_ _ _ _ _ _ a _ _
Lives: 6

  +---+
  |   |
  O   |
 /|\  |
 / \  |
      |
=========

Guess a Letter: e
e _ _ _ e _ a _ e
Lives: 6

  +---+
  |   |
  O   |
 /|\  |
 / \  |
      |
=========

Guess a Letter: i
e _ _ _ e _ a _ e
Lives: 5

  +---+
  |   |
  O   |
 /|\  |
 /    |
      |
=========
```

```
Guess a Letter: o
e _ _ _ e _ a _ e
Lives: 4

  +---+
  |   |
  O   |
 /|\  |
      |
      |
=========

Guess a Letter: u
e _ u _ e _ a _ e
Lives: 4

  +---+
  |   |
  O   |
 /|\  |
      |
      |
=========

Guess a Letter: t
e _ u _ e _ a t e
Lives: 4

  +---+
  |   |
  O   |
 /|\  |
      |
      |
=========

Guess a Letter: d
e _ u _ e _ a t e
Lives: 3

  +---+
  |   |
  O   |
 /|   |
```

```
      |
      |
=========

Guess a Letter: c
e _ u _ e _ a t e
Lives: 2

  +---+
  |   |
  O   |
  |   |
      |
      |
=========

Guess a Letter: n
e n u _ e _ a t e
Lives: 2

  +---+
  |   |
  O   |
  |   |
      |
      |
=========

Guess a Letter: n
You already guessed 'n'. Try a different letter.
Guess a Letter: g
e n u _ e _ a t e
Lives: 1

  +---+
  |   |
  O   |
      |
      |
      |
=========

Guess a Letter: j
e n u _ e _ a t e
```

```
Guess a Letter: j
e n u _ e _ a t e
Lives: 0

  +---+
  |   |
      |
      |
      |
      |
=========

You Lose
The word was: enumerate
Do you want to play Hangman? (y/n): n
Program Exit Successful
PS C:\Users\Deepanjali\OneDrive\Desktop\pythoncodes\hangman>
```

IMPLEMENTATION DETAILS

The Hangman program follows a sequential, function-based procedural logic. Its behavior can be broken down into several major stages:

1. Initialization of ASCII Art Stages

Before any gameplay, the program defines a list named stages that contains seven ASCII drawings.

Purpose

- Visually represent the hangman's progression as the player loses lives.
- stages[6 - lives] determines which drawing to display.

2. Word Retrieval (get_word() function)

Process

1. The function attempts to open a file named words.txt.
2. Reads all lines and creates a list of possible words.
3. Selects a random word using random.choice().

If file not found

- Print error message.
- Terminate the program using exit().

Purpose

- Centralizes word selection logic.
- Ensures each game begins with a random word.

3. Start of the Game (hangman() function)

When the player chooses to play, the program calls hangman().

Initialization inside hangman():

- chosen_word ← randomly selected word
- display ← list of underscores _ representing unguessed letters
- guessed_letters ← empty set
- lives = 6
- end_of_loop = False

Purpose

- Set up the game state before entering the gameplay loop.
- Represent progress through the display list.

4. Gameplay Loop

The program enters a while not end_of_loop: loop.

This loop continues until the player either wins or loses.

5. Player Input Handling

Steps

1. Prompt for a guess: guess = input(...).lower()
2. Check if letter already guessed:
    - If yes → warn the user and continue to next loop iteration.
    - If no → add to guessed_letters.

Purpose

- Enforce case-insensitive guessing.
- Prevent repeated guesses.

6. Guess Evaluation

Case 1: Incorrect guess

- If guess not in chosen_word:
- → lives -= 1

Case 2: Correct guess

- Loop through each index of the word:
-
- for index, char in enumerate(chosen_word):
- if char == guess:
- display[index] = guess

Purpose

- Update the player's visible progress.
- Penalize incorrect guesses.

7. Updating the Screen

After each guess:

- Display updated word:
- ' '.join(display)
- Show remaining lives.
- Show current hangman ASCII stage:
- print(stages[6 - lives])

Purpose

- Provide real-time feedback to the player.

8. Win Condition Check

The player wins if:

if "_" not in display:

Meaning:

All blank spaces have been replaced with correctly guessed letters.

If win

- Print "You Win!"
- end_of_loop = True

9. Lose Condition Check
The player loses if:

```
if lives == 0:
```

If lose

- Print "You Lose"
- Reveal the word
- end_of_loop = True

10. Replay Loop (Main Loop)
Outside the hangman function, the main part of the program runs:

```
while True:
    ask = input("Do you want to play Hangman? (y/n): ").lower()
```

Behavior

- If "yes" → start new game
- If "no" → print exit message & break
- Otherwise → ask again

Purpose

- Allow the player to play multiple rounds.
- Prevent accidental program termination.

TESTING APPROACH

The testing approach ensures that all components of the Hangman game function correctly, handle edge cases, and provide a smooth user experience. The testing strategy combines unit testing, integration testing, functional testing, and user acceptance testing (UAT).

1. Unit Testing

Unit testing focuses on verifying individual functions and logical components.

Components to Test

a. get_word()

Test CaseExpected Outcome

File exists and contains words

Returns a random non-empty string

File is empty

Should not return an empty string (manual handling required)

words.txt missing

Program prints error and exits gracefully

Word list contains special characters

Function still returns valid words

b. ASCII Stages

Test CaseExpected Outcome

Index calculation stages[6 - lives]

Correct stage printed for lives = 6 to 0

Incorrect index (lives < 0 or > 6)

Should never occur — validated by gameplay logic

The testing approach ensures that all components of the Hangman game function correctly, handle edge cases, and provide a smooth user experience. The testing strategy combines unit testing, integration testing, functional testing, and user acceptance testing (UAT).

1. Unit Testing

Unit testing focuses on verifying individual functions and logical components.

Components to Test

a. get_word()

Test CaseExpected Outcome

File exists and contains words

Returns a random non-empty string

File is empty

Should not return an empty string (manual handling required)

words.txt missing

Program prints error and exits gracefully

Word list contains special characters

Function still returns valid words

b. ASCII Stages

Test CaseExpected Outcome

Index calculation stages[6 - lives]

Correct stage printed for lives = 6 to 0

Incorrect index (lives < 0 or > 6)

Should never occur — validated by gameplay logic

4. Boundary Testing
These tests check behavior at the extreme edges of expected input.
Test Cases
Boundary CaseExpected Behavior
Word length = 1
Game still runs correctly
Word length extremely long
Display shows underscores correctly
Lives = 6 (start)
Full hangman not shown
Lives = 0
Game ends immediately
Guessing non-alphabet characters
Should give invalid input message (optional)
5. Error Handling Testing
Test how the program handles unexpected or invalid conditions.
Cases
- Missing words.txt
- Empty lines in words.txt
- Numerical or special character input
- Blank input (press Enter without entering a guess)
- System interrupts (Ctrl+C)

Expected Outcome
- Program avoids crashes
- Prints meaningful error messages
- Continues running when possible

6. Usability Testing (User Acceptance Testing)
To ensure the game feels natural and user-friendly.
Key Factors
- Clarity of instructions
- Legibility of ASCII art
- Feedback messages (win, lose, repeated guess)
- Smoothness of gameplay loop
- Replay loop behavior

Example Scenarios
- Player enters several wrong guesses: program is still understandable
- Player wins quickly: feedback is correct
- Player chooses to play again: program resets cleanly

7. Regression Testing

Whenever improvements or new features are added:

- Re-test all core functions
- Re-test gameplay start and end
- Re-test word selection
- Re-test ASCII stage transitions

Purpose

To ensure new changes don’t break old features.

8. Manual Testing + Automation Potential

Because this is a console-based interactive game:

- Most tests are manual, especially input-based ones.
- Automated tests can be added for:
    - get_word()
    - ASCII art validation
    - Game logic with mocked inputs

CHALLENGES FACED

Developing the Hangman game involved several design, implementation, and usability challenges. Some were technical, while others were related to ensuring smooth gameplay and user experience.

1. Handling File Input for Word Selection

The game relies on reading a word list from an external file (words.txt).

Challenges included:

- Ensuring the file exists in the correct directory during execution.
- Handling empty or incorrectly formatted files.
- Managing exceptions such as FileNotFoundError.
- Ensuring random and fair word selection every time the game starts.

This required careful implementation of error handling and testing.

2. Implementing Input Validation for User Guesses

Since the game is interactive, users may enter:

- Numbers
- Special characters
- Multiple characters at once
- Blank input
- Previously guessed letters

Each of these needed to be checked to avoid crashes or invalid game states. Handling this gracefully—while still keeping the gameplay fluid—was a key challenge.

3. Maintaining Game State Across Iterations

The game state involved variables such as:

- Remaining lives
- Correctly guessed letters
- Incorrect guesses
- Display list (underscores + revealed letters)

Keeping these updated accurately and consistently after each turn required careful logic.

Mistakes could lead to:

- Incorrect win/lose conditions
- Repeated letters not showing properly
- Hangman stages not matching the number of lives

4. Designing the ASCII Art Hangman Stages

Creating clear ASCII art for each stage was challenging because:

- The layout must stay aligned for different screen widths.
- Extra spaces or misaligned characters break the visual structure.
- Selecting the correct stage (stages[6 - lives]) had to be carefully indexed to avoid off-by-one errors.

Ensuring a clean, professional appearance took multiple iterations.

5. Handling the Win/Lose Logic

The game ends when:

- The player guesses all letters (win)
- The player loses all lives (lose)

Challenges included:

- Ensuring the win condition triggers at the exact moment the final letter is revealed.
- Avoiding repeated evaluation of the losing condition.
- Showing the correct final messages before exiting or restarting.

6. Managing the Replay Loop

Allowing the user to play multiple rounds required:

- Resetting all variables properly
- Selecting a new random word
- Resetting lives and guessed letters

Resetting the game state incorrectly often resulted in bugs like:

- Old words appearing again
- Lives not resetting
- Guessed letters carrying over from the previous game

This required a clean modular approach.

7. Ensuring a Smooth User Experience

Text-based games rely heavily on clarity.

Challenges included:

- Showing the display board in a readable format
- Making prompts simple and understandable
- Giving meaningful messages for repeated or invalid guesses
- Keeping the ASCII art uncluttered

Striking a balance between clarity and minimalism took multiple refinements.

8. Error Handling and Game Stability

Unexpected issues needed to be handled without crashing the game:

- Keyboard interrupts (Ctrl + C)
- Corrupted input
- System-specific encoding issues while reading the file

Proper exception handling helped make the game robust.

LEARNINGS AND KEY TAKEAWAYS

Developing the Hangman game provided several technical, logical, and design insights. These learnings helped improve understanding of Python programming, user interaction, program structure, and error handling.

1. Importance of Modular Programming

Creating functions such as get_word() and hangman() reinforced the value of modularity:

- Functions make code cleaner, reusable, and easier to maintain.
- Separation of concerns reduces bugs and improves organization.
- Debugging becomes much easier when logic is isolated.

2. File Handling and Exception Management

Working with an external file (words.txt) highlighted how crucial file I/O skills are:

- Awareness of directory structure and relative paths.
- Understanding how to detect missing files.
- Using try/except blocks to prevent program crashes.
- Ensuring graceful error messages for the user.

This experience strengthened practical exception handling skills.

3. Input Validation and User-Friendly Interaction

Building a text-based game showed how unpredictable user input can be:

- Users may enter invalid characters, repeat guesses, or make mistakes.
- Validation is necessary to prevent logical errors or infinite loops.
- Giving meaningful feedback improves user experience drastically.

This taught the importance of anticipating user errors in interactive programs.

4. Managing Game State and Logic Flow

The game required tracking:

- Lives
- Correct letters
- Incorrect guesses
- Display progress
- Hangman stages

This helped deepen understanding of:

- Lists, sets, and iteration
- Conditional branching
- Real-time updating of variables

It reinforced how to structure and manage stateful programs.

5. Working with ASCII Art and Index Mapping

Displaying the hangman stages taught:

- How to structure ASCII art within Python strings.
- How to manage indexing (stages[6 - lives]) carefully.
- How visual elements can be tied to program logic.

This improved awareness of how UI (even text-based) connects to backend logic.

6. Loop Control and Game Termination Conditions

The project strengthened knowledge of:

- While loops
- Break conditions
- Nested logic for win/lose detection

Understanding the flow of the game loop improved ability to design controlled iterative systems.

7. Importance of Clean, Readable Output

Since the game is fully text-based, clarity was essential:

- Spacing, indentation, and formatting matter.
- Readable output greatly enhances user engagement.
- Feedback messages must be informative but not overwhelming.

The project highlighted the value of good textual UI design.

8. Debugging and Iterative Refinement

Multiple issues likely arose during development, such as:

- Incorrect stage display
- Logic errors in letter replacement
- Repeated input handling
- Off-by-one indexing problems

Debugging these improved problem-solving skills and reinforced the importance of:

- Testing code incrementally
- Reading traceback messages carefully
- Refactoring when needed

9. Understanding Real-World Programming Concepts in a Simple Project

Even though Hangman is simple, it teaches many fundamental software engineering concepts:

- Edge case management
- Error handling
- Stateful logic
- User input processing
- Modularity and reusability
- UI/UX thinking

This reinforces how even basic games provide deep learning opportunities.

REFERENCES

- www.geeksforgeeks.org
- python-forum.io
- inventwithpython.com
- www.pythonforbeginners.com
- www.w3schools.com