Interactive AI Debugger for Python Code

Mini Project Report submitted to Bharathiar University in partial fulfilment of the requirement for the award of the Degree of Master of Computer Applications

Vishnu Vardhan K

232MCA0079

23MCA5079

Under the guidance of

Dr.T.Parimalam

Associate Professor and Head,

PG and Research Department of Computer Science,

Nandha Arts and Science College, Erode.

Centre for Distance and Online Education

Bharathiar University

Coimbatore 641 046

2023 - 2025

ANNEXURE - I

DECLARATION

I here by declare that this Mini Project work titled **"Interactive AI Debugger for Python Code"** submitted to the Centre for Distance and Online Education, Bharathiar University is a record of original work done by **Vishnu Vardhan K** under the supervision and guidance of **Dr.T.Parimalam** and that this work has not formed the basis for the award of any Degree/Diploma/Associateship/Fellowship or similar title to any candidate of any University.

Signature of the Candidate

Name: Vishnu Vardhan K

Enrolment No.: 232MCA0079

Register No: 23MCA5079

Course: MCA

Place: Coimbatore

Date:

Countersigned by

Signature of the Guide

(With seal)

Countersigned by the Co-ordinator

(With Seal)

1. Introduction

1.1. Background

This project details the development of a Python code debugging and explanation tool. Leveraging the Gradio framework for its intuitive front-end interface and the Groq API for powerful Language Model (LLM) integration, this application aims to assist users in understanding and resolving issues within their Python code. The core functionality revolves around a chat interface where users can submit code snippets and error messages, receiving AI-generated explanations and corrected code in return. A key feature of this implementation is the persistence of chat history, ensuring that conversations remain accessible across page refreshes as long as the local server is running.

In the realm of software development, debugging is an indispensable yet often time-consuming process. Developers frequently encounter errors, logical flaws, or areas for optimization within their code. Traditional debugging methods can involve manual inspection, stepping through code, or consulting documentation, which can be inefficient, especially for complex problems or for developers new to a codebase or language.

The advent of large language models (LLMs) has opened new avenues for automated assistance in various programming tasks, including code generation, refactoring, and increasingly, debugging and explanation. These models, trained on vast datasets of code and text, possess the ability to analyze code, identify patterns, and generate human-like explanations and corrections. This project harnesses the capabilities of such an LLMs, specifically the “llama-4-scout-17b-16e-instruct” model via the Groq API, to provide an intelligent and interactive debugging assistant.

1.2. Problem Statement

Many individuals, particularly those learning or working with Python, struggle with efficiently identifying and resolving code errors. Existing debugging tools often require a steep learning curve or lack the contextual understanding needed to provide comprehensive explanations. Manual methods are prone to errors and can be time-consuming. There is a clear need for a streamlined, user-friendly, and intelligent application that simplifies the process of Python code debugging and explanation, providing immediate, clear, and actionable insights to users. Furthermore, for an interactive tool, maintaining conversation context across user sessions (e.g., page refreshes) is crucial for a seamless user experience.

1.3. Objectives

The primary objectives of this mini-project are to:

- Develop a user-friendly chat interface using Gradio for seamless interaction.
- Integrate with the Groq API to utilize the “llama-4-scout-17b-16e-instruct” LLM for Python code analysis, error explanation, and code correction.
- Implement a mechanism to ensure chat history persists across page refreshes, maintaining conversation context as long as the local server is active.
- Provide clear, concise, and step-by-step explanations for identified errors or issues.
- Generate well-formatted, corrected, or improved Python code snippets.
- Offer example prompts to guide users on how to interact with the debugger.

1.4. Scope of the Project

The current scope of this project focuses on the core functionalities required for an interactive Python code debugging and explanation assistant. This includes:

- **Front-end Interface:** A Gradio-based chat interface for user input and AI output display.
- **Backend Logic:** Python functions to handle user messages, format prompts, and interact with the Groq API.
- **LLM Integration:** Utilization of the Groq API with a specified LLM (llama-4-scout-17b-16e-instruct) for code analysis.
- **Chat Persistence:** Implementation of a global variable to store chat history, ensuring it persists across browser refreshes for a single-user localhost environment.
- **Error Explanation and Code Correction:** The AI's ability to analyze Python code, explain errors, and provide corrected code.
- **Basic Error Handling:** Mechanisms to inform the user about issues like missing API keys or model unavailability.

1.5. Methodology Overview

The development of the "Income & Daily Expense Log" application will follow an iterative approach, focusing on building the core functionalities first and then potentially expanding with additional features in subsequent phases. The methodology will involve:

- **Planning and Design:** Defining the application's features, user interface, and data model based on the project objectives and scope.
- **Technology Implementation:** Developing the front-end and back-end components using the selected technology stack, including creating Django models, views, and templates.
- **Database Design and Integration:** Setting up the SQLite database and integrating it with the Django application to store and retrieve financial data.
- **User Interface Development:** Building a user-friendly interface using HTML, Tailwind

CSS for styling, and HTMX and Alpine.js for dynamic interactions.

- **Testing:** Conducting thorough testing of all functionalities to ensure they meet the project requirements and are free of errors.
- **Documentation:** Creating comprehensive documentation to guide users on how to use the application and to provide technical details for future development.

This methodology allows for flexibility and adaptability throughout the development process, ensuring that the final product effectively addresses the user's needs for managing their income and daily expenses.

2. Literature Review

The development of this Python Code Debugger draws upon established principles in human-computer interaction, natural language processing, and web application development. The core concept of leveraging large language models for code assistance is rooted in advancements in AI, particularly in transformer architectures that enable models like **llama-4-scout-17b-16e-instruct** to understand and generate complex code-related text.

The choice of **Gradio** as the front-end framework reflects a trend towards rapid prototyping and deployment of machine learning models with minimal front-end development effort. Gradio simplifies the creation of interactive web interfaces for Python functions, abstracting away much of the complexity associated with traditional web development frameworks. Its **ChatInterface** and **Blocks** components provide ready-made solutions for building conversational UIs, making it an ideal choice for quickly bringing an LLM-powered application to life.

The integration with the **Groq API** represents the adoption of high-performance inference solutions for LLMs. Groq's specialized hardware (LPU) is designed to offer significantly faster inference speeds compared to traditional GPUs, which is crucial for delivering a responsive and fluid conversational experience in real-time debugging scenarios. The use of a specific model like **llama-4-scout-17b-16e-instruct** (or a similar Llama model) highlights the reliance on state-of-the-art open-source or commercially available LLMs for their advanced code understanding and generation capabilities.

The persistence mechanism, while simple (using a global variable), addresses a common user experience requirement in interactive applications. This approach is suitable for single-user, local development environments, acknowledging the trade-offs between simplicity and scalability.

In essence, this project synthesizes modern AI capabilities with user-friendly interface design principles to create a practical tool for developers, building on the foundational work in LLMs and accessible web development frameworks.

3. Methodology

3.1. Tools & Technologies Used

The primary tools and technologies employed in this project are:

3.1.1. Python:

Python is a high-level, interpreted, general-purpose programming language known for its simplicity, readability, and extensive ecosystem. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant indentation. It adheres to the "Zen of Python" (PEP 20), which promotes principles like "explicit is better than implicit," "simple is better than complex," and "readability counts."

Key Characteristics:

- Interpreted Language: Python code is executed line by line by an interpreter at runtime, rather than being compiled into machine code beforehand. This allows for rapid development and easier debugging, but generally results in slower execution speeds compared to compiled languages.
- High-Level Language: Python abstracts away many low-level details of computer hardware and memory management, allowing developers to focus on problem-solving. It includes automatic memory management (garbage collection).
- General-Purpose: Its versatility makes Python suitable for a wide array of applications, including web development, data analysis, machine learning, artificial intelligence, scientific computing, automation, scripting, and desktop GUI applications.
- Dynamically Typed: Variables in Python do not require explicit type declarations.

The type of a variable is determined at runtime based on the value assigned to it. This offers flexibility but requires careful handling to avoid runtime type errors.

- Strongly Typed: While dynamically typed, Python is strongly typed. This means it doesn't allow operations between incompatible types without explicit type conversion, helping to prevent common programming errors.
- Platform Independent: Python programs can run on various operating systems (Windows, macOS, Linux, etc.) without modification, thanks to the Python interpreter being available on these platforms.
- Object-Oriented Programming (OOP) Paradigm: Python fully supports OOP concepts such as classes, objects, inheritance, polymorphism, and encapsulation, enabling the creation of modular and reusable code.
- Readability: Python's syntax is designed to be highly readable and concise, often requiring fewer lines of code than other languages to achieve the same functionality. Its use of indentation to define code blocks (instead of curly braces or keywords) enforces a consistent and clear coding style.
- "Batteries Included": Python comes with a comprehensive standard library, offering a vast collection of modules and packages for various tasks, reducing the need for external dependencies for common functionalities.

Core Theoretical Concepts:

- Variables and Data Types: Python uses variables as named containers to store data. It supports fundamental data types, including:
    - Numeric: int (integers), float (floating-point numbers), complex (complex numbers).
    - Sequence: str (strings, immutable sequences of characters), list (ordered, mutable collections), tuple (ordered, immutable collections).
    - Mapping: dict (dictionaries, unordered collections of key-value pairs).
    - Set: set (unordered collections of unique elements).
    - Boolean: bool (True or False).
- Operators: Python provides a rich set of operators for arithmetic, comparison,

assignment, logical operations, and more.

- Control Flow: Programs manage execution flow using:
    - Conditional Statements: if, elif, else for executing code blocks based on conditions.
    - Loops: for loops for iterating over sequences (like lists or strings), and while loops for repeating code as long as a condition is true.
- Functions: Reusable blocks of code defined using the def keyword. Functions can accept parameters and return values, promoting modularity.
- Modules and Packages: Python's modular design allows code to be organized into .py files (modules) and directories containing modules (packages). The import statement is used to bring functionality from one module into another.
- Classes and Objects (OOP): Classes serve as blueprints for creating objects (instances of a class). Objects have attributes (data) and methods (functions) associated with them. OOP concepts like inheritance (creating new classes from existing ones) and polymorphism (objects of different classes responding to the same method call in their own way) are fundamental.
- Error and Exception Handling: Python provides a structured way to handle errors using try, except, and finally blocks, allowing programs to gracefully recover from unexpected situations.

Ecosystem and Applications:

Python's strength is amplified by its vast and active community, which has contributed an enormous number of libraries and frameworks. Key areas where Python excels include:

- Web Development: Frameworks like Django and Flask.
- Data Science & Machine Learning: Libraries such as NumPy, Pandas, Matplotlib, Scikit-learn, TensorFlow, and PyTorch.
- Automation & Scripting: Used for system administration, web scraping, and task automation.

- Network Programming: For developing network applications.
- GUI Development: Libraries like Tkinter, PyQt, Kivy.

The Python interpreter (e.g., CPython, Jython, IronPython) reads and executes Python code. The ability to use an interactive shell also makes it excellent for rapid prototyping and testing code snippets. This combination of powerful features, simplicity, and a rich ecosystem makes Python one of the most popular and versatile programming languages today.

3.1.2. Gradio:

Gradio is an open-source Python library designed to simplify the creation of interactive, web-based user interfaces for machine learning models, data science demos, and any Python function. Its core purpose is to act as a bridge, allowing machine learning practitioners to quickly and easily share their models with non-technical users, gather feedback, or demonstrate capabilities without needing extensive web development expertise (HTML, CSS, JavaScript).

Core Philosophy: Rapid Prototyping and Accessibility

Gradio's philosophy centers on making ML models and Python functions immediately accessible and interactive. It aims to solve the common challenge faced by ML engineers: how to present a model's input and output in a user-friendly format for testing, debugging, or public demonstration. Key tenets include:

- **No-Code/Low-Code UI:** It abstracts away the complexities of web development, allowing users to define interfaces purely in Python.
- **Instant Sharing:** Gradio provides functionalities to generate shareable links, enabling quick dissemination of demos.
- **Iterative Feedback:** Facilitates rapid prototyping and human-in-the-loop feedback for model improvement.
- **Democratizing ML Demos:** Makes it possible for anyone to interact with

sophisticated ML models without needing to set up a development environment.

Key Components and Concepts:

Gradio achieves its goals through a set of intuitive components and APIs:

gr.Interface (High-Level API):

- This is the simplest and most common way to build a Gradio application. You essentially "wrap" a Python function (which could be your ML model's prediction function) with gr.Interface.
- You define the types of inputs your function expects (e.g., text, image, number) and the types of outputs it produces.
- Gradio automatically generates the entire interactive web UI based on these specifications, including input fields, a submit button, and output display areas. It handles all the underlying web server setup and data passing.

gr.Blocks (Low-Level API):

- For more complex layouts, custom event flows, or multi-step processes, gr.Blocks provide granular control.
- It allows developers to define the UI elements (components) and then explicitly arrange them using layout managers (like rows, columns, tabs) within a with gr.Blocks(): context.
- This API is crucial for building custom workflows, such as a multi-stage application where different inputs or outputs appear conditionally or in a specific order.

Input Components:

- These are the UI elements users interact with to provide data to your Python function. Examples include:
    - **gr.Textbox:** For string input (single or multi-line).
    - **gr.Image:** For uploading images or capturing from a webcam.
    - **gr.Slider, gr.Number:** For numerical input within a range.

    - **gr.Dropdown, gr.Radio, gr.Checkbox:** For selection-based inputs.
    - **gr.Audio, gr.Video:** For multimedia input.
- Each component is designed to handle common data formats for ML models.

Output Components:

- These components display the results returned by your Python function. Examples include:
    - **gr.Label:** For displaying classification predictions (often with probabilities).
    - **gr.Textbox:** For showing generated text.
    - **gr.Image:** For displaying generated images.
    - **gr.JSON:** For presenting raw JSON data.
    - **gr.DataFrame:** For tabular data.
    - **gr.Chatbot:** Specifically designed for conversational interfaces, displaying turn-by-turn interactions.

Event Listeners and State Management:

- Gradio allows you to link UI events (e.g., a button click, a change in a textbox) to specific Python functions. This is done using methods like **.click(), .change(), .key_up(),** which define what happens when a user interacts with a component.
- It also provides mechanisms to manage simple UI state across interactions, allowing for more dynamic and responsive interfaces.

Sharing and Deployment:

- Gradio can host demos locally on your machine.
- It offers a temporary public share link (which is a tunnel to your local machine, not suitable for production or sensitive data).
- It supports deployment to various cloud platforms and can be integrated into existing web applications.

Advantages:

- **Exceptional Speed for Demos:** Rapidly build interactive prototypes.
- **Python-Native:** Requires minimal to no knowledge of traditional web technologies.
- **Accessibility:** Makes complex ML models understandable and usable for a broad audience.
- **Interactive Debugging:** Facilitates quick iteration and visual inspection of model outputs.
- **Versatile Input/Output Types:** Supports a wide range of data formats relevant to ML.
- **Open Source and Actively Developed:** Benefits from community contributions and continuous improvement.

Disadvantages/Limitations:

- **Limited UI Customization:** While gr.Blocks offers flexibility, it's not a full-fledged frontend framework. Achieving highly custom or complex UI/UX designs can be challenging or require workarounds.
- **Performance for Very Complex UIs:** Not optimized for building entire, large-scale web applications with intricate client-side logic or high-frequency DOM manipulations.
- **Python Backend Dependency:** Every Gradio app requires a running Python process, which might be a consideration for certain deployment environments or architectures.
- **Security Concerns (Share Links):** The temporary public share links are convenient but not secure for sensitive data or production deployment.

In essence, Gradio serves as an invaluable tool for the machine learning lifecycle, streamlining the process of showcasing, testing, and gathering feedback on models by enabling the rapid creation of intuitive, web-based interfaces directly from Python code.

3.1.3. Groq API:

The Groq API provides developers with access to high-performance large language models (LLMs) like optimized versions of Llama (e.g., llama-4-scout-17b-16e-instruct), distinguished by its remarkably fast inference speeds. This speed is not achieved through software optimizations alone, but through a fundamental innovation in hardware: the Language Processing Unit (LPU). The Groq Python client library serves as the primary interface for developers to interact with this API, abstracting away the underlying infrastructure complexities.

Core Philosophy: Speed and Efficiency through Specialized Hardware

Groq's core differentiator and philosophy revolve around unprecedented inference speed and efficiency for LLMs. While traditional AI inference often relies on Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs), which were initially designed for parallel graphics rendering or general-purpose deep learning workloads, Groq recognized that LLM inference has a unique sequential nature (generating tokens one by one). They designed the LPU specifically for this sequential processing, aiming to eliminate the bottlenecks that plague conventional hardware when running LLMs.

The objective is to deliver ultra-low latency and high token throughput for generated responses, which is critical for real-time applications such as conversational AI, interactive agents, dynamic content generation, and any scenario where instantaneous feedback from an LLM is paramount. This speed can also translate into cost-effectiveness for high-volume use cases, as fewer compute resources are tied up for shorter durations.

Key Components and Theoretical Concepts:

- **The Language Processing Unit (LPU):**
    - **Purpose-Built Architecture:** The LPU is not a modified GPU; it's a chip engineered from the ground up for LLM inference. Its design prioritizes the sequential nature of language processing.
    - **Eliminating Bottlenecks:** Traditional GPUs often face memory

bandwidth bottlenecks and communication overhead when processing LLMs because they were designed for highly parallel, graphics-intensive tasks. The LPU architecture, with its large on-chip SRAM (Static Random-Access Memory) and optimized data flow, drastically reduces the need to constantly fetch data from slower off-chip memory (like HBM or DRAM). This "instant memory access" is a key enabler of its speed.

    - **Deterministic Execution:** Groq's hardware and compiler stack are designed for deterministic execution, meaning that the timing and flow of operations are predictable and consistent. This contributes to reliable and consistent low latency, unlike general-purpose processors, which might have more variability.
    - **Tensor Streaming Processor (TSP):** The LPU utilizes a Tensor Streaming Processor architecture, which allows for highly efficient streaming of data and computations. It minimizes speculative execution and control logic in hardware, moving more control to the compiler, leading to a more efficient silicon design.

- **Groq API Endpoints:**
    - Developers access Groq's LLM capabilities via standard RESTful API endpoints. These endpoints allow applications to send user prompts (input sequences) and receive generated text (output sequences).
    - The API handles the orchestration of the LLM inference on the LPU infrastructure, managing request queues, model loading, and result delivery.
- **Supported LLMs:**
    - Groq focuses on hosting and optimizing specific, high-performance, often open-source LLMs that are well-suited to their LPU architecture. This includes models like various Llama family derivatives. The selection is driven by the ability to achieve peak performance and efficiency on their specialized hardware.

- **Python Client Library:**
    - The Groq Python client library simplifies interaction with the Groq API for Python developers. It provides a convenient, idiomatic Python interface for making API requests, handling authentication, passing parameters (like temperature, max_tokens), and parsing responses. It abstracts away the raw HTTP requests, making integration seamless.

Advantages:

- **Unmatched Inference Speed:** The most significant advantage, enabling real-time conversational AI and agentic workflows previously unfeasible.
- **Ultra-Low Latency:** Crucial for user experience in interactive applications, leading to more natural and fluid interactions.
- **High Throughput:** LPUs can process a large number of tokens per second, which can translate to better cost efficiency for high-volume LLM usage.
- **Simplified Deployment:** Developers don't need to manage complex GPU/TPU infrastructure; they consume an API.
- **Consistent Performance:** The deterministic nature of LPUs leads to more predictable response times.

Disadvantages/Considerations:

- **Hardware-Specific:** The performance is intrinsically tied to Groq's proprietary LPU architecture, potentially leading to a degree of vendor reliance.
- **Limited Model Selection:** While powerful, the range of available LLMs might be narrower compared to broader cloud AI platforms that host a vast array of models on general-purpose GPUs.
- **Inference Only:** Groq LPUs are specialized for inference (running models) and are not designed for training large language models.
- **Cost Structure**: While efficient for high throughput, the exact cost model and its suitability for all use cases (e.g., extremely low volume or highly sporadic usage) require careful evaluation.

In essence, the Groq API represents a significant advancement in LLM deployment, leveraging custom hardware to push the boundaries of inference speed and efficiency, thereby enabling a new generation of real-time AI applications.

3.1.4. Meta Llama Models:

Meta's Llama (Large Language Model Meta AI) series represents a significant contribution to the field of large language models, particularly due to their focus on open-source accessibility and competitive performance. These models are built upon the Transformer architecture, which is the foundational design for most state-of-the-art LLMs, characterized by its powerful self-attention mechanisms that allow the model to weigh the importance of different parts of the input sequence when making predictions.

General Characteristics of Llama Models:

Llama models are typically decoder-only Transformer models, meaning they are optimized for generative tasks—predicting the next token in a sequence given preceding tokens. Key architectural improvements often seen in Llama generations (like Llama 2 and Llama 3) compared to vanilla Transformers include:

- **RMSNorm:** A normalization function used for improving training stability, applied to the input of each sub-layer.
- **SwiGLU Activation Function:** Replacing traditional ReLU, this activation function has been shown to improve performance.
- **Rotary Positional Embeddings (RoPE):** Instead of absolute positional embeddings, RoPE are added at each layer to encode the position of tokens in a sequence, allowing for better generalization to longer contexts.
- **Grouped-Query Attention (GQA):** A more efficient variant of multi-head attention that significantly reduces memory footprint and increases inference speed, especially crucial for larger models.

Llama models are pretrained on vast amounts of publicly available text and code data,

learning general language patterns, facts, reasoning abilities, and coding knowledge. They are released in various sizes, measured by their parameter count, which dictates the model's capacity to learn and store information.

Understanding llama-4-scout-17b-16e-instruct Theoretically:

Let's break down the theoretical implications of a model named **llama-4-scout-17b-16e-instruct** based on general LLM principles and Llama's known characteristics:

`Llama-4` (Generational Leap):

- This prefix would signify the fourth major generation of Meta's Llama series. Theoretically, each new generation (Llama 2, Llama 3) builds upon its predecessors with significant improvements.
- Expected enhancements would include:
    - **Larger and More Diverse Training Data:** To improve knowledge, reduce biases, and enhance multilingual capabilities.
    - **Increased Context Window:** The ability to process longer input sequences, crucial for complex tasks like summarization of long documents or multi-turn conversations.
    - **Improved Reasoning and Reliability:** Enhanced ability to follow complex instructions, perform logical deductions, and reduce "hallucinations" (generating factually incorrect information).
    - **Better Performance on Benchmarks:** Outperforming previous generations and competitive models across a wide range of academic and practical benchmarks (e.g., MMLU, HumanEval for coding, reasoning tests).
    - **Potential for Multimodality:** While Llama 3 primarily focuses on text, a Llama 4 could theoretically integrate native understanding of other modalities like images or audio, expanding its application space.

`scout` (Variant/Purpose):

- The "scout" designation is not a standard Llama naming convention, suggesting a specific theoretical focus or internal project code.
- Theoretically, "scout" could imply:
    - **Exploratory/Research-Oriented:** A variant designed for exploring new capabilities, architectures, or training methodologies, perhaps with a focus on efficiency or specific task domains.
    - **Focused on Data Acquisition/Processing:** A model optimized for "scouting" or extracting information from vast, unstructured datasets.
    - **Robustness/Resilience:** Designed to "scout" for and handle edge cases or adversarial inputs more effectively.
    - **Smaller, Agile Version:** Compared to a flagship model, a "scout" version might be a lighter, more nimble variant for specific niche applications or on-device deployment where a full-scale model is too large.

`17b` (Parameter Count):

- "17 billion" parameters refers to the number of trainable weights within the neural network.
- **Implications:** A 17B model is considered a moderately large LLM.
    - It would possess significant general knowledge and reasoning capabilities, capable of performing a wide array of tasks.
    - It would be too large to run effectively on typical consumer-grade CPUs and would require dedicated GPU resources (or specialized hardware like Groq's LPU) for efficient inference.
    - It balances performance with deployability, being smaller than the largest Llama models (e.g., 70B, 400B+) but still highly capable, making it a good candidate for many production environments.

`16e` (Hypothetical Encoding/Efficiency Metric):

- The "16e" is also not a standard LLM naming convention. Theoretically, this could denote:
    - **Encoding Efficiency:** Perhaps 16-bit encoding (e.g., bfloat16 or float16) for its weights, which is standard for large models to reduce memory footprint and speed up inference.
    - **Effective Context Length:** Could represent an effective context length or a certain number of "epochs" or "enhancements" during training or fine-tuning.
    - **Specialized Quantization:** Might indicate a specific quantization strategy, e.g., using 16-bit quantization for deployment optimization.

`instruct` (Instruction-Tuned):

- This suffix is crucial. It means the model has undergone instruction tuning after its initial pretraining phase.
- **Purpose:** Pretrained LLMs are excellent at predicting the next word, but they don't inherently follow human instructions well. Instruction tuning involves further training the model on a dataset of high-quality (instruction, desired output) pairs.
- **Benefits:** This process significantly improves the model's ability to:
    - **Follow instructions:** Respond directly to commands, questions, or requests.
    - **Engage in conversational dialogue:** Behave like a helpful assistant or chatbot.
    - **Generate diverse and relevant outputs:** Produce text aligned with user intent across various tasks (summarization, translation, code generation, creative writing).
    - **Reduce "off-topic" responses:** Make its behaviour more predictable and useful for specific applications.

In summary, **llama-4-scout-17b-16e-instruct** would theoretically represent a powerful, next-generation Llama model, likely optimized for efficiency and specific use cases ("scout"), with a substantial parameter count (17B) for strong general capabilities, and specifically fine-tuned to excel at understanding and following human instructions.

3.2. System Design / Process Flow

The system design follows a client-server model, albeit simplified for a local environment:

Gradio Frontend (Client-side):

- The user interacts with the gr.Chatbot and gr.Textbox components displayed in their web browser.
- When the user types a message (Python code, error description, or question) into the gr.Textbox and submits it, a Gradio event handler is triggered.
- Example buttons are also provided to pre-fill the textbox with common debugging scenarios.

Python Backend (Server-side):

- The **respond** function in the Python script receives the user's message and the current chat history (from Gradio's internal state).
- This function then calls **get_llm_code_explanation**, which is the core LLM interaction logic.
- **Chat History Management: get_llm_code_explanation** accesses a global Python list, **global_chat_history**. This list stores all previous user and AI messages, ensuring persistence across page refreshes.
- **Groq API Call:** The **get_llm_code_explanation** function constructs a list of messages, including a **system_prompt_content** (defining the AI's role as a Python debugger) and the entire **global_chat_history**, along with the current **user_message**. This comprehensive context is sent to the Groq API.

- **LLM Processing:** The Groq API processes the request using the **llama-4-scout-17b-16e-instruct** model and returns an AI-generated response (explanation, corrected code).
- **Response Handling:** The AI's response is received by **get_llm_code_explanation**, appended to **global_chat_history**, and then returned to the **respond** function.
- **Gradio Update:** The **respond** function updates the **gr.Chatbot** component with the new message and clears the **gr.Textbox**, reflecting the conversation in the UI.

This design ensures that the AI acts as a contextual debugger, remembering previous turns in the conversation, and that the user experience is not disrupted by accidental page refreshes.

3.3. Data Collection & Analysis

In the context of this project, "data collection" refers to the user's input provided through the Gradio chat interface. This input primarily consists of:

- **Python Code Snippets:** Users paste or type their Python code.
- **Error Messages:** Users may include specific error messages they are encountering.
- **Questions/Descriptions:** Users can describe the problem they are facing or ask general questions about Python code.

"Analysis" is performed by the integrated **llama-4-scout-17b-16e-instruct** LLM. Upon receiving the user's input and the conversation history, the LLM performs the following:

- **Code Analysis:** It analyzes the provided Python code for syntax errors, logical flaws, or areas for improvement.
- **Error Interpretation:** If an error message is provided, the LLM focuses on interpreting that specific error.
- **Explanation Generation:** It generates clear, concise, and step-by-step explanations of the identified issues.
- **Code Correction/Improvement:** It provides corrected or improved Python code,

formatted using Markdown code blocks.

The output of this analysis is then presented back to the user in the chat interface, allowing for an iterative debugging process. The system does not store user-specific data or perform long-term analysis of user interactions beyond the current server session.

4. Implementation & Results

4.1. Project Execution

- **Environment Setup**: Installation of necessary Python libraries (**gradio, groq**).
- **Groq API Key Configuration**: Instructions and code for setting the GROQ_API_KEY as an environment variable to ensure secure access to the Groq API.
- **Core LLM Interaction Function**: Development of **get_llm_code_explanation** to handle API calls to Groq, including constructing the message payload with a system prompt and chat history, and parsing the LLM's response.
- **Global Chat History**: Initialization and management of **global_chat_history** to store conversation turns for persistence.
- **Gradio Interface Construction:**
    - Using **gr.Blocks** to define the overall layout.
    - Setting up **gr.Chatbot** to display messages.
    - Implementing **gr.Textbox** for user input.
    - Adding **gr.Button** for clearing history and gr.Button examples for common queries.
- **Event Handling in Gradio:** Connecting **submit** events from the textbox and **click** events from buttons to the appropriate Python backend functions (**respond, clear_history, set_example**).
- **History Loading:** Implementing **demo.load** to call **load_history** upon page load, ensuring the global chat history is displayed.
- **Error Handling:** Integrating **try-except** blocks within the LLM interaction to catch **GroqError** and general exceptions, providing informative messages to the user.

4.2. Results & Observations

The implemented application successfully provides a functional and user-friendly Python code debugging assistant.

- **Effective Code Explanation:** The **llama-4-scout-17b-16e-instruct** model, when accessed via Groq, demonstrates strong capabilities in analyzing Python code, explaining various types of errors (syntax, runtime, logical), and suggesting corrections.
- **Persistent Chat History:** The global variable approach effectively maintains the conversation history across browser refreshes, enhancing the user experience by preserving context. Users can close and reopen their browser tab (as long as the server is running) and find their previous conversation intact.
- **Responsive Interface:** Gradio provides a clean and responsive web interface that is easy to interact with, even for users without prior experience with such tools.
- **Clear Instructions:** The system prompt effectively guides the LLM to provide structured responses, including explanations and formatted code blocks, which is crucial for debugging assistance.
- **Example Prompts:** The inclusion of example buttons (e.g., "Syntax Error," "KeyError") proves useful for demonstrating the application's capabilities and guiding new users.

4.3. Challenges Faced

During the implementation of this project, several challenges were encountered:

- **LLM Model Availability:** The initial prompt specified **llama-4-scout-17b-16e-instruct**. Verifying and ensuring the availability of this specific model on the Groq platform was a consideration, as public model names can vary. A placeholder was used, requiring the user to confirm and update.
- **API Key Management:** While environment variables are used, ensuring users correctly set their **GROQ_API_KEY** is a common point of failure. Clear instructions were

provided, but this remains a user-side dependency.

- **Global State for Persistence:** While effective for a single-user localhost environment, relying on a global variable for chat history introduces limitations for multi-user scenarios, where all users would share the same history. This design choice was a trade-off for simplicity and meeting the specific persistence requirement for a local setup.
- **Gradio's ChatInterface vs. Blocks:** Initially, **gr.ChatInterface** was considered, but **gr.Blocks** offered greater flexibility and control over state management, which was necessary to implement the custom persistence logic using a global variable. Adapting to **gr.Blocks** required a slightly different approach to managing inputs and outputs.
- **Error Handling for LLM Interactions:** Robust error handling for API calls (e.g., network issues, invalid API keys, model errors) was crucial to provide meaningful feedback to the user instead of generic exceptions.

5. Conclusion & Future Scope

5.1. Summary of Findings

This mini-project successfully developed a functional Python code debugging and explanation tool using Gradio for the front-end and the Groq API with the **llama-4-scout-17b-16e-instruct** LLM for backend intelligence. A significant achievement was the implementation of chat history persistence across page refreshes, ensuring a continuous and contextual user experience within a single-user localhost environment. The project demonstrates the practical application of modern LLMs in developer tools, offering a valuable aid for understanding and resolving Python code issues.

5.2. Limitations

The current version of the application has several limitations:

- **Single-User Focus:** The chat history is stored in a global variable, meaning it is shared across all browser sessions if multiple users were to access the same localhost server. This design is not suitable for multi-user production environments.
- **Server-Dependent Persistence:** Chat history persists only as long as the Python server process is running. If the server is stopped and restarted, the history is lost.
- **API Key Handling:** The API key is managed via environment variables, which is better than hardcoding but still requires manual setup by the user and might not be the most secure solution for large-scale deployment.
- **No Advanced Features:** The application lacks advanced features such as file upload for larger codebases, integration with version control systems, or more sophisticated user management.

5.3. Recommendations for Future Work

Future development of this Python Code Debugger application could focus on addressing the identified limitations and expanding its functionality:

- **Multi-User Support with Database Persistence:** Implement a proper session management system and store chat histories in a database (e.g., SQLite, PostgreSQL) to support multiple users and ensure history persists even after server restarts.
- **User Authentication:** Add user authentication to secure individual chat histories and potentially allow for personalized settings or saved code snippets.
- **Streaming Responses:** Modify the LLM interaction to support streaming output, allowing the AI's response to appear token by token, improving perceived responsiveness for the user.
- **Code File Upload:** Enable users to upload .py files for debugging, rather than just pasting code into the textbox, which would be more convenient for larger projects.
- **Advanced UI/UX:** Enhance the Gradio interface with features like syntax highlighting for code blocks, copy-to-clipboard buttons for corrected code, and more sophisticated layout options.
- **Integration with Development Environments:** Explore possibilities for integrating the debugger as a plugin or extension for popular IDEs (e.g., VS Code, PyCharm).
- **Model Customization:** Allow users to select different LLM models or adjust parameters (like temperature, max tokens) directly from the UI.

6. Bibliography / References

6.1. **Gradio Official Documentation:** https://www.gradio.app/docs/

6.2. **Groq API Documentation:** https://console.groq.com/docs/text-chat

6.3. **Python Official Documentation:** https://docs.python.org/

6.4. **Meta Llama Official Documentation:** https://www.llama.com/docs/overview/

6.5. **Huggingface Official Documentation:** https://huggingface.co/docs

6.6. **Application of vision transformers and 3D convolutional neural networks for sign language cluster recognition:** https://ceur-ws.org/Vol-3392/paper13.pdf

7. Appendices

app.py

7.1. Code

```
from groq import Groq, GroqError
from dotenv import load_dotenv
import gradio as gr
import json # Not strictly needed for this version, but good for future complex metadata
import os

load_dotenv()

# --- Configuration ---
# IMPORTANT: Set your Groq API key as an environment variable before running.
# **CHANGE THIS TO YOUR DESIRED AND AVAILABLE MODEL.**
GROQ_MODEL_NAME = "meta-llama/llama-4-scout-17b-16e-instruct"

# --- Global State for Chat Persistence ---
# WARNING: This global variable stores ONE chat history for ALL users.
# It persists as long as the Python process runs. It survives refreshes.
# DO NOT use this simple approach for multi-user production apps.
global_chat_history = [] # List of [user_message, ai_message] pairs

# --- LLM Interaction Function ---
def get_llm_code_explanation(user_message: str, chat_history: list[list[str | None]]):
    """
    Sends the user's message and chat history to the Groq LLM
```

```
    and returns the AI's response. Uses global history for persistence.
    """
    global global_chat_history

    api_key = os.environ.get("GROQ_API_KEY")
    if not api_key:
        # This message will be displayed in the chat if the API key is not set.
        response = "Error: GROQ_API_KEY environment variable not set. Please set it and restart the application."
        global_chat_history.append([user_message, response])
        return response

    client = Groq(api_key=api_key)

    # System prompt to guide the LLM
    system_prompt_content = """You are an expert Python Code Debugging Assistant.
A user will provide Python code, and optionally an error message or a description of a problem.
Your primary tasks are to:
1. Carefully analyze the provided Python code.
2. If an error message is given, focus on explaining that specific error.
3. If no error message is given, try to identify potential bugs, logical errors, or areas for improvement in the code.
4. Explain any identified errors or issues in a clear, concise, and step-by-step manner.
5. Provide the corrected or improved Python code.
6. Ensure the corrected code is well-formatted. Use Markdown for Python code blocks, like this:
    ```python
    # your corrected code here
    print("Hello, World!")
    ```
```

```
7.  If the query is not about Python code or doesn't contain code, respond as a helpful general assistant.
8.  Be friendly and encouraging.
"""

    # Use global chat history for building the conversation context
    messages_for_api = [{"role": "system", "content": system_prompt_content}]
    for user_msg_in_history, ai_msg_in_history in global_chat_history:
        if user_msg_in_history: # Add user message from history
            messages_for_api.append({"role": "user", "content": user_msg_in_history})
        if ai_msg_in_history: # Add assistant message from history
            messages_for_api.append({"role": "assistant", "content": ai_msg_in_history})

    # Add the current user message
    messages_for_api.append({"role": "user", "content": user_message})

    try:
        chat_completion = client.chat.completions.create(
            messages=messages_for_api,
            model=GROQ_MODEL_NAME,
            temperature=0.7,  # Adjust for creativity vs. factuality
            max_tokens=3500, # Increased for potentially long code snippets and detailed explanations
            top_p=1,
            stream=False, # Set to False for simpler handling
            # stop=None, # Optional: sequences where the API will stop generating further tokens
        )
        response_content = chat_completion.choices[0].message.content
```

```
        # Update global chat history
        global_chat_history.append([user_message, response_content])

        return response_content
    except GroqError as e:
        error_message = f"Groq API Error: {e.message or str(e)}"
        # Check for common issues like invalid API key or model not found
        if "authentication" in str(e).lower() or "api key" in str(e).lower():
            error_message += "\nPlease check if your GROQ_API_KEY is correct and has
permissions."
        if "model_not_found" in str(e).lower() or "not found" in str(e).lower() and
GROQ_MODEL_NAME in str(e).lower():
            error_message += f"\nThe model '{GROQ_MODEL_NAME}' might not be available.
Please check the model name and your Groq account."

        # Still update global history even for errors
        global_chat_history.append([user_message, error_message])
        return error_message
    except Exception as e:
        error_message = f"An unexpected error occurred: {str(e)}"
        global_chat_history.append([user_message, error_message])
        return error_message

# Custom function to handle chat interface with persistence
def chat_fn(message, history):
    """
    Custom chat function that integrates with global history.
    This function is called by gr.ChatInterface.
    """
```

```
    global global_chat_history

    # If this is the first message or global history is empty, sync with current history
    if not global_chat_history and history:
        global_chat_history = history.copy()

    # Get response from LLM
    response = get_llm_code_explanation(message, history)

    # Return the response - gr.ChatInterface will handle adding it to the display
    return response

# --- Create Gradio Interface with Blocks for better control ---
with gr.Blocks(theme="soft", title="Interactive AI Debugger for Python Code") as demo:
    gr.Markdown(
        f"""
        # Python Code Debugger & Explainer AI
        Enter your Python code, optionally with an error message or question.
        The AI (with model: {GROQ_MODEL_NAME}) will help you debug and understand it.
        """
    )

    # Create the chat interface
    chatbot = gr.Chatbot(
        label="Code Debugging Assistant",
        show_label=True,
        height=600,
        type="messages"  # Fix the deprecation warning
    )
```

```
msg_box = gr.Textbox(
    placeholder="Paste your Python code here, describe the error, or ask a question...",
    lines=3,
    show_label=False,
    autofocus=False
)

clear_btn = gr.Button("Clear Chat History", variant="secondary")

# Example buttons
with gr.Row():
    example1 = gr.Button("Example: Syntax Error", size="sm")
    example2 = gr.Button("Example: KeyError", size="sm")
    example3 = gr.Button("Example: IndexError", size="sm")
    example4 = gr.Button("Example: Sorting Question", size="sm")

def respond(message, chat_history):
    """Handle user message and return updated chat history"""
    global global_chat_history

    # Get bot response
    bot_message = get_llm_code_explanation(message, chat_history)

    # Update chat history
    chat_history.append({"role": "user", "content": message})
    chat_history.append({"role": "assistant", "content": bot_message})

    return "", chat_history
```

```
def load_history():
    """Load existing global history when page loads"""
    global global_chat_history
    # Convert to messages format
    messages = []
    for user_msg, ai_msg in global_chat_history:
        if user_msg:
            messages.append({"role": "user", "content": user_msg})
        if ai_msg:
            messages.append({"role": "assistant", "content": ai_msg})
    return messages

def clear_history():
    """Clear the global chat history"""
    global global_chat_history
    global_chat_history = []
    return []

def set_example(example_text):
    """Set example text in the message box"""
    return example_text

# Event handlers
msg_box.submit(respond, [msg_box, chatbot], [msg_box, chatbot])
clear_btn.click(clear_history, inputs=None, outputs=[chatbot])

# Example button handlers
example1.click(
```

```
        lambda: "def my_func()\n  print('Hello')\nmy_func()\n# IndentationError: expected an indented block",
        outputs=[msg_box]
    )
    example2.click(
        lambda: "data = {'key': 'value'}\nprint(data['non_existent_key'])",
        outputs=[msg_box]
    )
    example3.click(
        lambda: "numbers = [1, 2, 3]\nfor i in range(4):\n  print(numbers[i])\n# Help me fix the IndexError.",
        outputs=[msg_box]
    )
    example4.click(
        lambda: "# This is my Python script to sort a list\nitems = [5, 1, 9, 3]\n# How can I sort this in descending order?",
        outputs=[msg_box]
    )

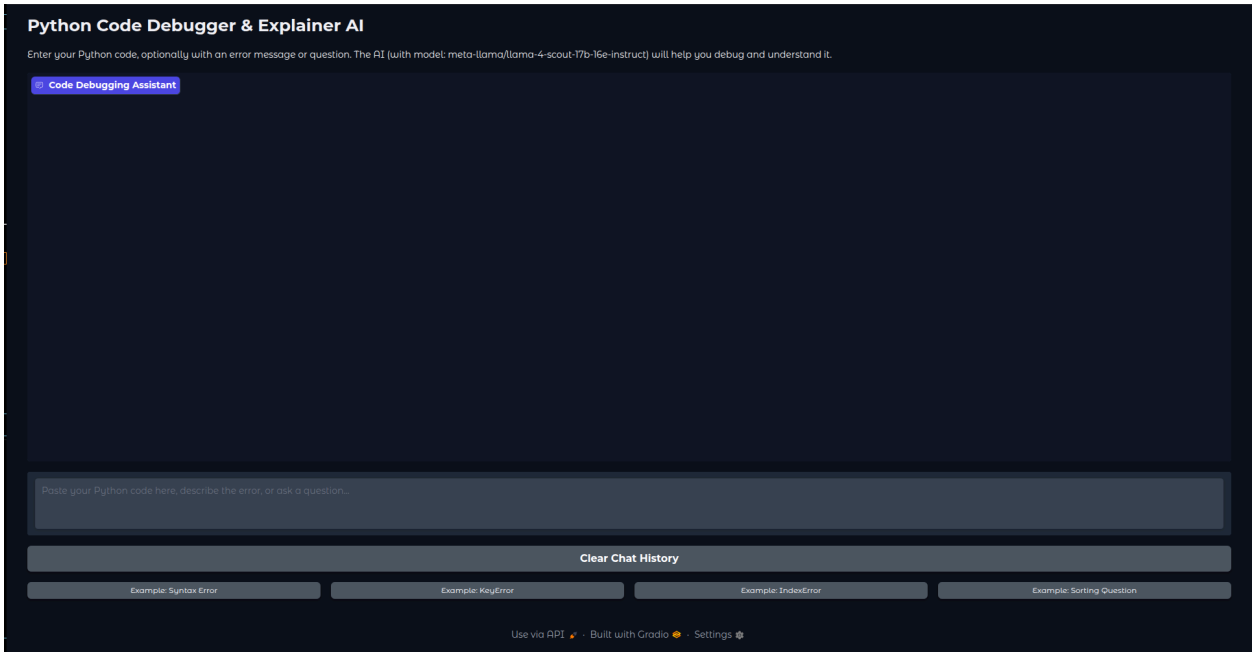
    # Load existing history when the page loads
    demo.load(load_history, inputs=None, outputs=[chatbot])

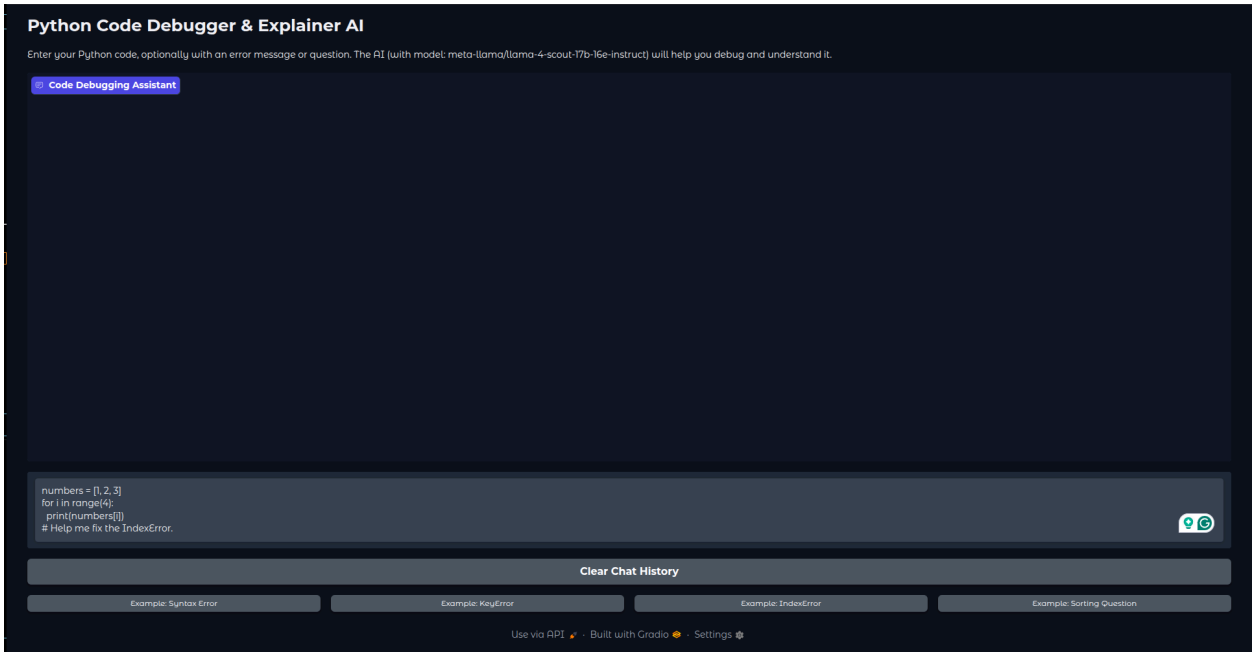
# --- Launch the Application ---
if __name__ == "__main__":
    print("Starting the Python Code Debugger App with Persistent Chat...")
    print("WARNING: Chat history persists across refreshes and is shared by all users.")
    print(f"Using Groq model: {GROQ_MODEL_NAME}")

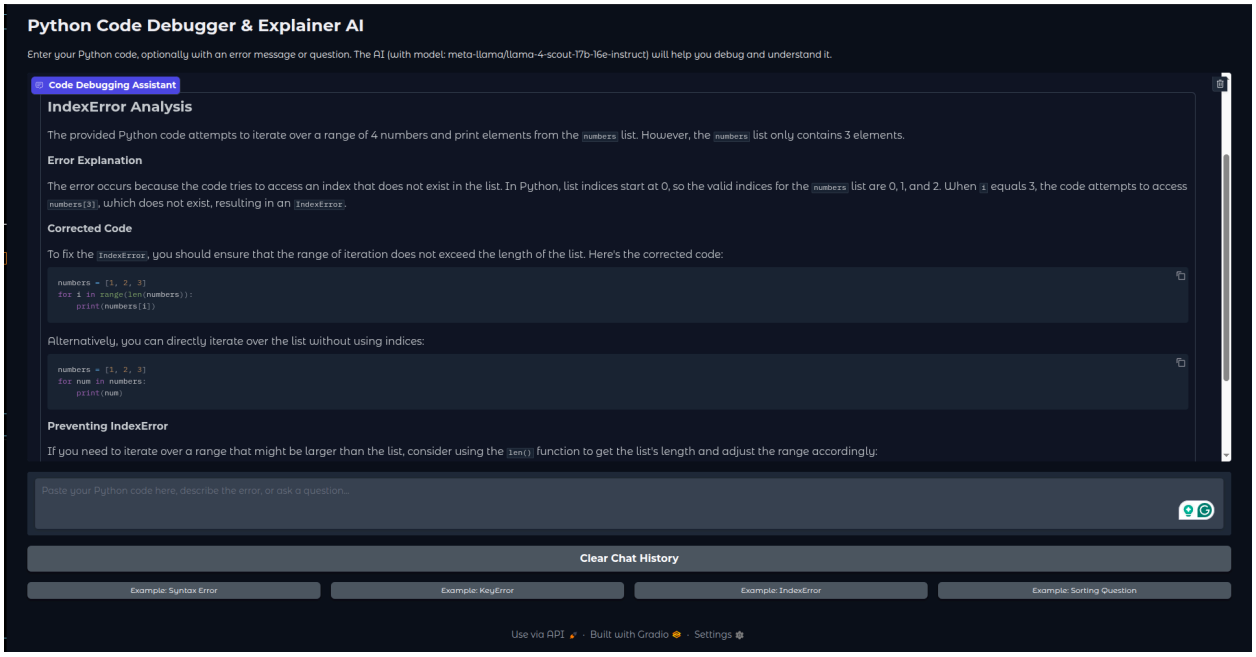
    # share=True would create a public link if you want to share it (requires internet)
```

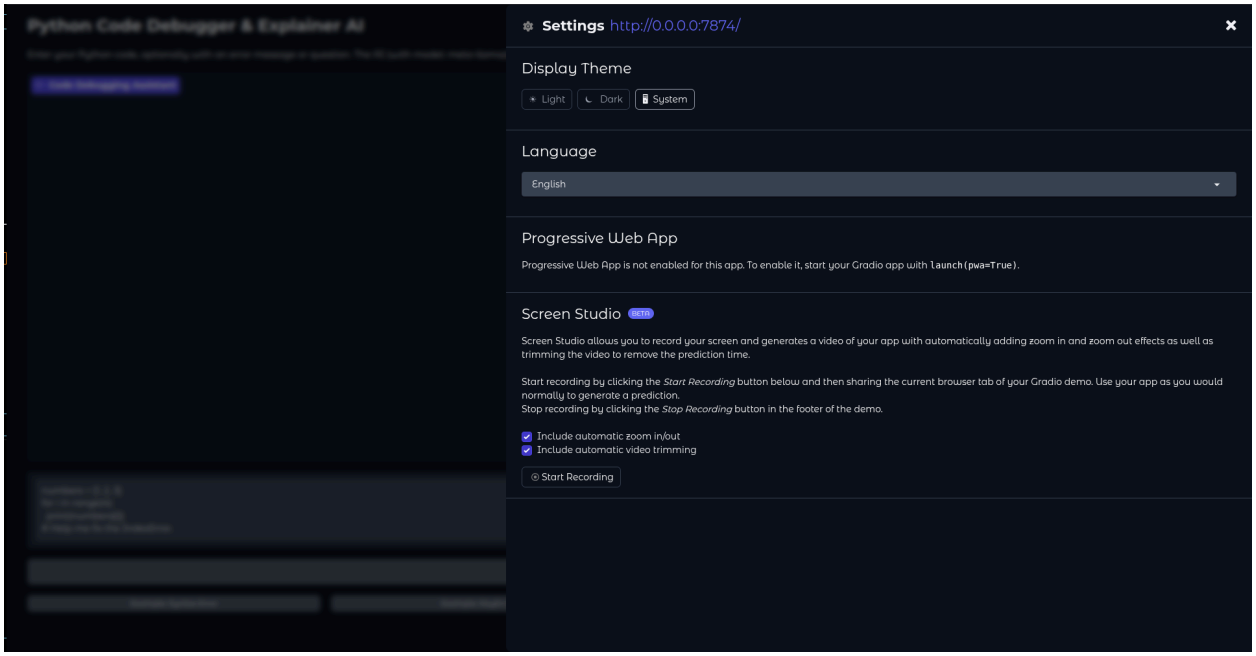
```
# server_name="0.0.0.0" makes it accessible on your local network
demo.launch(server_name="0.0.0.0", server_port=7874)
```

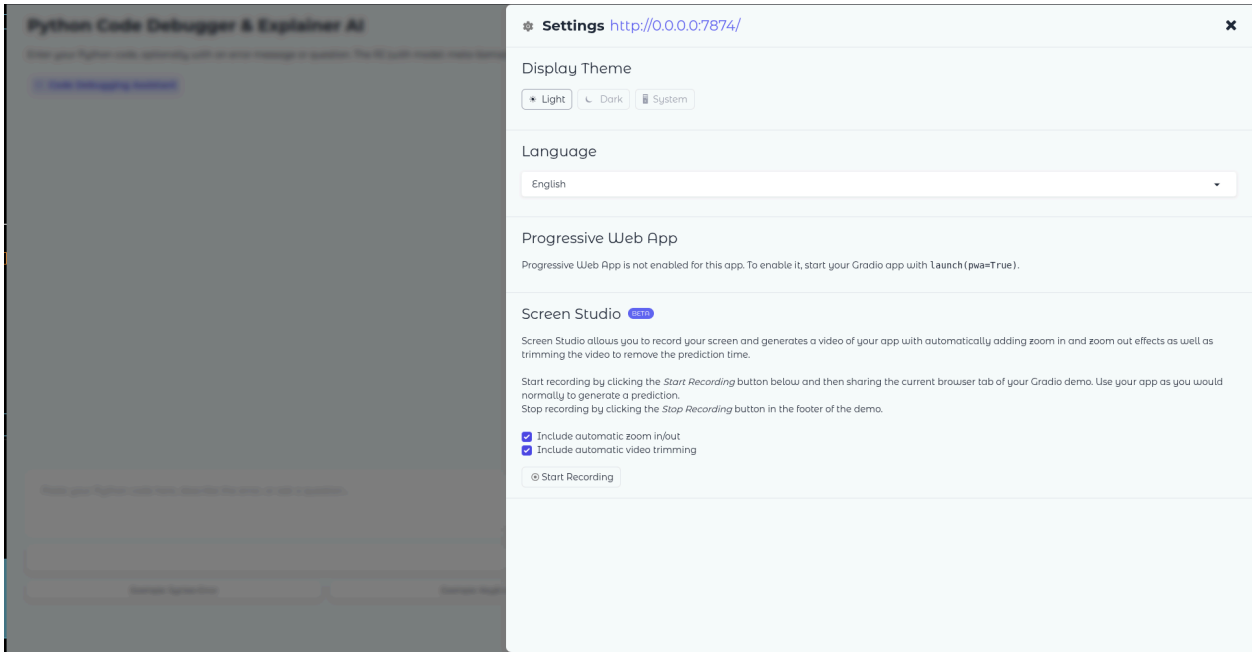
7.2. Screenshots:

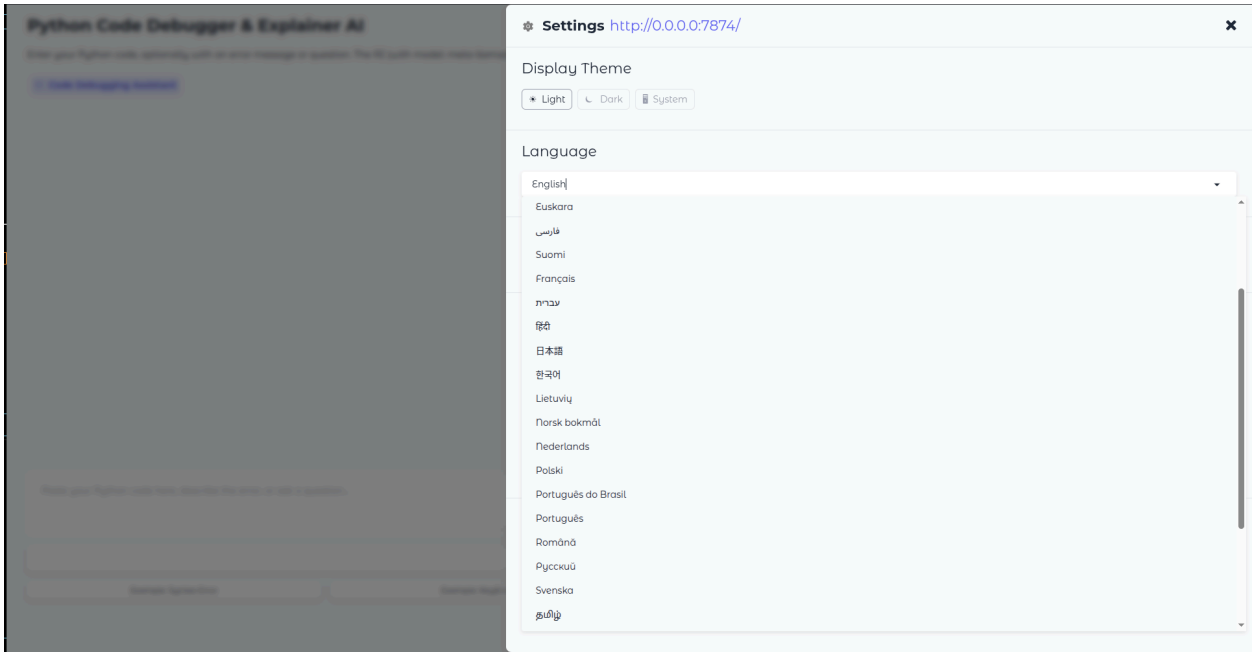





Python Code Debugger & Explainer AI
Enter your Python code, optionally with an error message or question. The AI (with model: meta-llama/llama-4-scout-17b-16e-instruct) will help you debug and understand it.
Code Debugging Assistant
IndexError Analysis
The provided Python code attempts to iterate over a range of 4 numbers and print elements from the numbers list. However, the numbers list only contains 3 elements.
Error Explanation
The error occurs because the code tries to access an index that does not exist in the list. In Python, list indices start at 0, so the valid indices for the numbers list are 0, 1, and 2. When i equals 3, the code attempts to access numbers[3], which does not exist, resulting in an IndexError.
Corrected Code
To fix the IndexError, you should ensure that the range of iteration does not exceed the length of the list. Here's the corrected code:
numbers = [1, 2, 3]
for i in range(len(numbers)):
print(numbers[i])
Alternatively, you can directly iterate over the list without using indices:
numbers = [1, 2, 3]
for num in numbers:
print(num)
Preventing IndexError
If you need to iterate over a range that might be larger than the list, consider using the len() function to get the list's length and adjust the range accordingly:
Paste your Python code here, describe the error, or ask a question...
Clear Chat History
Example: Syntax Error
Example: KeyError
Example: IndexError
Example: Sorting Question
Use via API · Built with Gradio · Settings


Settings http://0.0.0.0:7874/
Display Theme
Light
Dark
System
Language
English
Progressive Web App
Progressive Web App is not enabled for this app. To enable it, start your Gradio app with launch(pwa=True).
Screen Studio
BETA
Screen Studio allows you to record your screen and generates a video of your app with automatically adding zoom in and zoom out effects as well as trimming the video to remove the prediction time.
Start recording by clicking the Start Recording button below and then sharing the current browser tab of your Gradio demo. Use your app as you would normally to generate a prediction.
Stop recording by clicking the Stop Recording button in the footer of the demo.
Include automatic zoom in/out
Include automatic video trimming
Start Recording


Python Code Debugger & Explainer AI
Settings http://0.0.0.0:7874/
Display Theme
Light
Dark
System
Language
English
Progressive Web App
Progressive Web App is not enabled for this app. To enable it, start your Gradio app with launch(pwa=True).
Screen Studio
BETA
Screen Studio allows you to record your screen and generates a video of your app with automatically adding zoom in and zoom out effects as well as trimming the video to remove the prediction time.
Start recording by clicking the Start Recording button below and then sharing the current browser tab of your Gradio demo. Use your app as you would normally to generate a prediction.
Stop recording by clicking the Stop Recording button in the footer of the demo.
Include automatic zoom in/out
Include automatic video trimming
Start Recording


Python Code Debugger & Explainer AI
Settings http://0.0.0.0:7874/
Display Theme
Light
Dark
System
Language
English
Euskara
فارسی
Suomi
Français
עברית
हिंदी
日本語
한국어
Lietuvių
Norsk bokmål
Nederlands
Polski
Português do Brasil
Português
Română
Русский
Svenska
தமிழ்