SimpliPy: A Source-Tracking Notional Machine for Simplified Python

Moida Praneeth Jain[0009−0006−8192−6041] and Venkatesh Choppella[0000−0003−1085−3464]

International Institute of Information Technology Hyderabad, Telangana, India
moida.praneeth@students.iiit.ac.in, venkatesh.choppella@iiit.ac.in

Abstract. Misconceptions about program execution hinder many novice programmers. We introduce SimpliPy, a notional machine designed around a carefully chosen Python subset to clarify core control flow and scoping concepts. Its foundation is a precise operational semantics that explicitly tracks source code line numbers for each execution step, making the link between code and behavior unambiguous. Complementing the dynamic semantics, SimpliPy uses static analysis to generate Control Flow Graphs (CFGs) and identify lexical scopes, helping students build a structural understanding before tracing. We also present an interactive web-based debugger built on these principles. This tool embodies the formal techniques, visualizing the operational state (environments, stack) and using the static CFG to animate control flow directly on the graph during step-by-step execution. SimpliPy thus integrates formal semantics, program analysis, and visualization to offer both a pedagogical approach and a practical demonstration of applying formal methods to program understanding.

Keywords: Operational Semantics · Static Analysis · Program Visualization · Programming Education · Python.

1 Motivation and Introduction

1.1 Misconceptions and Difficulties in Learning Programming

Learning programming encompasses the acquisition of a multitude of related skills: reading programs, tracing their execution, debugging, writing test cases, documenting and writing code. Acquiring these skills by students is hampered by the many misconceptions they harbour about how programs are structured and how they run [11,12,15,23].

In this paper, we are motivated by the larger problem of developing a pedagogy of teaching how programs run. Students construct a variety of mental models and often represent them as 'sketches' [4] when tasked with the problem of tracing code. These sketches reflect the thought processes and mental models that students construct to express their understanding of program execution.

1.2 Syntactic vs. Semantic Approach

In the teaching of programming, we are seeing a shift in pedagogy that increasingly emphasizes a semantic, rather than a syntactic approach to understanding programs [10,22]. The semantic view emphasizes program comprehension [18], encourages the construction of mental models that reflect an accurate understanding of how a program runs on a *notional machine* [3]. Understanding the notional machine and its dynamics then becomes central to understanding how any program in a programming language runs. Indeed, it has been argued that teaching the notional machine ought to be an explicit objective in introductory programming education [22]. An accurate definition of a notional machine necessarily borrows from the theory of programming language semantics. The effective teaching and learning of programming is therefore intimately intertwined with the knowledge of the semantics of the programming language of the program [20].

1.3 Purpose and specificity of a notional machine

Knowledge of a programming language spans multiple dimensions, each of which needs to be understood to gain mastery of the language. The *control* dimension specifies how the program's flow moves across different program locations. The *scope* dimension specifies how variables are looked up. The *datatype* dimension specifies how data is defined. The *storage* dimension specifies how data is laid out in memory and shared. When designing notional machines, one needs to consider which knowledge dimensions of the language are being modeled [10]. A notional machine that tries to address all knowledge dimensions might end up being too complicated for the student to follow. It therefore typically focuses on one or more, but not necessarily all knowledge dimensions [7,22]. Its purpose then becomes the need to clearly elucidate the program's behaviour along the collection of dimensions.

A pedagogy based on semantics should encourage constructing well-defined artefacts around the program's structure that also approximate its putative runtime behaviour. These artefacts should be unambiguously interpretable to relate to the program's actual behaviour along a given set of knowledge dimensions. The artefacts could be diagrams, formula, or even plain text. Examples of such artefacts are stack diagrams for procedure calls, box and pointer diagrams for structure sharing, labelled directed graphs for control flow [7].

1.4 SimpliPy

Towards this end, we introduce **SimpliPy**, a new notional machine designed specially to study control flow and scoping in Python. The SimpliPy notional machine operates on a subset of Python, suitably chosen to make tracing easier, while retaining the essential control features of the language.

Python has wide usage as the programming language of choice for introductory programming in many parts of the world [16]. Therefore, our focus on Python seems well justified.

The SimpliPy semantics is precise, yet accessible after some careful practice. SimpliPy exploits the line-oriented structure of Python, enabling precise source tracking, meaning it can trace the origin of every transition back to a specific program location in the source code. This level of detail allows students to connect each operational step directly to the Python code they see, reinforcing their understanding of how specific lines influence program behavior.

SimpliPy assumes students of programming are familiar with the elementary concepts of sets, relations, (partial) functions, and sequences, usually taught in high-school.

2 Related Work

We present a brief tour of related work and connect it with the SimpliPy pedagogy. The concept of a notional machine goes back to over forty years when Boulay et al. [3] introduced the idea of a notional machine as "an idealised, conceptual computer whose properties are implied by the constructs in the programming language employed". Sorva [22] emphasizes that a notional machine should be explicitly studied as part of programming education. The SimpliPy approach embraces this pedagogical principle.

Various notional machines, their evolution and uses have been explored by the ITiCSE 2020 Working Group on Notional Machines [7]. However, the emphasis of these notional machines discussed is visual and informal. The SimpliPy notional machine, on the other hand, is based on a precise mathematical semantics of a well-defined subset of Python. Understanding the SimpliPy semantics, however, only demands a facility with elementary set theory. SimpliPy semantics can be stated (a bit more verbosely) in natural language. This makes it accessible to beginning students of programming.

Dickson et al. [5] argue that the complexity of notional machines should increase as the teaching goes from the early to advanced stages. The various types of control flow in Python: sequential, conditional, iterative and procedural are dealt with in an incremental way in SimpliPy. Nelson and co-workers [18,24] advocate a "comprehension first" approach to programming. They emphasise that understanding the structure and dynamics of a program should be preceded by first introducing an appropriate ontology of concepts. In their work, the semantics of a program is traced to the control flow path in the code of the underlying interpreter.

SimpliPy introduces its notional machine using the terminology of systems thinking (observables, states, actions, transitions) and it introduces the key notion of closures, crucial to understanding functions.

The SimpliPy approach shares the spirit of Nelson et al. [18] in using interpreter rules to define program flow, but differs in three key aspects. First, SimpliPy's notional machine instructions directly correspond to the program's source code instructions, allowing students to apply the machine's rules by interpreting the code they see. Second, the SimpliPy pedagogy encourages initial understanding through paper-and-pencil exercises – constructing execution traces and state

diagrams by hand leverages the cognitive benefits of slower, deliberate engagement [17]. This manual process is then actively supported and reinforced by an interactive visualization tool, which dynamically displays the same execution states (environments, stack) and control flow transitions, allowing students to verify their understanding and explore execution step-by-step. Third, SimpliPy promotes a comprehension-first strategy by requiring students to construct intermediate artifacts derived from static analysis before tracing a program. These artifacts include scope-related information (lexical blocks, variable declarations) and control information (control transfer functions, the program's control flow graph), which are also visualized by our tool to solidify the link between static program structure and dynamic behavior.

Notional machines developed for Python in the past have been either informal and incomplete [6] or have addressed only very specific issues like mutation and sharing [11] or have focused on identifying a subset for better error messages [2]. Lu and Krishnamurthi [15] present an online platform called SMoL Tutor. It comes with a web application called Stacker that steps through a program's execution. SMoL handles multiple languages, including Python. However, it requires that code be edited in a Lisp-style syntax and precludes experimenting with scoping mechanisms unique to Python, like `nonlocal`, `global` and function-only scope. On the other hand, in SimpliPy, all functions need to be named (the `lambda` keyword is not considered, but this is not a serious limitation because a fresh name can always be introduced).

Existing formal semantics for Python lack source tracking, limiting their utility in educational contexts. Smeding [21] models Python 2.5 as a transition system and does track instructions, but the approach omits some scoping details (like the global keyword) and is outdated for modern Python versions. Politz et al. [19] rewrite Python to a scheme-like core language, which introduces scoping inaccuracies with classes, while Guth [9] and Köhl [14] employ rewrite-based models that do not directly follow Python source lines. These limitations contrast with SimpliPy, which preserves source-level tracking, making it a more effective tool for teaching Python's execution flow.

3 SimpliPy Overview

In this section, we provide a comprehensive overview of the SimpliPy notional machine, detailing its language subset, the structure of the programs, the static analysis artefacts it utilizes, the notional machine's state, and the dynamics governing its execution.

3.1 Language Subset and Program Structure

SimpliPy operates on a carefully curated subset of Python designed for teaching control flow and scoping. It follows Python's line-oriented structure but introduces specific constraints for pedagogical purposes.

Main Syntactic Entities

- **Expressions (`Exp`):** Simple computations including constants, variables and operations using unary or binary operators (`Uop`, `Bop`). Crucially, expressions in SimpliPy are designed to be simple and *do not* contain function calls.
- **Instructions (`Instr`):** The basic syntactic unit corresponding to a single line of Python code within the SimpliPy subset.
- **Statements (`Stmt`):** Represent complete logical steps or control structures. A simple statement might correspond directly to a single instruction (like `pass` or `ExpAssign`). Compound statements (`If`, `While`, `Def`) encompass multiple instructions, including nested blocks.
- **Blocks (`Blk`):** A sequential composition of one or more statements. Blocks may define scopes and group related statements, such as the body of an `if`, `while`, or `def`.
- **Program (`Pgm`):** The entire SimpliPy program, defined as a single top-level block.

A SimpliPy program P of length N (containing N instructions across all lines) can be viewed as a mapping from locations (line numbers) to instructions:

$$P : [1..N] \to \texttt{Instr}$$

We define the set of program locations as $L = [1..N+1]$, where $N+1$ represents a conceptual end-of-program location.

The precise syntactic structure allowed in SimpliPy is formally defined by a BNF grammar, which is available in our supplementary materials repository: https://github.com/PraneethJain/simplipy.

3.2 Static Analysis

An important aspect of the pedagogy of SimpliPy is the analysis of a program before it is run. This involves two types of analyses: scope and control, each producing multiple artefacts. Scope analysis defines the program's lexical blocks, identifies *locals* (the set of variables declared within each lexical block, excluding nonlocal and global variables), *nonlocals* and *globals* (variables designated as nonlocal or global within each block). Control flow analysis generates three primary artifacts: a structural abstraction of the program where each statement is represented by its syntactic category, a set of control transfer functions utilized by the notional machine's transition function, and a control flow graph that statically approximates the trace of program locations during execution. Students are expected to construct these artefacts as part of exercises in program comprehension before they build the trace of the program.

The control transfer functions are partial functions that map a location to another location. SimpliPy consists of four control transfer functions: *next*, *true*, *false*, and *err*. The functions *true* and *false* are specifically associated with if and while statements, determining the control flow based on the evaluated condition. The *err* function is defined universally for all locations. *next* facilitates

sequential execution, transitioning to the location corresponding to the subsequent statement in the program, while *err* enables handling of erroneous cases, mapping a location to itself, to indicate a fixed point.

Additionally, *next* is configured to manage control flow for break and continue statements appropriately. For instance, break within a while loop redirects control to the location mapped by the *false* function of the while statement, thereby exiting the loop. Conversely, continue within a while loop directs control back to the location of the while statement itself, enabling a re-evaluation of the loop condition.

3.3 Notional Machine Structure

$$(e, h, k) \in State$$
$$State = LexicalMap \times LexicalHierarchy \times Continuation$$

The notional machine is a labelled transition system whose state space is defined by the triple consisting of the following: the lexical map e, the lexical hierarchy h, and the continuation k.

- **Lexical Map (e):** This component holds all the environments created during execution. It functions as a map from unique environment identifiers (integers, with 0 representing the global environment) to environments. An environment itself is a dictionary mapping variable names (identifiers) within that scope to their currently assigned values. Values can include primitive types (numbers, booleans, strings), special markers (like an uninitialized 'bottom' value, denoted $\perp$), or closures representing function definitions. The lexical map is essentially an abstraction of the program's memory for variable bindings across all scopes.
- **Lexical Hierarchy (h):** This structure captures the well founded relation on the environments. It maps each environment's unique identifier (except the root) to the identifier of its immediate parent environment. The global environment (Id 0) serves as the root of this tree and has no parent. This hierarchy is fundamental to understanding Python's lexical scoping: when resolving a variable name, a typical search begins in the current environment and proceeds upwards towards the root by following the parent links defined in h until the variable is found or the global scope is checked. New branches in the tree are created during function calls, linking the newly created function's environment to the environment where the function was lexically defined.
- **Continuation (k):** This component is an abstraction of the control state, acting as the machine's execution stack. It comprises a list of *contexts*. A context is defined as a tuple (location, environment_id), indicating the instruction to be executed (location) and the environment (environment_id) required for its execution. The context at the head of the list (top of the stack) determines the immediate step. Function calls introduce a new context onto the list, preserving the caller's return point (location of the call, caller's environment Id). Function returns remove the current context, effectively

transferring control back according to the previously preserved context. In this way, the continuation directs the control flow through procedure calls and returns.

Initially, the notional machine is configured as follows: The lexical map e contains a single entry for the global environment, identified by Id 0. The lexical hierarchy h contains only the global environment Id 0, which has no parent. The continuation k begins with one context on its stack, pointing to the location of the program's first instruction and the global environment Id 0.

3.4 Notional Machine Dynamics

The notional machine is parametrized by the program P, which maps locations to instructions. The actions on the notional machine are program instructions. The dynamics of the notional machine is specified by a transition relation

$$(e, h, k) \xrightarrow{c} (e', h', k')$$

where $c = P_i$ is the instruction at the current location i from the top context.

Variable Lookup Mechanism Before detailing the state transitions, it is essential to understand how SimpliPy resolves variable names, adhering to Python's lexical scoping rules. When the value of a variable `var` is needed within the current execution context (defined by environment `env_id`), the lookup proceeds as follows:

1. **Current Environment:** The environment identified by `env_id` is checked first. If `var` is found here, its value is returned.
2. **Lexical Ancestors:** If `var` is not in the current environment, the search continues recursively in the parent environment, determined by consulting the lexical hierarchy (h). This process follows the chain of parent links upwards from the current environment.
3. **Global Environment:** If the search reaches the global environment (Id 0) without finding `var` locally or in any ancestor scope, the global environment is checked.
4. **Nonlocal/Global Directives:** The static analysis identifying `nonlocal` and `global` variables within lexical blocks influences this search. If a variable is marked `nonlocal`, the search explicitly skips the immediate local environment and the global environment, and starts in the parent environment. If marked `global` (or if it's accessed in the top-level scope), the search targets the global environment directly or proceeds there if not found locally.
5. **Lookup Failure:** If the variable is not found after checking all relevant lexical ancestors, a lookup error occurs.

The update mechanism for assignments similarly uses this lookup process to find the correct environment where an existing variable binding should be

updated, or determines that a new binding needs to be created (the global scope for top-level assignments or variables declared `global`).

The precise mathematical rules defining the transition relations for each instruction type are detailed in the formal operational semantics, available in our supplementary materials repository: https://github.com/PraneethJain/simplipy. We briefly describe the conceptual behavior for each instruction type below.

Pass, Global, Nonlocal: These instructions primarily serve structural or declarative purposes. `Pass` performs no operation. `Global` and `Nonlocal` affect the static analysis and subsequent variable lookups but do not modify the machine state during their own execution step. Control simply proceeds to the next instruction determined by the `next` control transfer function.

Break, Continue: These alter control flow within loops. `Break` transfers control to the instruction immediately following the innermost enclosing `while` loop. `Continue` transfers control back to the condition check (the `while` instruction itself) of the innermost enclosing loop. The `next` control transfer function determines this behaviour.

Expression Assignment: The expression on the right-hand side is evaluated using the current environment context. If successful, the variable on the left-hand side is updated with the resulting value in the appropriate environment (determined by the lookup mechanism). Control then moves to the next instruction. Evaluation errors lead to an error transition.

If / While: The conditional expression is evaluated. Based on whether the result is true or false, control transfers to the location specified by the corresponding `true` or `false` control transfer function (entering the appropriate block for `if`, entering or exiting the loop for `while`). Non-boolean results or evaluation errors lead to an error transition.

Function Definition: A closure value is created, capturing the starting location of the function's code block, the list of formal parameter names, and a reference to the current lexical environment (its Id). This closure is then bound to the function's name in the current environment. Control proceeds to the next instruction.

Call Assignment: The expression identifying the function is evaluated to retrieve a closure. The argument expressions are evaluated. If successful and type/arity checks pass, a new environment is created for the function call. Formal parameters are bound to the evaluated argument values in this new environment, and other local variables declared within the function are initialized to $\perp$. The new environment is linked to its parent (based on the closure's captured environment Id) in the lexical hierarchy (h). The context for the function's entry point and the new environment Id is pushed onto the continuation k. Control transfers to the function's entry point. Errors during lookup or evaluation lead to an error transition.

Return: The return expression is evaluated in the function's current environment. The top context is popped from the continuation stack (the function's current context). The variable targeted by the original call assignment instruction (found at the top of the stack) is updated in the caller's environment with the evaluated return value. Execution resumes at the instruction

following the call site in the caller's context. Evaluation errors lead to an error transition.

An execution trace consists of the sequence of states generated by repeatedly applying these transition rules, starting from the initial state, until a final state (fixed point) is achieved.

4 SimpliPy Visualization Tool

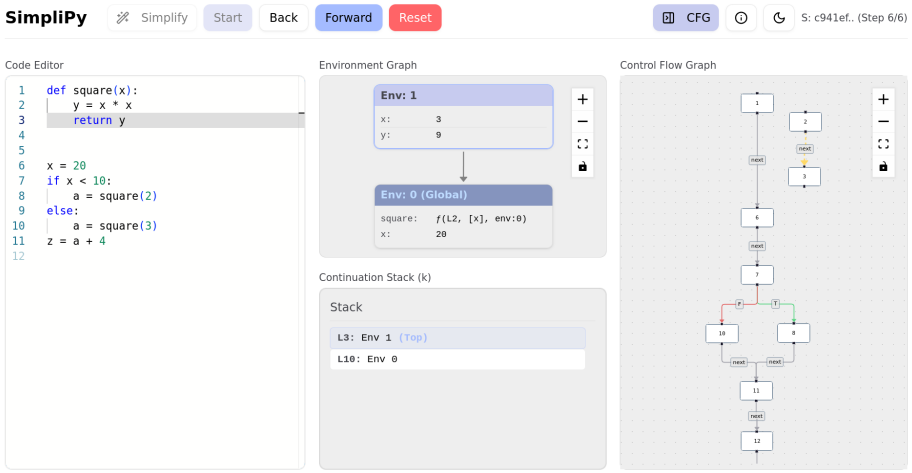

Fig. 1. The SimpliPy interactive visualization tool interface. Coordinated views display the program state according to the SimpliPy notional machine: (Left) Source code with the current instruction highlighted. (Top Middle) The Environment Scope View visualizes the lexical map (e) and hierarchy (h) as a tree, showing active environments and their variable bindings. (Bottom Middle) The Continuation Stack (k) displays active execution contexts ($location, env\_id$). (Right) The Control Flow Graph (CFG) highlights the edge representing the most recent control transfer. Debugging controls at the top allow simplifying code, starting/resetting execution, stepping forward/backward, and toggling the CFG view.

To operationalize the SimpliPy notional machine and support student learning, we developed an interactive web-based visualization tool, publicly accessible at https://simplipy.web.app/. This tool serves as a dynamic complement to manual program tracing, allowing students to observe the step-by-step execution according to the defined semantics, verify their understanding, and explore the relationship between code, state, and control flow. The design prioritizes pedagogical clarity and faithful semantic representation over the feature breadth of production development tools.

4.1 Pedagogical Features

The SimpliPy tool presents coordinated views designed to showcase concepts central to the notional machine:

- **Source Code View:** Displays the Python code, persistently highlighting the current line number indicated by the top of the continuation stack. This directly links the machine's current focus to the corresponding source instruction.
- **Environment Scope View:** A key feature that explicitly visualizes the relationship between different scopes (e and h) active during execution, rendered as a tree. Each node represents an environment and displays the variables currently bound within that specific scope along with their values (including closures or uninitialized markers $\perp$). This makes Python's scoping rules tangible, allowing students to see exactly which environment holds a given variable and understand the lookup process.
- **Continuation Stack View:** Displays the current continuation (k), showing the sequence of active execution contexts `(location, env_id)`. This clarifies procedural control flow, showing pending return points and their associated environments.
- **Control Flow Graph View:** Integrates the static control flow graph. During execution stepping, the tool highlights the edge representing the control transfer just taken (e.g., 'next', 'true', 'false'), visually connecting the dynamic execution path to the program's static structure. The 'err' control transfer function is omitted from this view to reduce clutter.
- **Step-by-Step Execution Control:** Enables forward execution one instruction at a time, following the SimpliPy transition rules precisely. A history mechanism allows stepping backward to review previous states.
- **Simplification Support:** An integrated feature attempts to automatically convert standard Python code into the SimpliPy subset, lowering the barrier for using the tool with existing examples.

4.2 Comparison with Other Tools

The SimpliPy tool occupies a specific pedagogical niche, differing significantly from both production debuggers and other educational or analysis tools, primarily through its chosen level of abstraction and its direct embodiment of a formal notional machine.

Compared to production debuggers found in environments like VS Code or PyCharm, SimpliPy offers distinct pedagogical advantages via deliberate simplification. Production tools, designed for development efficiency, often present complex interfaces and expose low-level implementation details (e.g., specific stack frame layouts, memory addresses) that can distract learners from core language semantics. Their typical "Locals" view often flattens the representation of nested scopes, obscuring the hierarchical nature of lexical environments crucial for understanding variable lookup rules and lifetimes. While powerful for

debugging with features like breakpoints and live value modification, their focus is not on illustrating the fundamental semantic steps in a clear, abstracted manner. SimpliPy also allows stepping backward through the execution history to review previous states and transitions, a feature generally absent in the standard execution modes of production debuggers, which facilitates learning and exploration of cause-and-effect in program execution. SimpliPy intentionally abstracts away memory implementation specifics, focusing instead on a faithful, observable step-by-step execution of its formal semantics, prioritizing conceptual clarity.

SimpliPy also distinguishes itself from other educational and analysis tools. While Python Tutor [8] provides excellent visualizations of program state including memory layout (stack/heap), SimpliPy deliberately abstracts these aspects to emphasize the lexical environment hierarchy. Beginner-focused IDEs like Thonny [1] offer a simplified debugging experience but operate closer to standard Python execution. Tools like Localizer [13], on the other hand, focus on fault localization in full Python code using dynamic analysis of test results and visualizing Control Flow Graphs (CFGs) annotated with suspiciousness scores. In contrast to these tools, SimpliPy's unique contribution lies in its specific focus on visualizing its formal operational semantics for a simplified language subset—particularly scope structure (via the environment tree) and control flow (via the continuation stack and CFG path highlighting)—at a high level of abstraction, rigorously embodying its notional machine and supporting both forward and backward stepping to maximize learning impact. Table 1 provides a comparative summary of these features.

Table 1. Pedagogical Feature Comparison: SimpliPy vs. Other Tools

Feature	SimpliPy Tool	Localizer [13]	Python Tutor [8]	Thonny [1]	VS Code / PyCharm
Scope Structure	Yes (Tree)	No	Limited (Stack/Heap)	Simplified Locals	No (Flattened)
Language Subset	Restricted	Full Language	Broad Subset	Full Language	Full Language
CFG	Yes	Yes	No	No	No
Variable Modification	No	No	No	Yes	Yes
Breakpoints	No	No	Limited	Yes	Yes
Primary Goal	Visualize Semantics	Fault Localization	Visualize State/Refs	Beginner Debugging	Production Debugging

5 Limitations and Future Work

While the SimpliPy notional machine and its associated visualization tool provide a focused pedagogical foundation for understanding Python's control flow and scope, certain limitations exist in the current implementation, suggesting avenues for future development.

One key limitation is the restricted language subset, omitting features like exceptions, comprehensions, modules, asynchronous programming, and objects. Future work involves carefully extending the semantics for such features, balancing coverage with pedagogical clarity.

The visualization tool, while effective for observation, currently lacks interactive debugging capabilities common in production environments, such as setting breakpoints or modifying variable values during execution. Future work could explore adding limited, pedagogically motivated interactive features, alongside enhanced visualizations focusing on data flow.

Finally, rigorous empirical studies are required to formally evaluate the effectiveness of the SimpliPy approach. Future work includes conducting controlled user studies with introductory programming students to compare learning outcomes (particularly regarding mental model accuracy and misconception reduction) when using SimpliPy (both manual tracing and the tool) versus traditional teaching methods or other visualization tools.

6 Conclusion

SimpliPy presents a formally grounded approach to teaching fundamental aspects of Python program execution, specifically control flow and lexical scoping. By defining a precise operational semantics for a carefully curated language subset with explicit source-tracking, integrating static analysis artifacts like Control Flow Graphs and lexical scope information, and providing an interactive visualization tool, it aims to help novice programmers build more accurate and robust mental models. The emphasis on comprehension prior to tracing, facilitated by static analysis, and the subsequent reinforcement via the visualization tool which faithfully renders the notional machine's state (environments, lexical hierarchy, continuation) and control flow, offers a distinct pedagogical pathway.

The SimpliPy tool, in particular, demonstrates the value of applying formal methods in an educational context, providing a level of abstraction over low-level memory details (stack frames, heap) that allows learners to focus directly on semantic rules. While intentionally simpler than production debuggers or broader educational visualizers, its strength lies in this focused, semantic-driven visualization. SimpliPy provides a solid foundation for both teaching core programming concepts and further research into semantics-based programming education.

The supplementary materials repository, available at https://github.com/PraneethJain/simplipy, contains the detailed resources referenced in this paper, including the formal BNF grammar, the complete operational semantics transition rules, and the implementation of the SimpliPy visualization tool.

References

1. Annamaa, A.: Thonny, a python ide for learning programming. In: Proceedings of the 15th Koli Calling International Conference on Computing Education Research (Koli Calling '15). pp. 115–119. Koli Calling '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2828959.2828969, https://doi.org/10.1145/2828959.2828969
2. Aycock, J.: spy3: A python subset for cs1. In: Proceedings of the 25th Western Canadian Conference on Computing Education. WCCCE '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3593342.3593352, https://doi.org/10.1145/3593342.3593352
3. Boulay, B.d., O'Shea, T., Monk, J.: The black box inside the glass box: presenting computing concepts to novices. International Journal of Man-Machine Studies **14**(3), 237–249 (Apr 1981). https://doi.org/10.1016/S0020-7373(81)80056-9, https://www.sciencedirect.com/science/article/pii/S0020737381800569
4. Cunningham, K., Blanchard, S., Ericson, B., Guzdial, M.: Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw. In: Proceedings of the 2017 ACM Conference on International Computing Education Research. pp. 164–172. ICER '17, Association for Computing Machinery, New York, NY, USA (Aug 2017). https://doi.org/10.1145/3105726.3106190, https://dl.acm.org/doi/10.1145/3105726.3106190
5. Dickson, P.E., Brown, N.C.C., Becker, B.A.: Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices. In: Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education. pp. 159–165. ITiCSE '20, Association for Computing Machinery, New York, NY, USA (Jun 2020). https://doi.org/10.1145/3341525.3387404, https://dl.acm.org/doi/10.1145/3341525.3387404
6. Dickson, P.E., Richards, T., Becker, B.A.: Experiences Implementing and Utilizing a Notional Machine in the Classroom. In: Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1. SIGCSE 2022, vol. 1, pp. 850–856. Association for Computing Machinery, New York, NY, USA (Feb 2022). https://doi.org/10.1145/3478431.3499320, https://dl.acm.org/doi/10.1145/3478431.3499320
7. Fincher, S., Jeuring, J., Miller, C.S., Donaldson, P., du Boulay, B., Hauswirth, M., Hellas, A., Hermans, F., Lewis, C., Mühling, A., Pearce, J.L., Petersen, A.: Notional Machines in Computing Education: The Education of Attention. In: Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education. pp. 21–50. ITiCSE-WGR '20, Association for Computing Machinery, New York, NY, USA (Dec 2020). https://doi.org/10.1145/3437800.3439202, https://doi.org/10.1145/3437800.3439202
8. Guo, P.J.: Online python tutor: Embeddable web-based program visualization for CS education. In: Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13). pp. 579–584. SIGCSE '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2445196.2445368, https://doi.org/10.1145/2445196.2445368
9. Guth, D.: A formal semantics of Python 3.3. Master's thesis, University of Illinois (2013)
10. Guzdial, M., Krishnamurthi, S., Sorva, J., Vahrenhold, J.: Notional Machines and Programming Language Semantics in Education (Dagstuhl Seminar 19281). DROPS-IDN/v2/document/10.4230/DagRep.9.7.1 (2019). https://

doi.org/10.4230/DagRep.9.7.1, https://drops.dagstuhl.de/entities/document/10.4230/DagRep.9.7.1, publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik
11. Johnson, F., McQuistin, S., O'Donnell, J.: Analysis of student misconceptions using python as an introductory programming language. In: Proceedings of the 4th Conference on Computing Education Practice. CEP '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372356.3372360, https://doi.org/10.1145/3372356.3372360
12. Kaczmarczyk, L.C., Petrick, E.R., East, J.P., Herman, G.L.: Identifying student misconceptions of programming. In: Proceedings of the 41st ACM Technical Symposium on Computer Science Education. pp. 107–111. SIGCSE '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1734263.1734299, https://doi.org/10.1145/1734263.1734299
13. Khan, S., Sudheerbabu, G., Truscan, D., Ahmad, T.: Localizer: A visual debugging assistant for python programs. In: Proceedings of the 2nd ACM International Workshop on Future Debugging Techniques (DEBT '24). pp. 34–35. ACM, New York, NY, USA (Sep 2024). https://doi.org/10.1145/3678720.3685321, https://doi.org/10.1145/3678720.3685321, vienna, Austria, September 19, 2024
14. Köhl, M.A.: An Executable Structural Operational Formal Semantics for Python. Master's thesis, Saarland University (2021), https://arxiv.org/abs/2109.03139
15. Lu, K.C., Krishnamurthi, S.: Identifying and Correcting Programming Language Behavior Misconceptions. Proc. ACM Program. Lang. **8**(OOPSLA1), 106:334–106:361 (Apr 2024). https://doi.org/10.1145/3649823, https://doi.org/10.1145/3649823
16. Mason, R., Simon, Becker, B.A., Crick, T., Davenport, J.H.: A global survey of introductory programming courses. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1. p. 799–805. SIGCSE 2024, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3626252.3630761, https://doi.org/10.1145/3626252.3630761
17. Mueller, P.A., Oppenheimer, D.M.: The Pen Is Mightier Than the Keyboard: Advantages of Longhand Over Laptop Note Taking. Psychological science **25**(6), 1159–1168 (Jun 2014). https://doi.org/10.1177/0956797614524581, https://doi.org/10.1177/0956797614524581
18. Nelson, G.L., Xie, B., Ko, A.J.: Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In: Proceedings of the 2017 ACM Conference on International Computing Education Research. pp. 2–11. ICER '17, Association for Computing Machinery (Aug 2017). https://doi.org/10.1145/3105726.3106178, https://doi.org/10.1145/3105726.3106178
19. Politz, J.G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., Krishnamurthi, S.: Python: the full monty. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. p. 217–232. OOPSLA '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2509136.2509536, https://doi.org/10.1145/2509136.2509536
20. Reynolds, J.C.: Some thoughts on teaching programming and programming languages. SIGPLAN Not. **43**(11), 108–110 (Nov 2008). https://doi.org/10.1145/1480828.1480852, https://doi.org/10.1145/1480828.1480852
21. Smeding, G.: An executable operational semantics for Python. Master's thesis, University of Utrecht (01 2009)
22. Sorva, J.: Notional machines and introductory programming education. ACM Transactions on Computing Education **13**(2), 8:1–8:31 (Jul 2013). https://doi.org/10.1145/2483710.2483713, https://dl.acm.org/doi/10.1145/2483710.2483713

23. Vainio, V., Sajaniemi, J.: Factors in novice programmers' poor tracing skills. In: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education. pp. 236–240. ITiCSE '07, Association for Computing Machinery, New York, NY, USA (Jun 2007). https://doi.org/10.1145/1268784.1268853, https://dl.acm.org/doi/10.1145/1268784.1268853
24. Xie, B., Nelson, G.L., Ko, A.J.: An explicit strategy to scaffold novice program tracing. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education. pp. 344–349. SIGCSE '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3159450.3159527, https://doi.org/10.1145/3159450.3159527