International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Code Visualizer: An AI-Integrated Python Code Visualization Tool for Enhanced Programming Comprehension

Mrs. Turai Swathi[1], Sreehasa Pendyala[2], Vaddemgunta Tejaswini[3], Abhiram Gundu[4], Boindla Pranay Kumar[5]

[1] Assistant Professor, Dept. of CSE-Data Science, ACE Engineering College, India.
[2345] B.Tech in CSE-Data Science, ACE Engineering College, India.
Emails: swathiturai12@gmail.com, pendyalasreehasa@gmail.com, tejaswini.v317@gmail.com, abhiram.g004@gmail.com, pranayboindla@gmail.com

DOI : https://doi.org/10.5281/zenodo.15743362

A B S T R A C T :

Learning programming effectively requires not only the ability to write code but also a clear understanding of how code executes at runtime. Many beginners struggle to visualize abstract concepts like variable scope, memory management, and control flow, often leading to misconceptions and frustration. Traditional programming environments, while powerful for code editing and execution, fall short in providing intuitive visualization of program behavior, especially for novice learners. Furthermore, the absence of immediate, personalized guidance compounds the learning curve for self-learners and students in large classrooms.

To address these challenges, this paper introduces a web-based Python Code Visualizer with an integrated AI Tutoring System. The platform enables users to step through code execution while clearly displaying the relationships between stack frames, heap objects, and control structures via interactive, graphical representations. The visualization fosters a concrete model of abstract computational processes, thereby enhancing conceptual clarity.

What distinguishes this system from existing tools is the incorporation of an AI-powered tutor that provides real-time feedback, explanations, and debugging assistance. By analyzing the execution trace and user queries, the AI tutor generates context-aware responses tailored to the learner's needs, effectively simulating one-on-one mentorship in a scalable, web-based format. This integration bridges a critical gap in programming education by combining execution visualization with intelligent, adaptive guidance.

Keywords: Programming, Code Visualization, Python, AI Tutoring, Educational Technology, Artificial Intelligence in Education (AIED), Web-based Learning Tools, Interactive Learning Environments

1. Introduction

Mastering programming requires more than just writing syntactically correct code - it requires a deep understanding of how that code behaves during execution. Beginners frequently face challenges when dealing with abstract concepts such as variable scope, memory management, loops, and function calls. These difficulties often lead to confusion, mistakes, and a lack of confidence, making it harder for learners to progress effectively.
While there are several tools and programming environments available, most fall short in two critical areas: clear visualization of program execution and immediate personalized assistance when learners encounter problems. Text-based outputs and static representations often fail to provide the conceptual clarity needed by novice programmers, and existing tools rarely adapt to a user's individual learning needs.
To solve these challenges, we developed Code Visualizer, a web-based code visualization platform using Python, designed specifically to enhance programming comprehension. Code Visualizer visualizes the program's execution in real time, allowing learners to see exactly how their code behaves step by step.

2. Literature Review

Program visualization has long been recognized as an effective approach to improve programming comprehension, particularly for beginners. Several prior studies have explored how visualization tools can bridge the gap between abstract programming concepts and students' mental models of how programs work. In [1], the authors discuss various visualization platforms designed to support beginner programmers. The study emphasizes how tools that show stack frames, variable states, and control flow diagrams help learners grasp abstract execution processes. However, while these visualization tools have been beneficial in classroom contexts, they often lack interactivity and real-time adaptability to individual learners' needs.

Further, [2], explores the relationship between visualization and student engagement. The study highlights that while visual aids can improve understanding, student motivation and sustained interest require more than static visual representations. The authors suggest that engagement is further enhanced when visualization tools provide active learner participation and interactive feedback loops. In [4], researchers assessed how exposure to visualization tools influences learners' motivation in introductory programming courses. The findings demonstrate that while visualization improves comprehension, its impact on motivation significantly increases when learners are provided with interactive, exploratory environments where they can experiment with execution and receive guidance.

Despite these advancements, existing works fall short of providing personalized, real-time assistance in addition to visualization. Most tools evaluated rely solely on static execution diagrams and predefined feedback mechanisms, limiting their adaptability to individual learners' misconceptions or queries. Our project builds upon these foundations by introducing Code Visualizer, an interactive visualization platform enhanced with an AI-powered tutoring system. By combining execution visualization with context-aware AI guidance, our project addresses both conceptual clarity and motivational engagement, offering a more complete and personalized learning experience for programming students.

3. Methodology

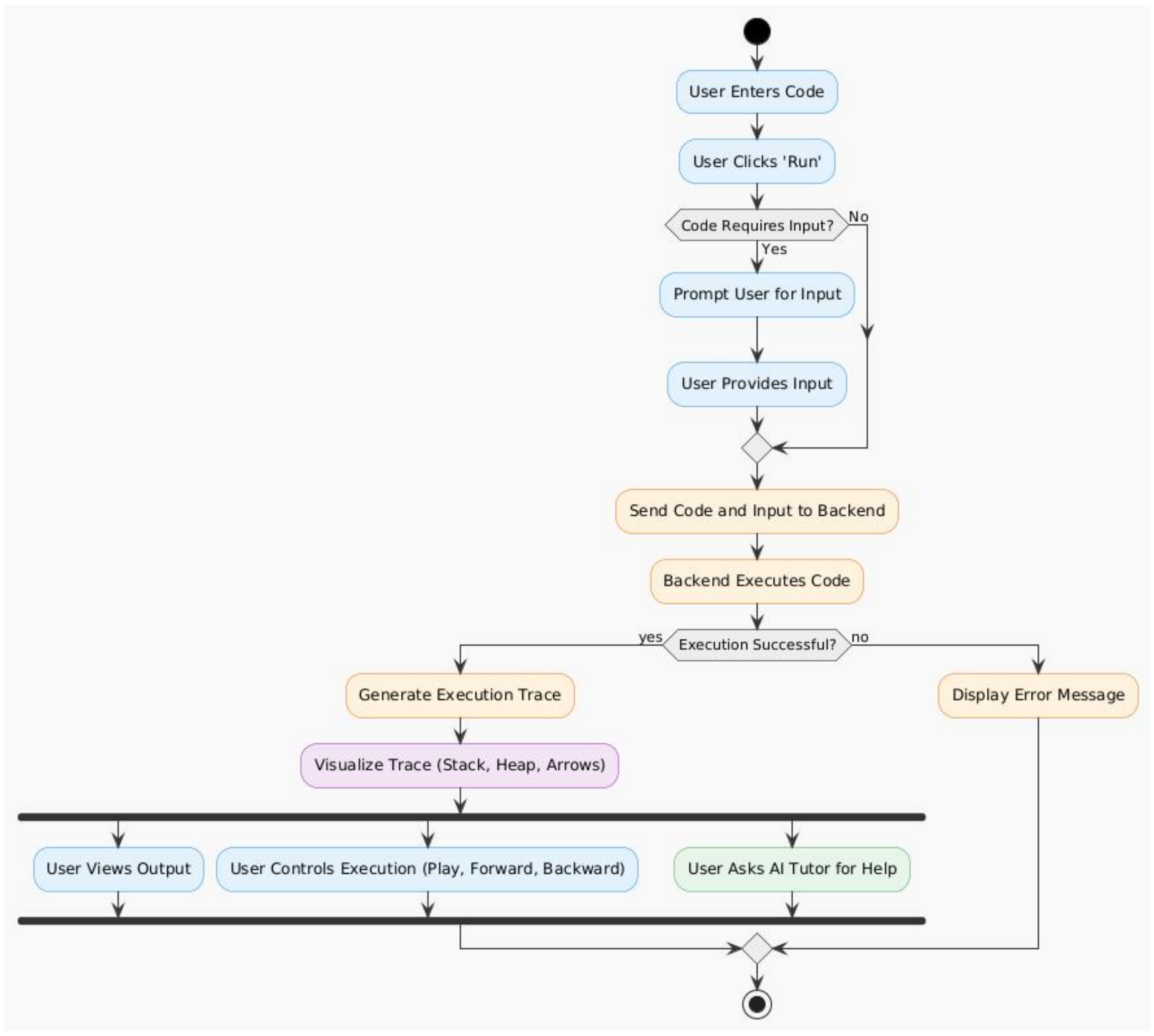


Fig - 1: Methodology for Code Visualizer

The proposed system, Code Visualizer, utilizes interactive visualization and AI-powered tutoring to enhance programming education by helping users better understand code execution. The methodology is divided into three primary phases: Code Submission, Execution & Visualization, and User Interaction with AI Tutoring, as illustrated in Figure-1.

In the Code Submission phase, the user enters Python code into the online code editor. When the 'Run' button is clicked, the code along with the inputs, is transmitted to the backend execution engine. The Execution & Visualization phase begins when the backend securely executes the Python code. If the execution is successful, an execution trace is generated, capturing the program's behavior at each step, including stack frames, heap objects, and control flow. This trace is then rendered visually in the frontend using animated diagrams, allowing the user to observe how variables change, functions execute, and memory is utilized. If execution fails, a detailed error message is displayed to guide the user in debugging.

The User Interaction with AI Tutoring phase enhances the learning experience by providing real-time, personalized help. At any point during visualization, the user can ask for explanations or assistance through the AI-powered tutor. By analyzing the execution trace and the user's query, the AI generates context-aware responses, offering guidance on concepts, debugging strategies, or code improvements. This combination of execution visualization and adaptive AI feedback makes Code Visualizer a comprehensive educational tool for programming learners.
This methodology ensures that users not only see what their code does but also receive instant help to understand why it behaves that way, bridging gaps in comprehension and promoting deeper learning.

3.1.1 Code Submission and Input Handling

- The user writes Python code in the embedded code editor
- The code and user inputs (if there are any) are combined and sent to the backend execution engine via API.

3.1.2 Execution and Trace Generation

- The Flask-based backend securely executes the user's Python code.
- The system generates a step-by-step execution trace, recording:
    - Stack Frames: Active function calls and their variables.
    - Heap Objects: Data structures and objects stored in memory.
    - Control Flow: Line-by-line program execution order.
    - Standard Output: Results from print statements.
    - Error Messages: Detailed error output if execution fails.
- The trace is formatted into a structured JSON object compatible with the frontend visualization.

3.1.3 Visualization Rendering

- The execution trace is rendered in the frontend using an interactive visualization graph.
- Users can step forward or backward through execution to observe:
    - Variable changes
    - Function calls
    - Memory usage
    - Arrows connecting variables to objects for better clarity.
- Printed output is displayed in a dedicated output console.

3.1.4 Error Detection and Display

- If errors or exceptions occur during execution, the backend captures the error trace.
- These error messages are immediately sent to the frontend and clearly displayed to the user.
- The user can request AI assistance for explanations of any error.

3.1.5 AI Tutoring Integration

- Users can ask questions at any point during execution or debugging.
- The AI Tutor Module prepares a structured query with:
    - The original code
    - The current execution state
    - The user's specific query
- This is sent to the AI API, which generates a contextual, natural-language explanation.
- The explanation is displayed alongside the visualization, helping the user understand code behavior or fix mistakes.

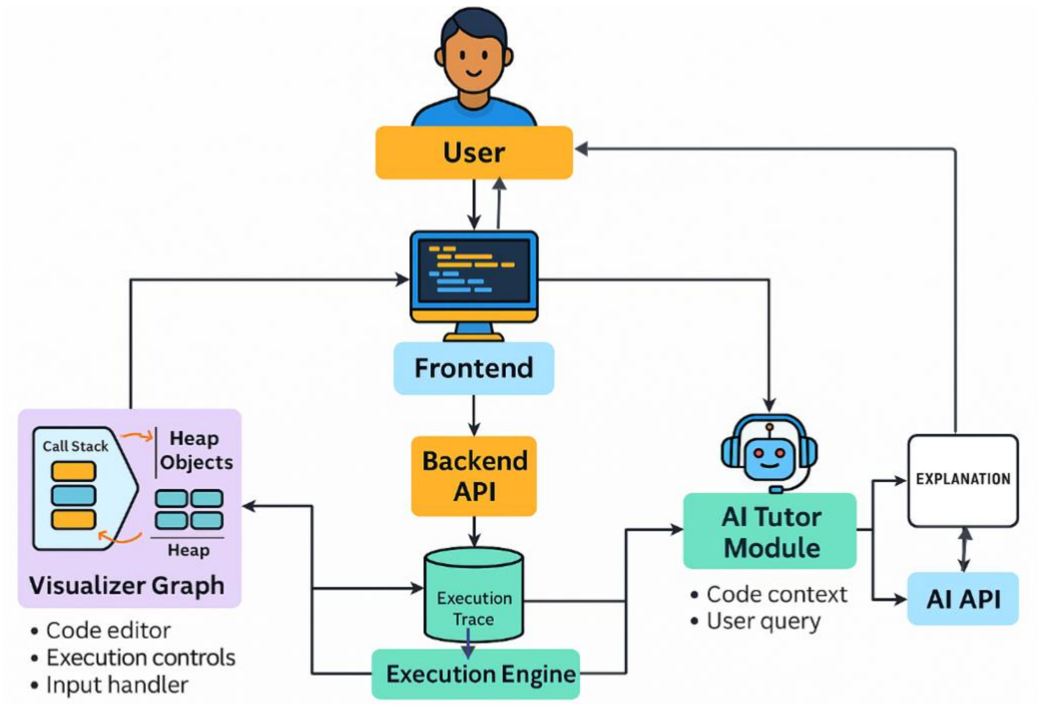
3.1.6 Technology Stack

Component	Technology
Frontend	ReactJS, React-Archer
Backend	Python, Flask

3.1.7 Deployment

- Web-based, lightweight architecture.
- Deployable locally or on web servers for educational use.

3.2 System Architecture



The Code Visualizer system follows a modular client-server architecture designed for flexibility, real-time execution visualization, and intelligent tutoring. The system comprises seven key modules, starting with the user and moves through each part of the system, ending with the code's output and explanations shown to the user:

1. **Code Editor (Frontend):**
   The entry point for users to write and submit code. It provides syntax highlighting and integrates with execution controls (Run, Step, Reset).
2. **Frontend Application:**
   Responsible for managing the UI, handling input prompts for programs, rendering the execution visualization, and managing API communications with the backend.
3. **Backend API (Flask):**
   Implements RESTful endpoints that handle code execution requests, receive inputs, generate execution traces, and interface with the AI tutoring module.
4. **Execution Engine:**
   Executes Python code securely in an isolated environment and produces a detailed trace that represents program execution behavior, including stack frames and heap objects.
5. **Visualizer Graph:**
   A specialized component within the frontend responsible for graphically rendering execution traces, stack frames, memory references, and variable relationships.
6. **AI Tutor Module:**
   Processes user queries, generates the execution context, and sends structured requests to the AI API to receive context-aware explanations for programming concepts, errors, or debugging suggestions.
7. **AI API:**
   Powered by language models, this external service generates natural language explanations that are returned to the frontend for displaying to the user.

This modular separation ensures scalability, maintainability, and the ability to integrate future features such as multi-language support or collaborative coding environments.

4. Work Flow

The Code Visualizer system follows a structured pipeline that integrates code execution visualization and AI-based tutoring to offer an interactive programming learning experience. The complete workflow is detailed below:

Step 1: Code Input

The user accesses the web interface and navigates to the code editor. Using the embedded code editor, the user writes or pastes their Python code. The editor supports syntax highlighting and formatting to help beginners easily write clean and readable code.

Step 2: Code Execution

Once the "Run" button is clicked, the code—along with any input—is sent to the backend via an API. The Flask-based execution engine runs the code in a secure environment and generates an execution trace capturing variable states, stack frames, heap objects, output, and any errors.

Step 3: Error Detection and Display

4. If an error occurs during execution (e.g., syntax errors, runtime exceptions), the backend captures the error message and sends it to the frontend. The Output Console then displays the error clearly to the user, allowing them to understand that something went wrong. This prevents the program from crashing unexpectedly and helps the learner identify problems early.

Step 4: Visualization

After execution, the frontend receives a step-by-step trace of the program. The system then displays a dynamic visualization of stack and heap, along with the control flow between variables and objects. The user can use navigation controls to step forward or backward through the trace to observe how the program evolves at each line.

Step 5: AI Tutor Interaction

At any point during the execution, the user can interact with the AI-powered tutor by asking questions about the code, output, or logic. The backend packages the current execution context along with the user's question and sends it to the AI API, which returns a detailed explanation tailored to the user's query.

Step 6: Output Display

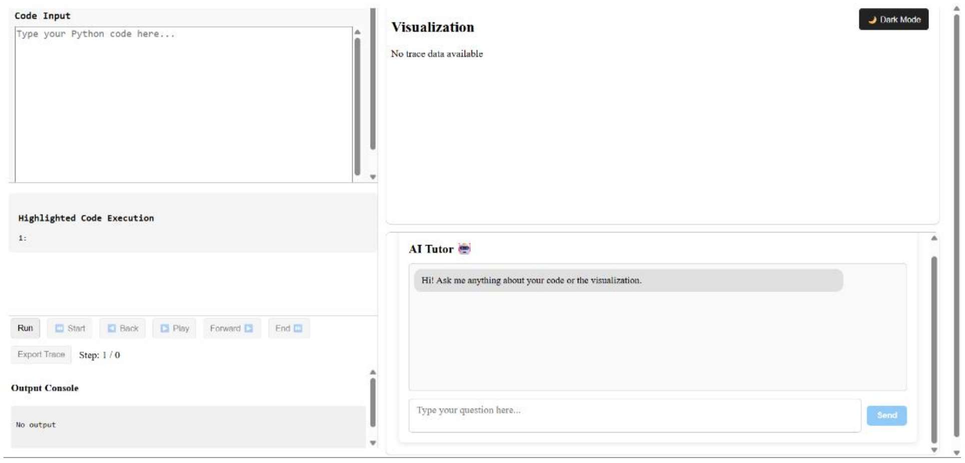
The final result of the program execution—including outputs and any runtime messages—is displayed in a dedicated output console. Alongside, the AI tutor's response is presented in a formatted text box, giving users a complete view of what the code does and why it behaves that way.

Step 7: Learning Loop Completion

This combined experience of writing code, seeing it execute visually, and getting personalized explanations helps users build a strong model of code behavior. It closes the loop between writing, running, debugging, and understanding—all within a single interface.

The system is designed to be lightweight and modular, using JSON-based trace handling and API-driven AI interaction, making it easy to scale and extend without complex infrastructure.

5. Output Screens



Type your Python code here...

Step: 1 / 0

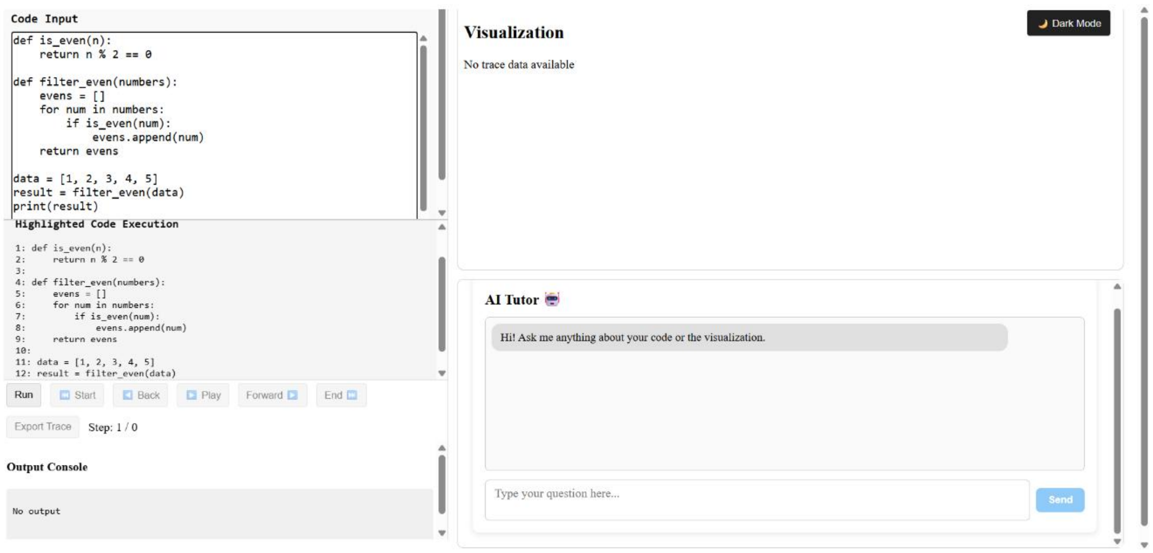
No output

No trace data available

Hi! Ask me anything about your code or the visualization.

Type your question here...

Fig – 1: User Interface of Code Visualizer



```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

data = [1, 2, 3, 4, 5]
result = filter_even(data)
print(result)
```

```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

data = [1, 2, 3, 4, 5]
result = filter_even(data)
```

Step: 1 / 0

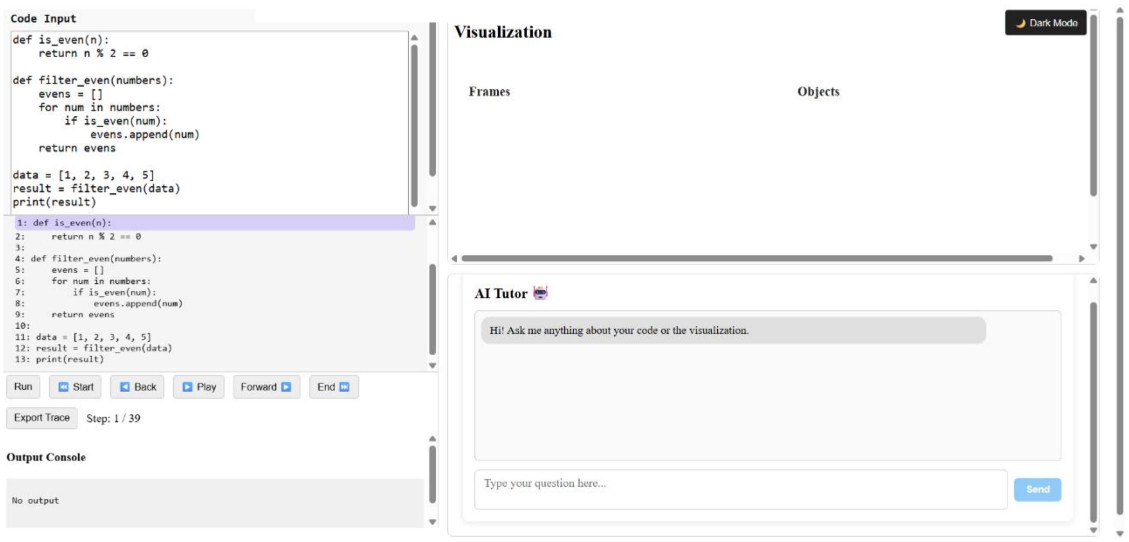
No output

No trace data available

Hi! Ask me anything about your code or the visualization.

Type your question here...

Fig – 2: Writing code into the Code Editor



```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

data = [1, 2, 3, 4, 5]
result = filter_even(data)
print(result)
```

```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

data = [1, 2, 3, 4, 5]
result = filter_even(data)
print(result)
```

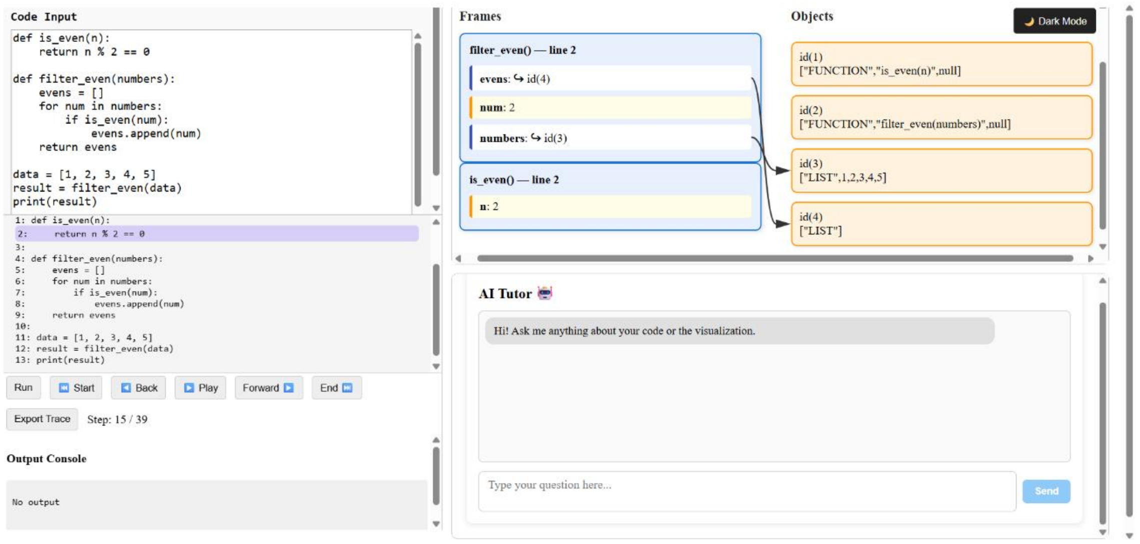
Step: 1 / 39

No output

Hi! Ask me anything about your code or the visualization.

Type your question here...

Fig – 3: Executing the program using the “Run” button


Code Input

```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

data = [1, 2, 3, 4, 5]
result = filter_even(data)
print(result)
```

```
1: def is_even(n):
2:     return n % 2 == 0
3:
4: def filter_even(numbers):
5:     evens = []
6:     for num in numbers:
7:         if is_even(num):
8:             evens.append(num)
9:     return evens
10:
11: data = [1, 2, 3, 4, 5]
12: result = filter_even(data)
13: print(result)
```

Step: 15 / 39

Output Console

```
No output
```

Frames

filter_even() — line 2
evens: ↪ id(4)
num: 2
numbers: ↪ id(3)

is_even() — line 2
n: 2

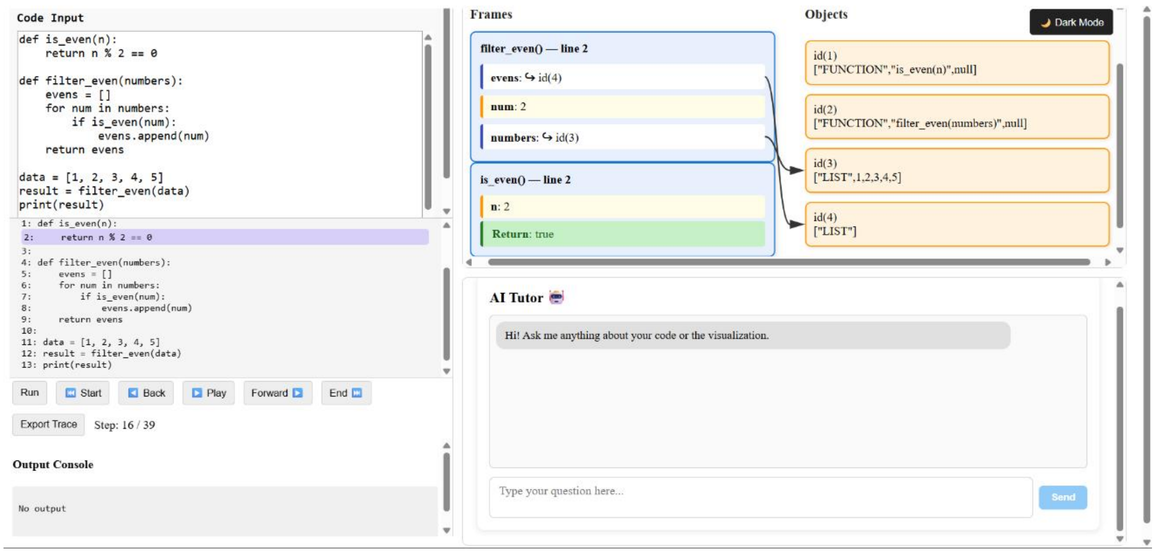
Objects

id(1) ["FUNCTION","is_even(n)",null]
id(2) ["FUNCTION","filter_even(numbers)",null]
id(3) ["LIST",1,2,3,4,5]
id(4) ["LIST"]

AI Tutor

Hi! Ask me anything about your code or the visualization.

Fig – 4: Step-by-Step visualization of the code execution


Code Input

```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

data = [1, 2, 3, 4, 5]
result = filter_even(data)
print(result)
```

```
1: def is_even(n):
2:     return n % 2 == 0
3:
4: def filter_even(numbers):
5:     evens = []
6:     for num in numbers:
7:         if is_even(num):
8:             evens.append(num)
9:     return evens
10:
11: data = [1, 2, 3, 4, 5]
12: result = filter_even(data)
13: print(result)
```

Step: 16 / 39

Output Console

```
No output
```

Frames

filter_even() — line 2
evens: ↪ id(4)
num: 2
numbers: ↪ id(3)

is_even() — line 2
n: 2
Return: true

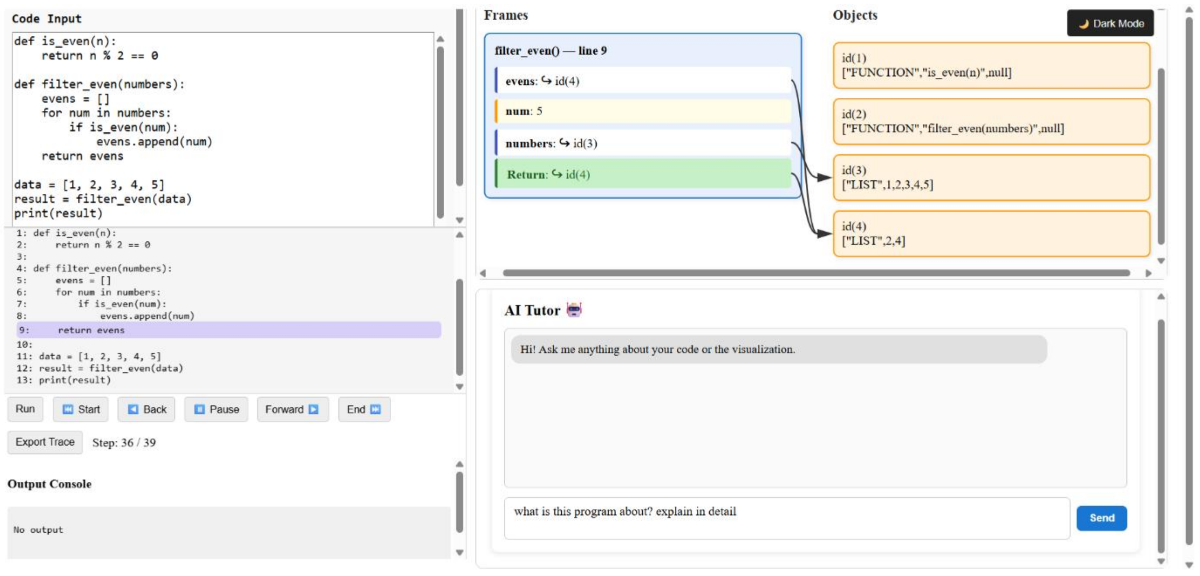
Objects

id(1) ["FUNCTION","is_even(n)",null]
id(2) ["FUNCTION","filter_even(numbers)",null]
id(3) ["LIST",1,2,3,4,5]
id(4) ["LIST"]

AI Tutor

Hi! Ask me anything about your code or the visualization.

Fig – 5: Detailed visualization of the Stack Frames and Heap Objects



Code Input

```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

data = [1, 2, 3, 4, 5]
result = filter_even(data)
print(result)
```

```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

data = [1, 2, 3, 4, 5]
result = filter_even(data)
print(result)
```

Step: 36 / 39

Output Console

```
No output
```

Frames

filter_even() — line 9

evens: ↪ id(4)

num: 5

numbers: ↪ id(3)

Return: ↪ id(4)

Objects

id(1) ["FUNCTION","is_even(n)",null]

id(2) ["FUNCTION","filter_even(numbers)",null]

id(3) ["LIST",1,2,3,4,5]

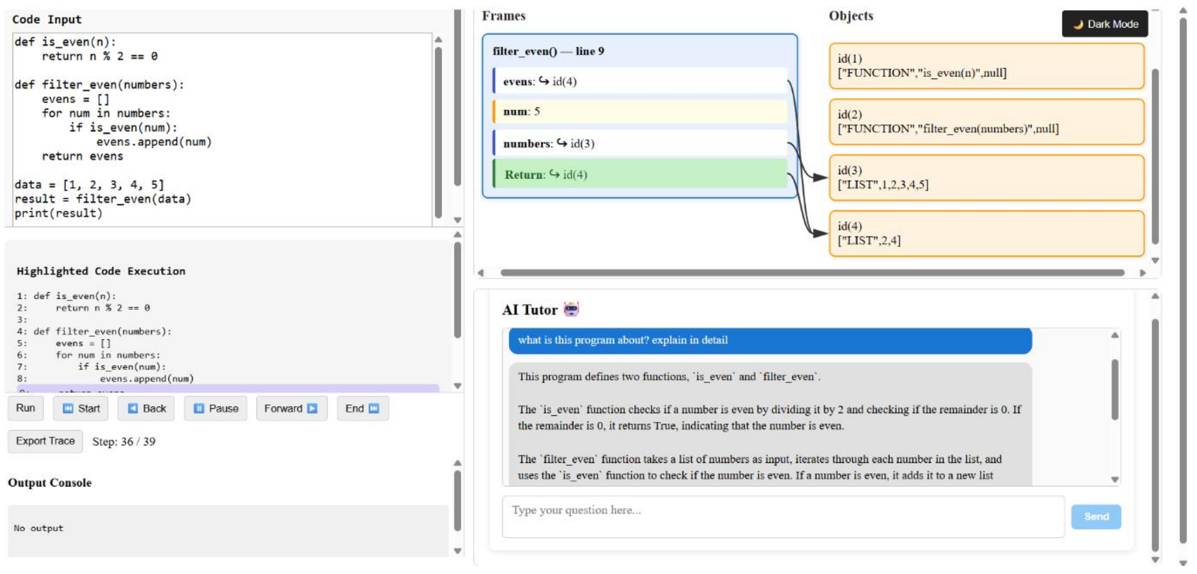
id(4) ["LIST",2,4]

AI Tutor

Hi! Ask me anything about your code or the visualization.

what is this program about? explain in detail

Fig – 6: User interacting with the AI Tutor for conceptual help



Code Input

```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

data = [1, 2, 3, 4, 5]
result = filter_even(data)
print(result)
```

Highlighted Code Execution

```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
```

Step: 36 / 39

Output Console

```
No output
```

Frames

filter_even() — line 9

evens: ↪ id(4)

num: 5

numbers: ↪ id(3)

Return: ↪ id(4)

Objects

id(1) ["FUNCTION","is_even(n)",null]

id(2) ["FUNCTION","filter_even(numbers)",null]

id(3) ["LIST",1,2,3,4,5]

id(4) ["LIST",2,4]

AI Tutor

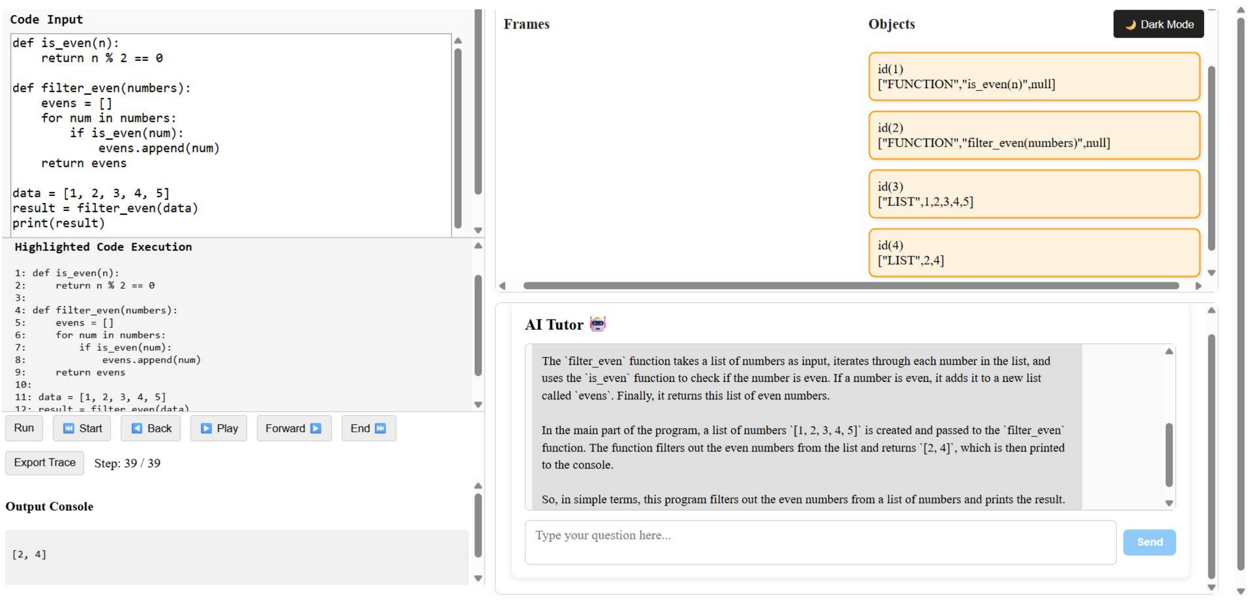
what is this program about? explain in detail

This program defines two functions, `is_even` and `filter_even`.

The `is_even` function checks if a number is even by dividing it by 2 and checking if the remainder is 0. If the remainder is 0, it returns True, indicating that the number is even.

The `filter_even` function takes a list of numbers as input, iterates through each number in the list, and uses the `is_even` function to check if the number is even. If a number is even, it adds it to a new list

Fig – 7: Context-aware explanation provided by the AI Tutor



```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

data = [1, 2, 3, 4, 5]
result = filter_even(data)
print(result)
```

```
def is_even(n):
    return n % 2 == 0

def filter_even(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

data = [1, 2, 3, 4, 5]
result = filter_even(data)
```

```
[2, 4]
```

```
id(1)
["FUNCTION","is_even(n)",null]
id(2)
["FUNCTION","filter_even(numbers)",null]
id(3)
["LIST",1,2,3,4,5]
id(4)
["LIST",2,4]
```

The `filter_even` function takes a list of numbers as input, iterates through each number in the list, and uses the `is_even` function to check if the number is even. If a number is even, it adds it to a new list called `evens`. Finally, it returns this list of even numbers.

In the main part of the program, a list of numbers `[1, 2, 3, 4, 5]` is created and passed to the `filter_even` function. The function filters out the even numbers from the list and returns `[2, 4]`, which is then printed to the console.

So, in simple terms, this program filters out the even numbers from a list of numbers and prints the result.

Fig – 8: Final output display with execution visualization and AI explanation

6. Conclusion and Future Scope

The development of the Python Code Visualizer with AI Tutor represents a significant advancement in the way programmers—particularly beginners and students—learn and understand code execution. This project effectively addresses two of the most common challenges in programming education: the difficulty of visualizing code behavior during execution, and the lack of immediate, personalized guidance.

By integrating a code execution visualizer with a real-time AI tutoring system, the platform not only enables users to observe step-by-step execution but also helps them comprehend programming concepts through interactive, context-aware explanations. This dual approach fosters deeper conceptual understanding, bridges learning gaps, and encourages exploratory, self-directed learning.

In conclusion, this project demonstrates that interactive visualization combined with AI-powered guidance can significantly improve programming education outcomes. It lays a robust foundation for further research and innovation in the field of intelligent educational tools.

Future Scope:

Potential future enhancements of the system include:

- **Multi-language Support:** Extend the visualization engine to support additional programming languages beyond Python.
- **Collaborative Learning:** Introduce real-time collaborative coding sessions for use in classrooms or remote learning environments.
- **Mobile Application:** Develop mobile apps for Android and iOS platforms to enhance accessibility and convenience.
- **Offline Mode:** Provide offline functionality to support learners in regions with limited or unreliable internet connectivity.
- **Adaptive AI Tutor:** Implement adaptive learning models based on user progress to deliver more personalized, skill-appropriate tutoring.
- **Gamification Features:** Incorporate gamification elements such as badges, points, and leaderboards to increase learner motivation and engagement.

7. Acknowledgements

We express our sincere gratitude to all who supported us throughout the development of this project. We are especially thankful to Prof. Y. V. Gopala Krishna Murthy, General Secretary, and Mrs. M. Padmavathi, Joint Secretary, for providing us the opportunity and environment to carry out this work. Our heartfelt thanks to Dr. P. Chiranjeevi, Head of the Department, for his guidance and encouragement. We are deeply grateful to our internal guide Mrs. T. Swathi, and project coordinators Mrs. B. Saritha and Mr. G. Parwateeswar, for their consistent support, valuable feedback, and motivation throughout the project. Lastly, we thank all the faculty members, staff for their constant encouragement and support.

REFERENCES

[1] “Visualization Code Tools for Teaching and Learning Introductory Programming” by Arik Kurniawati, Ari Kusumaningsih, and Mochammad Kautsar Sophan, Dept. of Informatic Engineering, University of Trunojoyo Madura Bangkalan, Indonesia

[2] “Exploring the Role of Visualization and Engagement in Computer Science Education” (Report of the Working Group on "Improving the Educational Impact of Algorithm Visualization") by Thomas L. Naps (co-chair), U Wisconsin Oshkosh, USA, Guido Rößling (co-chair), Darmstadt U Techn., Germany, Rudolf Fleischer, Hong Kong U Sc. & Techn.

[3] “Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education” by Philip J. Guo Google, Inc. Mountain View, CA, USA

[4] “Evaluating the Effect of Program Visualization on Student Motivation” by J. Ángel Velázquez-Iturbide, Senior Member IEEE, Isidoro Hernán-Losada and Maximiliano Paredes-Velasco.