# A Comparative Study of Control Flow Graph in Python

**Areesha Imtiaz**, Department of Computer Science
**Faculty Mentor:** Dr. Salam Salloum
RSCA Conference 2026

## Abstract

Control Flow Graphs (CFGs) are fundamental structures in program analysis, software testing, and compiler design, representing the flow of execution within a program. This study presents a comparative evaluation of multiple approaches for constructing CFGs in Python, including AST-based, bytecode-based, and token/indentation heuristic-based methods. Based on structural reliability, visualization clarity, and suitability for source-level analysis, AST-based approaches were selected for deeper evaluation. Among the AST-based tools examined, PY2CFG was chosen due to its direct traversal strategy and interpretable graph output.

The selected framework was modernized and extended to improve compatibility with recent Python versions and to enhance control-flow modeling. Improvements included support for Python 3.10+ features (e.g., match/case), accurate handling of while/else and for/else constructs, enhanced exception modeling, and integration of McCabe's cyclomatic complexity computation. A case study demonstrates CFG generation from sample Python code and illustrates practical complexity analysis. The results highlight the effectiveness of AST-based CFG construction for structural program analysis in Python.
**Keyword(s):** Control Flow Graph (CFG), Abstract Syntax Tree (AST), Cyclomatic Complexity, Graph Visualization.

## Introduction

According to a 2018 survey conducted by "Stripe", around **40%** of an average engineers' time is spent on technical debt and bad code. This equates to around **300 billion dollars** annual opportunity cost. In the fast-paced world of software development, ensuring software quality is paramount. However, analyzing unknown source code to comprehend it is both difficult and expensive task. Visual representations have proven to be especially valuable in improving **Program Comprehension** (PC).

A **Control Flow Graph (CFG)** is a directed graph that represents the possible flow of execution through a program [3]. A **Node** represents the program's basic blocks (e.g., statements), and an **Edge** indicates the potential control flow between these blocks. CFGs have one entry block at the beginning and one exit block at the end of the instruction [4]. **Figure 1** shows a generic CFG for an `if-then-else` statement.

CFGs not only aid in program comprehension, but also make software easier to test, and development more efficient. Some noteworthy applications of this tool include software testing (e.g., fault localization), machine learning research (e.g., AI code optimization), static analysis, data flow analysis (e.g., cyclomatic complexity), advanced analysis (e.g., pattern/anomaly detection), etc [2][3][4].

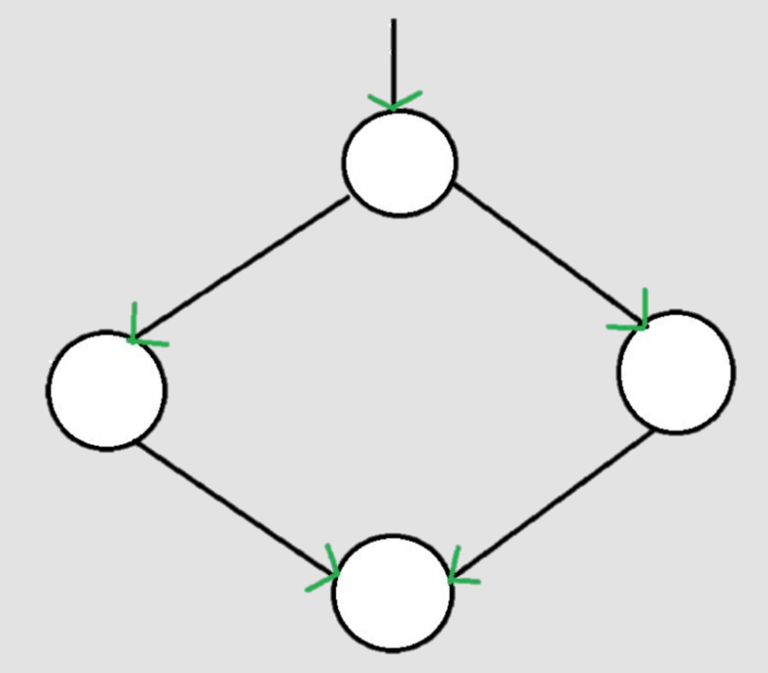

**Figure 1:** Generic CFG for an `if-then-else` statement [5]

## Materials

**Python** was selected as the primary language of this research due to its strong industry relevance. According to the TIOBE Index (February 2026), Python currently ranks as the most popular programming language.

After a thorough literature review of **25+** scholarly/non-scholarly articles, **5** distinct algorithms for constructing CFGs were identified:

- **AST-Based Algorithm(s)**
  - PY2CFG
  - STATICFG
  - Python-Graphs
- **Bytecode-Base Algorithm(s)**
  - EtherSolve
- **Token/Indentation Heuristic-Based Algorithm(s)**

## Methodology

*Algorithmic Overview*

**Abstract Syntax Tree** (AST)-**Based Algorithm** parses the source code and generates an AST. An AST is an ordered tree representation of the context-free grammar structure of the code, capturing both lexical (i.e., tokens) and syntactic information (refer to Figure 2). AST nodes are traversed, starting from the root node, to identify control flow. Basic blocks are created based on the analysis, and edges are added to connect the control constructs.

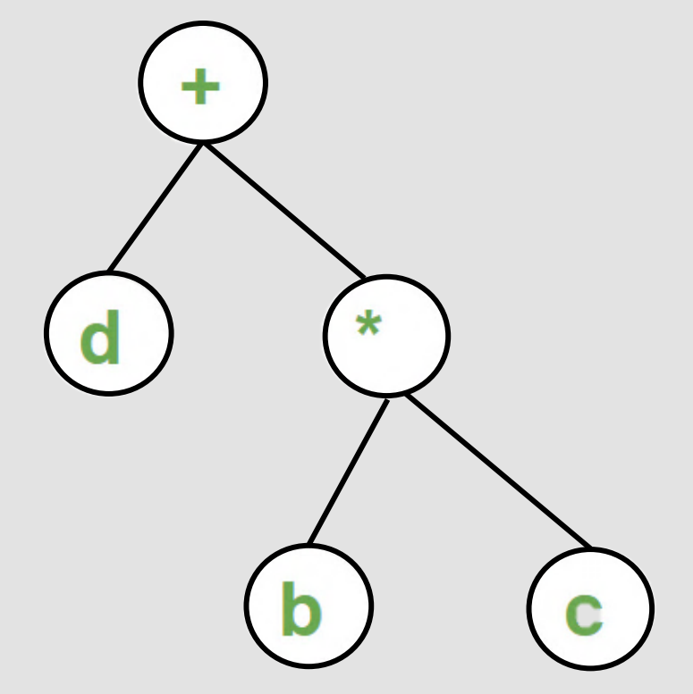

**Figure 2:** Abstract Syntax Tree for `d + (b * c)` [8]

**Bytecode-Based Algorithm** first compiles the source code into bytecode and then analyzes the low-level instructions to identify the basic blocks that constitute the CFG.

**Token/Indentation Heuristic-Based Algorithm** analyzes lexical tokens and indentation structure to heuristically approximate control blocks.

*Comparative Evaluation Criteria*

Among the methods discussed, bytecode-based algorithm proved to be the fastest. Nevertheless, AST-based approaches are structurally more reliable and better suited for debugging and visualization. Moreover, AST parsing allows partial passing even with minor syntactic issues [1]. Therefore, AST-based algorithms were shortlisted for further evaluation.

*Tool Selection Process*

Although 3 AST-based tools were initially identified, STATICFG was excluded from the discussion since PY2CFG is a significantly re-worked and improved fork of it [7]. Python-Graphs incorporates advanced static analysis techniques whereas PY2CFG performs direct AST traversal, producing a more visually interpretable graph. For the purposes of this study, PY2CFG was selected.

## Results

*Improvements & Enhancements*

- Tested diverse control structures (`if`, `while`, `for/else`, `try/except/finally`, `raise`, `match/case`).
- Modernized deprecated AST components (`ast.Constant`, `ast.unparse`).
- Added support for Python 3.10+ `match/case`.
- Corrected loop and exception flow handling.
- Implemented full support for `while/else` and `for/else` control flow constructs.
- Introduced labeled exception edges (`enter-try`, `exc:TypeError`, `no-exception`).
- Preserved entry blocks during CFG cleanup.
- Improved graph readability and edge labeling.
- Added McCabe-based cyclomatic complexity computation.

*Complexity*

Cyclomatic complexity measures how many different execution paths exist within a program. It is computed directly from the program's CFG.

A commonly used method for computing cyclomatic complexity is **McCabe's Method** [3], defined as:

$$M = E - N + 2P$$

where:

$E$ = edges in the CFG
$N$ = nodes in the CFG
$P$ = distinct connected components

*Case Study: CFG Generation and Cyclomatic Complexity*

The following example (Figure 3) demonstrates the generation of a Control Flow Graph (CFG) from Python source code using the enhanced framework. The resulting graph is used to compute cyclomatic complexity, illustrating the practical application of CFG-based analysis.

**Code:**

```
x = 1
if x > 0:
      i = 0
else:
      i = 3
while i < 2:
      i += 1
```

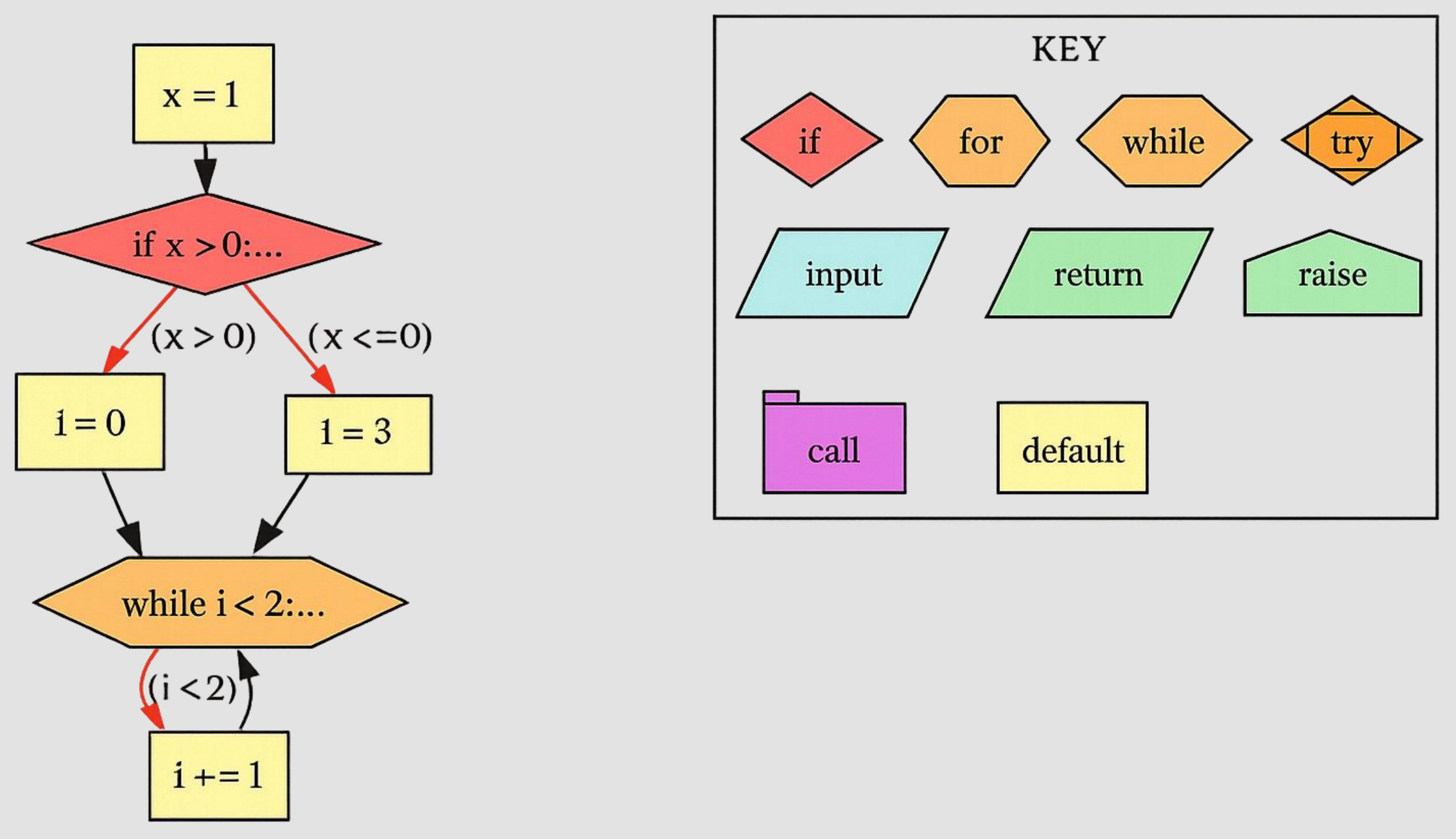

**Figure 3:** Control Flow Graph for the sample code

**Cyclomatic Complexity:**

```
Overall McCabe Complexity: 3
Breakdown:
      main: 3
      main (total): 3
```

## Conclusion

This study demonstrates that AST-based approaches provide a reliable and structurally clear method for constructing Control Flow Graphs in Python. Through comparative evaluation, PY2CFG was selected and subsequently enhanced to improve compatibility, correctness, and interpretability. The implemented extensions strengthened loop and exception modeling, introduced support for modern Python language features, and improved overall graph readability.

By integrating cyclomatic complexity computation using McCabe's method, the work connects theoretical CFG construction to practical software testing and analysis applications. The case study validates the correctness of the enhanced framework and illustrates how CFGs can be used to quantify program complexity. Future work may extend this approach toward interprocedural analysis, automated testing integration, and advanced static analysis applications.

## References

[1] Salling, Jackson Lee. *Control flow graph visualization and its application to coverage and fault localization in Python.* Diss. 2015.
[2] Srivastava, Akhilesh Kumar, Rijwan Khan, and Sneha Jain. "A tool for generation of automatic control flow graph in unit testing of python programs." *Int J Eng Adv Technol (IJEAT)* 8.5 (2019): 1178-1184.
[3] Bieber, David, et al. "A library for representing python programs as graphs for machine learning." *arXiv preprint arXiv:2208.07461* (2022).
[4] Samodra, Bayu, et al. "Development of Graph Generation Tools for Python Function Code Analysis." *JITK (Jurnal Ilmu Pengetahuan dan Teknologi Komputer)* 10.3 (2025): 690-697.
[5] GeeksforGeeks. "Control Flow Graph (CFG) Software Engineering." *GeeksforGeeks*, 15 May 2019, www.geeksforgeeks.org/software-engineering/software-engineering-control-flow-graph-cfg/.
[6] GeeksforGeeks. "Compiler Design Variants of Syntax Tree." *GeeksforGeeks*, 15 Feb. 2022, www.geeksforgeeks.org/compiler-design/compiler-design-variants-of-syntax-tree/.
[7] "Py2cfg — Py2cfg 0.5.1 Documentation." Readthedocs.io, 2020, py2cfg.readthedocs.io/en/latest/. Accessed 25 Feb. 2026.

**The full project repository, including implementation details and development progress, is available at:**
https://github.com/areeshaimtiaz/CS-4610

## Acknowledgements

This project was completed as part of **CS 4610** Senior Project I, in the Department of Computer Science at Cal Poly Pomona. I thank **Dr. Salam Salloum** for his mentorship and guidance. This work also fulfills the CPP Kellogg Honors College Engaged Learning Experience requirement and will be presented at **CPP RSCA 2026**.