

Basic Health Symptom Checker

A Python-Based Conditional Logic System

Project Report

Submitted by:

Jyotirmay Mahendra Adwant
25BAI11619

Contents

1	Introduction	3
2	Problem Statement	3
3	Functional Requirements	3
4	Non-Functional Requirements	3
5	System Architecture	4
6	Design Diagrams	4
6.1	Use Case Diagram	4
6.2	Workflow Diagram (Flowchart)	5
6.3	Sequence Diagram	5
6.4	Component Diagram	6
6.5	ER Diagram	6
7	Design Decisions & Rationale	6
8	Implementation Details	6
9	Screenshots & Results	7
9.1	Scenario 1: Emergency Path	7
9.2	Scenario 2: General Assessment	7
10	Testing Approach	7
11	Challenges Faced	7
12	Learnings & Key Takeaways	8
13	Future Enhancements	8
14	References	8

1 Introduction

The **Automated Health Symptom Checker** is a software utility designed to simulate a basic medical triage process. Developed using the Python programming language, the system interacts with a user through a command-line interface (CLI), asking a series of binary (Yes/No) questions regarding their physical condition.

Based on the user's responses, the system navigates a logical decision tree to provide a preliminary assessment. This project demonstrates the practical application of conditional control structures (**if-elif-else**) in solving real-world decision-making problems.

2 Problem Statement

The Problem: In the absence of immediate professional medical advice, individuals often struggle to accurately assess the severity of their physical symptoms. This leads to two opposing issues:

1. **Critical Delays:** Patients with life-threatening conditions (e.g., chest pain) may hesitate to call emergency services.
2. **Resource Strain:** Patients with minor ailments may overcrowd healthcare facilities unnecessarily.

The Solution: A digital tool that performs an initial “triage”—sorting patients based on urgency—to guide them toward Emergency Care, a Doctor Visit, or Home Care.

3 Functional Requirements

The system must fulfill the following functional requirements:

- **Input Handling:** The system must accept user inputs via the console.
- **Input Validation:** The system must handle case sensitivity (e.g., accepting “YES”, “yes”, “Yes”) and reject invalid inputs.
- **Emergency Detection:** The logical flow must prioritize life-threatening symptoms immediately.
- **Branching Logic:** The system must support at least two distinct diagnostic paths (e.g., Fever vs. Stomach Pain).
- **Assessment Output:** The system must display a clear, readable recommendation at the end of the session.

4 Non-Functional Requirements

- **Usability:** The interface must be text-based but clear and easy to read for non-technical users.
- **Performance:** The response time for each question should be instantaneous (< 0.1 seconds).

- **Reliability:** The program must run without crashing on standard inputs.
- **Portability:** The script must be executable on any machine with a standard Python 3.x installation.

5 System Architecture

The system follows a simple linear script architecture running within the Python Interpreter environment. The architecture consists of:

- **User Interface Layer:** Handles input/output operations via the standard system console.
- **Application Logic Layer:** Contains the Decision Tree implemented via conditional statements.
- **Data Layer:** Uses transient runtime variables; no persistent storage is required.

6 Design Diagrams

6.1 Use Case Diagram

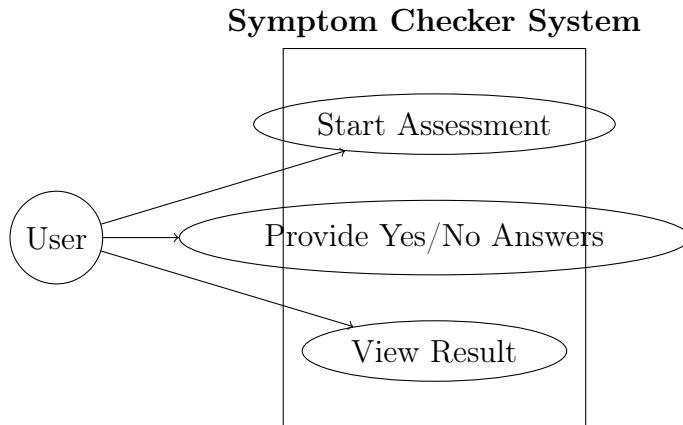


Figure 1: Use Case Diagram: Interaction between User and System.

6.2 Workflow Diagram (Flowchart)

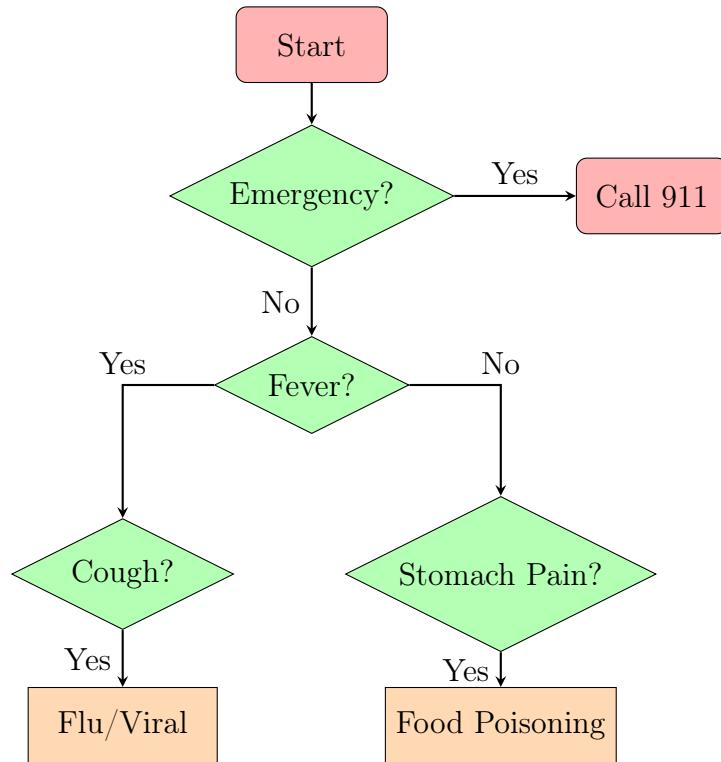


Figure 2: Simplified Workflow Diagram representing the Decision Tree.

6.3 Sequence Diagram

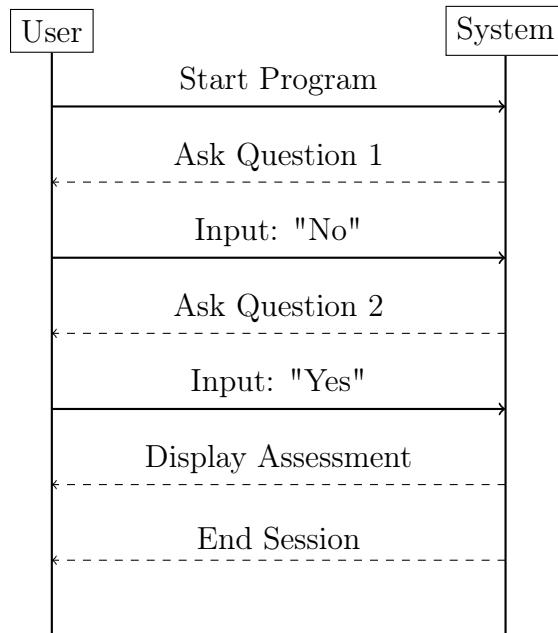


Figure 3: Sequence Diagram demonstrating a typical interaction flow.

6.4 Component Diagram

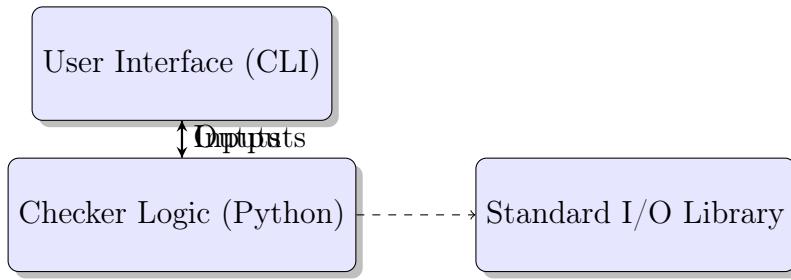


Figure 4: Component Diagram showing modular dependencies.

6.5 ER Diagram

Not Applicable. This system utilizes in-memory processing and does not maintain a persistent database or entity relationships.

7 Design Decisions & Rationale

1. **Language Selection (Python):** Python was chosen for its readability and rapid prototyping capabilities. The concise syntax allows the logical flow to be easily audited by non-developers.
2. **Data Structure (Nested If/Else):** For a small-scale checker, nested conditional statements provide the most direct implementation of a binary tree. While a dictionary-based graph approach is more scalable, the `if/else` approach was selected for this iteration to maximize code transparency for educational purposes.
3. **Input Sanitization:** The use of `.lower().strip()` was a deliberate design choice to improve user experience, ensuring that variations in capitalization or whitespace do not break the program.

8 Implementation Details

The core logic is implemented in a single function `start_checker()`.

```

1 def start_checker():
2     # Question 1: Emergency Check
3     ans = input("Chest pain/breathing difficulty? ").lower().strip()
4
5     if ans == "yes":
6         print(">>> URGENT: Call emergency services.")
7         return
8
9     else:
10        # Check for fever
11        ans = input("Do you have a fever? ").lower().strip()
12        if ans == "yes":
13            # ... logic continues ...
  
```

Listing 1: Core Logic Snippet

9 Screenshots & Results

Below are representations of the console output during execution.

9.1 Scenario 1: Emergency Path

Terminal Output

```
-- Basic Health Symptom Checker --
Please answer 'yes' or 'no'.
-----
Are you having chest pain or difficulty breathing? yes
»> URGENT: Call emergency services (911/112) immediately.
```

9.2 Scenario 2: General Assessment

Terminal Output

```
Are you having chest pain or difficulty breathing? no
Do you have a fever? yes
Do you have a dry cough? yes
»> ASSESSMENT: Possible viral infection or Flu.
```

10 Testing Approach

Testing was conducted manually using Black Box Testing techniques.

1. **Positive Testing:** Valid “yes” and “no” inputs were tested for every logical branch to ensure the correct assessment was reached.
2. **Negative Testing:** Inputs such as “maybe”, numbers, and empty strings were entered to verify the system’s error handling mechanisms (or default else behavior).
3. **Boundary Testing:** Tested mixed case inputs (e.g., “YeS”) to verify the sanitization logic.

11 Challenges Faced

- **Logic Depth:** Managing deep nesting of `if/else` statements became visually confusing. Proper indentation and commenting were crucial to maintain readability.
- **User Ambiguity:** Users might want to answer “sometimes” or “a little bit”. Since the logic is binary (Yes/No), framing the questions precisely was a challenge to ensure users could answer definitively.

12 Learnings & Key Takeaways

This project provided valuable insights into software logic design:

- **Importance of Triage:** I learned that in health applications, the order of questions matters. Critical questions must always come first to ensure safety.
- **Input Handling:** I realized that users are unpredictable. Robust input cleaning (like handling upper/lower case) is essential for a smooth user experience.
- **Complexity Management:** While nested ‘if’ statements work for small programs, I learned that they become hard to manage as the program grows, suggesting that data-driven structures (like dictionaries) would be better for larger systems.

13 Future Enhancements

1. **GUI Implementation:** Migrating from CLI to a Graphical User Interface using Tkinter or PyQt for better accessibility.
2. **Fuzzy Logic:** Implementing AI or fuzzy logic to handle non-binary answers (e.g., “Mild pain” vs “Severe pain”).
3. **Knowledge Base:** Storing symptoms in a JSON/SQL database to allow dynamic updates without changing the source code.

14 References

1. Python Software Foundation. (2025). *Python 3.12 Documentation*. retrieved from python.org.
2. Mayo Clinic Staff. (2024). *Symptom Checker Logic Principles*.