Journal of Computer and Electrical Sciences (JCES)- Volume-1 Issue-4, April 2026
ISSN: 3049-2602

VISUCODE: PYTHON CODE EXECUTION VISUALIZER

Y. Nagalakshmi[1], K. pushpavalli[2]

[1]Student, Department of Computer Science & Engineering

Andhra Loyola Institute of Engineering and Technology, Andhra Pradesh, India.

[2]Associate Professor, Department of Computer Science & Engineering.

Andhra Loyola Institute of Engineering and Technology, Andhra Pradesh, India

Abstract: This paper presents VisuCode, a web-based Python code execution visualizer that transforms static Python source code into dynamic, interactive flowcharts in real time. VisuCode uses Python's built-in Abstract Syntax Tree (AST) module on the backend to parse and simulate code execution, generating a Mermaid.js flowchart that reflects the actual step-by-step execution logic including variable assignments, function definitions and calls, class definitions, conditional branches (if/else), while loops, and for loops. A Flask-based REST API serves the visualization results to a lightweight HTML/CSS/JavaScript frontend, which renders the diagram using the Mermaid.js library. The system also captures print output and tracks live variable state, providing students and developers a tool to understand program flow without executing the code in a traditional interpreter. VisuCode is particularly valuable as an educational aid for beginner programmers learning Python control structures and execution semantics.

Keywords: Python, AST, Code Visualization, Flowchart, Mermaid.js, Flask, Education, Execution Simulation

1. INTRODUCTION

Understanding how a program executes is one of the fundamental challenges for beginner programmers. Traditional tools such as debuggers and interpreters require a specific environment, command-line interaction, and significant background knowledge. Students often struggle to connect the written code to the actual dynamic behavior it produces at runtime. VisuCode addresses this problem by providing an intuitive, browser-based visualization of Python code execution in the form of a Mermaid.js flowchart, produced in real time without running the code in a full interpreter.

The system leverages Python's Abstract Syntax Tree (AST) module, which parses source code into a tree of syntactic nodes representing the structure of the program. A custom AST executor traverses this tree and simulates execution, generating a series of flowchart nodes and edges that represent each statement executed. The resulting diagram is served through a Flask REST API and rendered on a web frontend using Mermaid.js. This approach provides a safe, sandboxed, and pedagogically clear view of program logic.

VisuCode supports a wide range of Python constructs including variable assignments, arithmetic operations, conditional statements, while loops, for loops, function definitions and calls, class definitions and instantiation, and print statements. The system also captures standard output and maintains a live snapshot of variable state, which is displayed alongside the diagram.

2. Literature Survey

Several tools and research efforts have explored code visualization and program comprehension aids for education. Python Tutor, developed by Guo [1], is one of the most widely used tools that visualizes Python execution step-by-step, showing memory state and

https://doi.org/10.5281/zenodo.19627590

variable bindings. While effective, Python Tutor executes code in a sandboxed server and focuses on memory visualization rather than flowchart-based control flow representation.

Scratch [2] and Blockly [3] offer block-based programming environments that visually represent program structure, but are limited to custom block languages and do not accept standard Python syntax. Flowgorithm [4] allows users to draw flowcharts that then generate code, but this is an inverse workflow compared to VisuCode, which generates flowcharts from existing code.

Research in automated flowchart generation from source code has explored parsing-based approaches. Nakakoji et al. [5] discussed how visual representations can significantly improve program comprehension. Studies by Hundhausen et al. [6] showed that algorithm visualizations improve student learning outcomes when students are actively engaged in producing the visualization themselves. VisuCode aligns with these findings by allowing students to write any Python program and immediately see its execution as a diagram.

From the reviewed literature, the following key observations can be made:

- Flowchart-based visualizations offer an intuitive way to understand control flow in programs.
- Existing tools either require code execution in a live interpreter or are limited to block-based languages.
- AST-based static analysis can simulate execution flow without full interpretation.
- Web-based educational tools increase accessibility for students without local environment setup.
- Real-time feedback on code structure significantly improves learning outcomes for beginner programmers.

3. Proposed System

The proposed system, VisuCode, is a full-stack web application consisting of a Python Flask backend and an HTML/CSS/JavaScript frontend. The backend exposes a single POST endpoint (/visualize) that accepts Python source code as a JSON payload and returns a Mermaid.js diagram definition, the captured print output, and a snapshot of variable state.

The core of the backend is the simulate_code() function in simulator.py, which uses Python's built-in ast module to parse submitted code into an AST. A custom Executor class, derived from ast.NodeVisitor, traverses the AST and visits each node type. For each node, it creates a corresponding flowchart node using Mermaid.js syntax and connects it to the previous node. Variable values are tracked in a shared dictionary, allowing expressions to be evaluated and reflected in the diagram labels.

The frontend consists of a textarea for Python code input, a Visualize button, and three output panels: the Mermaid.js flowchart diagram, the standard output panel, and the variables panel. The Mermaid.js library is loaded via CDN and used to render the diagram definition returned by the backend directly in the browser without requiring any additional plugins or installations.

https://doi.org/10.5281/zenodo.19627590

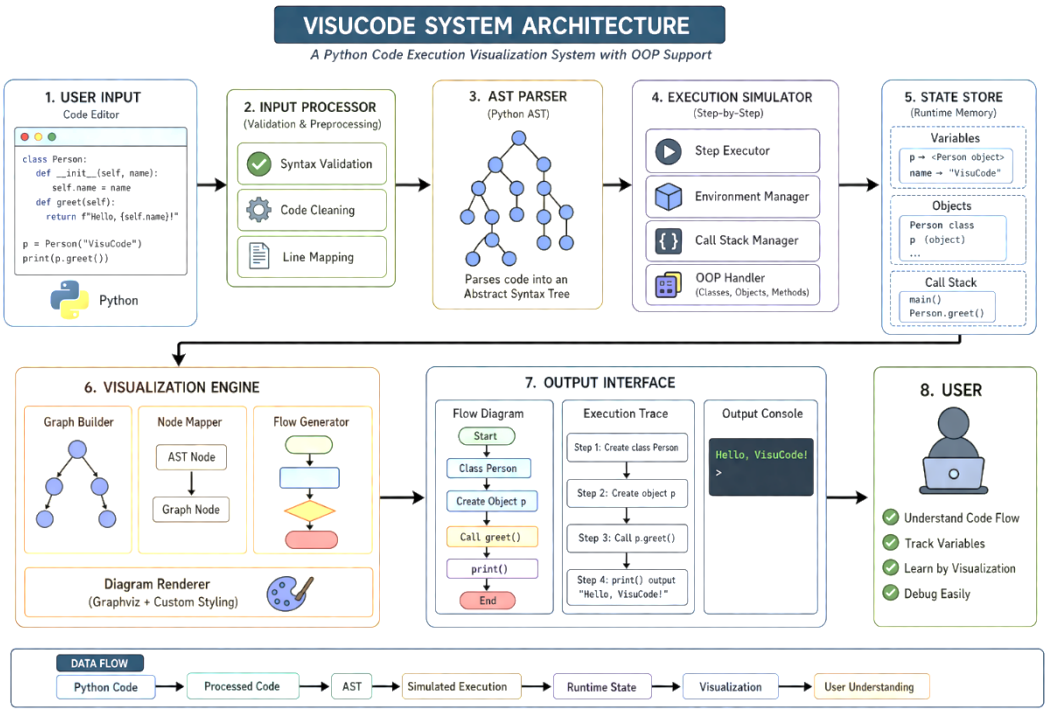

Fig 1: VisuCode System Architecture

The system also incorporates error handling at both the parsing and execution stages. If the submitted code contains syntax errors, the AST parser raises a SyntaxError which is caught and returned as an informative error node in the diagram. If a runtime exception occurs during AST traversal, an Execution Error node is added to the diagram and the traceback is returned as output.

4. Methodology

The methodology of VisuCode is organized into the following steps:

1. Code Submission: The user types Python code into the textarea on the frontend and clicks the Visualize button. The JavaScript fetch API sends a POST request with the code as a JSON body to the Flask backend at the /visualize endpoint.

2. AST Parsing: The backend receives the code and invokes Python's ast.parse() to produce an Abstract Syntax Tree. This step validates the syntax of the code without executing it.

3. AST Traversal and Simulation: The custom Executor class walks the AST. For each supported node type (assignments, conditionals, loops, functions, classes, print calls), the executor creates a Mermaid.js node, connects it to the previous node, and evaluates expressions to update the variable state.

4. Diagram Generation: Each visit_* method appends Mermaid.js node and edge declarations to a running list. Decision nodes (if, while, for) generate branching paths with TRUE and FALSE labels. Loop nodes include a back-edge to the condition node to represent iteration.

5. Output Capture: Calls to the print() function are intercepted by the executor, which evaluates the argument and appends the result to an output list.

6. Response Assembly and Rendering: The backend assembles the diagram string, output lines, and variable dictionary into a JSON response. The frontend receives this, calls mermaid.render() to produce an SVG, injects it into the diagram panel, and displays the output and variables in their respective panels.

https://doi.org/10.5281/zenodo.19627590

7. Error Handling: Both syntax errors (caught at ast.parse()) and runtime errors (caught during AST traversal) are represented as error nodes in the diagram, so the user receives visual feedback even when the code is invalid.

5. Results

VisuCode was tested with a range of Python programs covering the supported construct types: simple variable assignments, arithmetic, if/else branching, for loops, while loops, function definitions with calls, and class definitions with instantiation.

- For a simple assignment program (x = 5, y = 3, z = x + y, print(z)), the system correctly generated a linear flowchart with nodes for each assignment and a Print node at the end, with output '8' displayed in the output panel and the variable state {x: 5, y: 3, z: 8} displayed in the variables panel.
- For an if/else program, the system correctly produced a diamond-shaped decision node with TRUE and FALSE branches, each leading to the corresponding body statements, and merging into a single Merge node.
- For a for loop over a range, the system generated a loop-start node, a repeated decision node ('Has Next Value?'), assignment nodes for the loop variable, body nodes for each iteration, a back-edge to the decision node, and an Exit For Loop node upon exhaustion of the iterator.
- Function definitions were represented as 'Define Function' nodes, and function calls correctly triggered inline execution of the function body with argument binding, followed by return to the call site.
- Class definitions were captured as 'Define Class' nodes, and instantiation triggered __init__ execution and an 'Instantiate' node in the diagram.

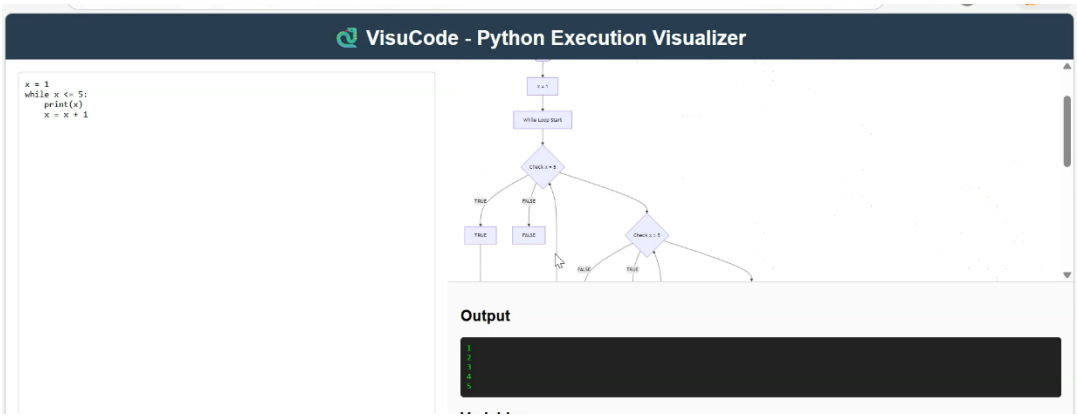

Fig 2: VisuCode Output for a For Loop Program

The system responded to all tested inputs with diagram rendering in under one second on a local development machine. The Mermaid.js library rendered flowcharts of up to 50 nodes without visible performance degradation in the browser.

6. CONCLUSION

This work successfully developed VisuCode, a web-based Python code execution visualizer that transforms Python source code into interactive Mermaid.js flowcharts using AST-based simulation. The system provides an accessible, browser-based tool that requires no local Python installation and supports a comprehensive range of Python constructs including assignments, conditionals, loops, functions, and classes.

VisuCode demonstrates that Python's AST module can be effectively leveraged to simulate program execution flow and generate accurate control-flow diagrams without running the

https://doi.org/10.5281/zenodo.19627590

code in a live interpreter. This makes the system safe, fast, and suitable for educational environments where sandboxed code execution may not be available.

The integration of a Flask REST API with a lightweight HTML/CSS/JS frontend ensures that VisuCode is easy to deploy and use. Future work will extend the system to support additional Python constructs such as list comprehensions, exception handling (try/except), lambda functions, and nested functions, as well as step-by-step animation of the flowchart execution.

REFERENCES

1. P. J. Guo, "Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education," ACM SIGCSE Technical Symposium, 2013.

2. J. Maloney et al., "The Scratch Programming Language and Environment," ACM Transactions on Computing Education, vol. 10, no. 4, 2010.

3. N. Fraser, "Blockly: A Visual Programming Editor," Google Open Source Blog, 2012.

4. P. Carlisle, "Flowgorithm: Flowchart Programming Language," Sacramento State University, 2011.

5. K. Nakakoji et al., "Software Visualization: Design, Implementation, Deployment," IEEE Software, 2003.

6. C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A Meta-Study of Algorithm Visualization Effectiveness," Journal of Visual Languages and Computing, vol. 13, no. 3, pp. 259-290, 2002.

7. Python Software Foundation, "ast — Abstract Syntax Trees," Python Documentation, 2024.

8. Pallets Projects, "Flask Documentation," https://flask.palletsprojects.com, 2024.

9. Mermaid.js Contributors, "Mermaid: Generation of Diagram and Flowchart," https://mermaid.js.org, 2024.

10. M. Ala-Mutka, "A Survey of Automated Assessment Approaches for Programming Assignments," Computer Science Education, vol. 15, no. 2, 2005.

11. T. Ben-Ari, "Constructivism in Computer Science Education," Journal of Computers in Mathematics and Science Teaching, vol. 20, no. 1, 2001.

12. D. Perkins, "The Many Faces of Constructivism," Educational Leadership, vol. 57, no. 3, pp. 6-11, 1999.

https://doi.org/10.5281/zenodo.19627590