Evaluating How Novices Utilize Debuggers and Code Execution to Understand Code

Mohammed Hassan
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
mhassan3@illinois.edu

Grace Zeng
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
gzeng6@illinois.edu

Craig Zilles
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
zilles@illinois.edu

ABSTRACT

Background: Previous work has shown that students can understand more complicated pieces of code through the use of common software development tools (code execution, debuggers) than they can without them.

Objectives: Given that tools can enable novice programmers to understand more complex code, we believe that students should be explicitly taught to do so, to facilitate their plan acquisition and development as independent programmers. In order to do so, this paper seeks to understand: (1) the relative utility of these tools, (2) the thought process students use to choose a tool, and (3) the degree to which students can choose an appropriate tool to understand a given piece of code.

Method: We used a mixed-methods approach. To explore the relative effectiveness of the tools, we used a randomized control trial study ($N = 421$) to observe student performance with each tool in understanding a range of different code snippets. To explore tool selection, we used a series of think-aloud interviews ($N = 18$) where students were presented with a range of code snippets to understand and were allowed to choose which tool they wanted to use.

Findings: Overall, novices were more often successful comprehending code when provided with access to code execution, perhaps because it was easier to test a larger set of inputs than the debugger. As code complexity increased (as indicated by cyclomatic complexity), students become more successful with the debugger. We found that novices preferred code execution for simpler or familiar code, to quickly verify their understanding and used the debugger on more complex or unfamiliar code or when they were confused about a small subset of the code. High-performing novices were adept at switching between tools, alternating from a detail-oriented to a broader perspective of the code and vice versa, when necessary. Novices who were unsuccessful tended to be overconfident in their incorrect understanding or did not display a willingness to double check their answers using a debugger.

Implications: We can likely teach novices to independently understand code they do not recognize by utilizing code execution and debuggers. Instructors should teach students to recognize when code is complex (e.g., large number of nested loops present), and to carefully step through these loops using debuggers. We should additionally teach students to be cautious to double check their understanding of the code and to self-assess whether they are familiar with the code. They can also be encouraged to strategically switch between execution and debuggers to manage cognitive load, thus maximizing their problem-solving capabilities.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**.

KEYWORDS

code comprehension, debuggers, execution

ACM Reference Format:
Mohammed Hassan, Grace Zeng, and Craig Zilles. 2024. Evaluating How Novices Utilize Debuggers and Code Execution to Understand Code. In *ACM Conference on International Computing Education Research V.1 (ICER '24 Vol. 1), August 13–15, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3632620.3671126

ICER '24 Vol. 1, August 13–15, 2024, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-0475-8/24/08
https://doi.org/10.1145/3632620.3671126

1 INTRODUCTION

The ability to read and understand code, to see "the forest for the trees", that is, to understand the overall purpose of a piece of code from its individual lines, is critical for debugging, making changes to existing code, and coming up with programming solutions [34, 38]. A multi-national, multi-institutional study by the Leeds working group revealed that most novices fail to develop this crucial skill by the end of an introductory programming course.

Prior work [20, 21] has found that novice programmers are able to understand more complex programs through the use of common programming tools (e.g., code execution, debuggers) than they can through source code inspection alone. This shouldn't be surprising as expert programmers use testing (code execution) and debuggers to debug, verify, and understand code. We believe that if students are taught to use these tools to understand code, they can potentially independently understand larger, more complex, and less familiar code, helping them to learn new programming patterns and plans, assisting in their development into expert programmers.

As such, in this paper, we conduct an exploratory study to better understand how we should approach teaching novice programmers to use tools for program comprehension. We specifically focus on two tools. The first is code execution (shown in the middle of Figure 1), which given a code fragment and input values produces output values. Notably, code execution does not show the process of how the actions done on inputs turn into outputs, i.e., "The dynamic execution of a program is invisible. We can see only the effect (output) of the program" [40]. Code execution can be used

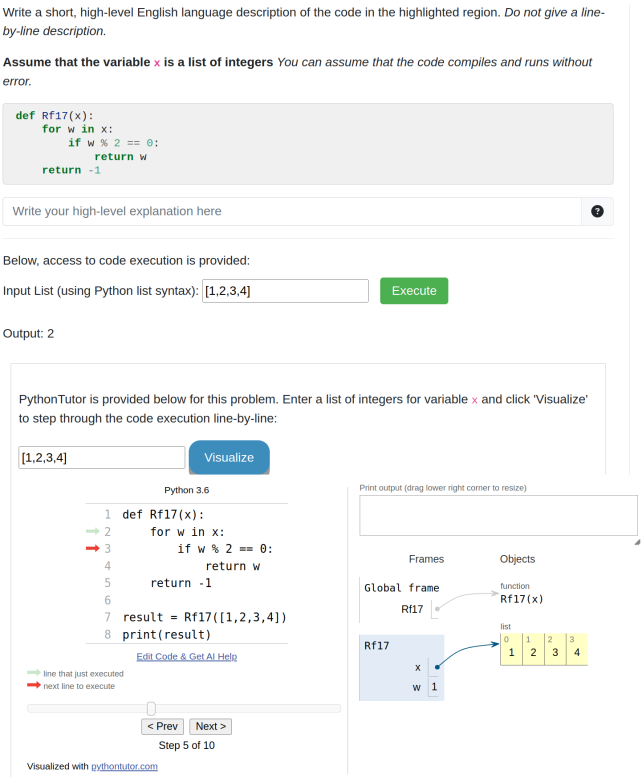


Figure 1: Student's view of provided tools, execution & debugger (PythonTutor) embedded on the EiPE question page. Order of tools presented vary randomly.

to generate a series of input-output pairs that students can use to deduce the behavior of code [51].

The second, line-by-line debuggers (henceforth referred to as just "debuggers"), allow users to step forward the execution one step at a time and see the effect of each step on the state of the machine. Debuggers can be thought of a "white-box" tool in comparison to the "black box" of code execution. For this study, we used the widely used PythonTutor debugger (shown in the bottom of Figure 1), which additionally permits users to step backwards or jump to an arbitrary point of the execution by dragging a slider.

While prior work has shown both of these tools to be helpful, it isn't immediately clear how we should approach teaching students to use them. Does one tool dominate the other one, such that we should focus instruction on just one? Or are there distinct situations where each tool shines? And if the latter, can we convey to students how to choose an appropriate tool? This study aims to fill this gap by exploring novices relative performance using and choice between these tools.

Specifically, our research aims to address the following research questions:

RQ1 How do code execution and debuggers vary in effectiveness toward helping students understand code correctly?

RQ2 What factors influence novices' choice of tool, and do they make appropriate decisions?

To answer these research questions, we conducted a mixed methods study using students taking an introductory programming course. In each part, students are provided tool access to solve 'Explain in Plain-English' (EiPE) questions (an example of which is shown at the top of Figure 1). To answer RQ1, we conducted a randomized control trial with 421 students, where students were (randomly) given access to only one of the tools to understand one of a set of EiPE questions. To answer RQ2, we conducted think-aloud interviews of 18 students that had access to both tools while they solved a series of EiPE questions.

2 BACKGROUND

Unlike novices, experts exhibit metacognition, recognizing when it's necessary to engage in specific strategies for code comprehension based on their self-awareness of their familiarity with a given code base. This includes deciding when to perform code tracing, execute code to see its outputs directly, or use debuggers with breakpoints for an in-depth understanding of code they are not familiar with [10, 18]. When experts encounter code that they do not recognize, they frequently resort to tracing to help them understand the code [10]. This adeptness at selecting and applying the most effective strategy based on their familiarity with the code highlights an important difference between novice and expert programmers, one that this study seeks to address by exploring when and why novices select one strategy over the other.

2.1 Metacognition

Metacognition [46] refers to an individual's awareness of their own understanding of a problem and awareness of their progress through the problem-solving process. Novices often exhibit deficits in metacognitive skills, demonstrated by a reluctance to abandon incorrect solutions because of an incorrect belief that they are nearly done solving a problem, rushing through problems without a thorough understanding, which results in a false sense of achievement, and an inaccurate perception of their position within the problem-solving process. Kalley et al. [28] highlights three different types of metacognition:

- **Declarative metacognitive knowledge** represents one's awareness of **what** they know, awareness of their own abilities and strategies.
- **Procedural metacognitive knowledge** represents one's understanding of **how** to use these strategies efficiently. For high performing learners and experts, they can perform these strategies more accurately and efficiently than novices.
- **Conditional metacognitive knowledge** represents one's understanding of **when** and **why** to use specific strategies. High performing learners and experts use these strategies more easily and with greater flexibility across different situations and contexts.

How novices lack metacognition in their process of debugging programs has been investigated. For example, Gugerty *et al.* studied how novice and expert programmers modified code to fix a bug. Both novices and experts begin by reading the code and making an initial modification to the code in an attempt to fix the bug. Novices often had an initial incorrect fix (modification) and created more bugs in the code, contrary to experts who often had an initial correct

fix. This may demonstrate that experts often created a correct initial hypothesis of the code after reading it, because they had an idea of what the bug in the code was and proposed a fix [18]. In an eye-tracking study, Busjahn *et al.* found that novices read code more linearly (top-to-bottom, left-to-right, like reading English text) than experts, suggesting that experts are more strategic in how they read code to comprehend programs (e.g., following execution order and relevant parts of code) [4].

Murphy et al. and Fitzgerald et al. observed productive and unproductive strategies novice programmers employed when fixing logical errors within a given code snippet. Some productive behaviors include mentally executing code on paper, reading library documentation, using line-by-line debuggers, and recognizing prior learned patterns. Some unproductive behaviors included commenting out correct line(s) of code, completely changing the given code to get the right output (bypassing the problem), and copying irrelevant line(s) of code from other programs [14, 42]. Novice programmers often lacked systematic strategies toward locating and fixing bugs. For instance, novices opt to work arbitrarily with `print` statements and repeatedly apply random fixes [57]. Some novices lacked the meta-cognitive awareness that they may use code execution (e.g., test `print` statements) to off-load cognition to debug a program and instead struggled to mentally execute a large program [13].

Metacognition consists of self-regulation and self-monitoring throughout the problem-solving process. Self-regulation involves planning out solutions and monitoring progress, recognizing when to adjust problem-solving strategies as needed. Self-monitoring, a subset of self-regulation, refers to the process of observing and reflecting on one's cognitive activities, such as their understanding and problem-solving strategies used so far (e.g., was it helpful?).

A common issue among novices is the tendency to jump right into coding when they receive an error without first re-interpreting what the problem is asking, often floundering as they solve for the wrong problem [37]. Denny et al. found that when students are asked to understand input-output test cases before writing code, they are much more likely to understand the problem correctly before beginning to code, thus improving their metacognitive skills [8]. Making students aware of the skills or stages they must progress through via explicit instruction empowers them to monitor their own progress and reflect on the effectiveness of their strategies and approaches. Xie et al. [58] and Loksa et al. [36] found that explicit labeling of skills provided students with a framework to monitor their progress and evaluate their confidence in mastering related skills. Teaching programming problem-solving as a sequence of distinct stages helps learners plan and monitor their progress, evaluate the efficacy of their strategies, and encourages reflection when asking for help. This structured approach supports metacognitive awareness and promotes better learning outcomes [37].

Research has shown that scaffolding metacognitive skills can improve learning outcomes. Scaffolding techniques like reflective prompts, self-assessment, self-questioning, and the use of graphic organizers have been found effective in helping novices. For example, Rum et al. highlighted the positive impact of these scaffolding techniques on novice programmers' performance [47]. Michaeli et al. taught a systematic debugging process in classrooms, which improved students' debugging skills by fostering a planned and deliberate approach, moving away from typical novices' trial-and-error methods [41]. However, the effectiveness of metacognitive scaffolding highly depends on the level of student engagement; active participation and response to the scaffolding are crucial for its success [47]. Wang et al. emphasized the importance of teaching metacognitive strategies to students just starting their degree program and noted the crucial role of instructors in arousing students' interest and motivation in learning metacognition, to ensure students are actively engaged with metacognitive practices by explaining to students that metacognition is a useful skill to learn [54].

Tools designed to scaffold metacognition in comprehension and debugging have shown varied levels of effectiveness. For example, the study by Hull et al. investigated the impact of metacognitive feedback in the SQL-Tutor system. Despite the integration of metacognitive feedback, the differences between control and experimental groups were not significant, likely due to low engagement with the feedback provided [26]. Similarly, SeeC, a tool designed to assist novice C programmers with debugging, showed that while students found the tool useful, the overall uptake was low, which limited its effectiveness [24]. Ardimento et al.'s debugging scaffolding tool demonstrated improvements in students' debugging performance, yet there was no significant impact on students' self-efficacy in debugging [2]. Hauswirth et al. discussed the challenges of metacognitive calibration, where students may engage superficially with self-assessment tasks, such as mechanically self-rating skills without deeper consideration just to get it over with. This suggests that while tools can support metacognitive activities, the design and implementation must ensure deeper engagement [23].

Two significant issues with metacognitive scaffolding tools are their limited availability and whether students engage with provided scaffolds. Custom educational tools designed to enhance metacognitive skills are not widely available, limiting their impact. Additionally, the effectiveness of these tools can vary among students, often due to their lack of active engagement with the provided scaffolding. This lack of engagement can significantly reduce the potential benefits of metacognitive interventions provided within tools. Therefore, in our study, we compare two commonly available tools, PythonTutor and code execution, both of which can engage students in different ways.

2.2 Code Comprehension

Students who are adept at providing high-level explanations of code tend to perform well at writing programs, implying that explaining code (which necessitates understanding) may be a precursor to writing code [16, 38]. On the contrary, explaining code at a low level by restating the code line-by-line does not demonstrate an understanding of the purpose of the code [56]. "Explain in Plain English" (EiPE) questions (e.g., Figure 1), which require students to give a high-level explanation of code, are a common way to evaluate the skills of understanding and explaining code. A recent theory toward learning programming suggests that EiPE questions should be used towards learning common programming patterns [58] to help students read code written by others.

Expert programmers tend to use a mix of top-down and bottom-up strategies for code comprehension. The bottom-up model [45]

(M) Macrostructure	Understanding the overall structure of the program text.	Understanding the *algorithm* underlying a program.	Understanding the goal/purpose of the program (in the context at hand).
(R) Relationships	Relations & references between blocks (e.g. method calls, object creation, data access...).	Sequence of method calls, *object sequence diagrams*.	Understanding how subgoals are related to goals, how function is achieved by subfunctions.
(B) Blocks (Chunks)	*Regions of Interest* (ROI) that syntactically or semantically build a unit.	Operations of a block, a method, or a ROI (chunk from a set of statements).	Understanding the function of a block, seen as a subgoal.
(A) Atoms	Language elements.	Operation of a statement.	Function of a statement: its purpose can only be understood in a context.
	(T) Text Surface	(P) Program Execution	(F) Function/Purpose
Duality	Architecture/Structure Dimensions		Relevance/Intention Dimension

Figure 2: The Block Model for assessing types of code comprehension tasks.

is commonly applied when the code initially seems unfamiliar to the programmer. This model consists of a program model and a situation model. The program model is a control-flow mental representation of the program, made by grouping chunks of the code and leveraging beacons (discussed below) throughout this process (e.g., identifiers, comments, cues to common patterns or plans). Afterward, they build a situation model, a data-flow mental representation of the program, also built by grouping chunks, where programmers apply their problem domain knowledge.

In Brooks' top-down model [3], the programmer begins with an initial hypothesis of the purpose of the entire program based on their domain knowledge (real-world knowledge of the problem the program aims to solve) and hierarchically creates subsidiary hypotheses to explore specific, implementation-level details needed to verify the parent hypothesis leveraging beacons throughout the process. Meanwhile, Soloway et al.'s top-down model involves recognizing code fragments (common patterns or plans) that seem familiar [50].

As tracing code is often regarded as a precursor to writing and abstracting code [25, 31, 32, 35, 39, 52, 53], tracing is a crucial skill that novice programmers must learn before they can reliably fix bugs. Soloway et al. found that if higher-level strategies toward understanding a program fail, programmers resort to concrete line-by-line tracing [10]. Code tracing is the act of executing code either mentally or through bookkeeping on some media, which is commonly evaluated through "find the output" or "find the outcome" of code problems [43]. Being able to trace code does not necessarily indicate that a student has a high-level understanding of the purpose of the code [33]. Teague et al. [52] found that some students use tracing to understand code through inductive reasoning, where students would trace code multiple times and identifying patterns in the input-output pairs [51].

The Block Model (Figure 2) distinguishes between program comprehension tasks pertaining to understanding: 1) text-surface (syntactical *structure*), 2) execution behavior (algorithmic *structure*), and 3) purpose (intent, *function*) [27]. A program's intent is concerned with understanding why a programmer has written a program (i.e., external context, domain), which is the extrinsic purpose of the code and is qualitatively different from understanding execution behavior [44].

An example of a task pertaining to understanding the *structural* relationships between subsets of the code in the *program execution* dimension of the block model can be tracing code for a particular input to develop an understanding of the execution-state relationship between caller code and called procedural units. An example of a task pertaining to understanding the algorithm of the whole program at a high level (i.e., the macrostructure of *program execution* dimension) can be identifying a comprehensive set of inputs to access all control-flow paths of a program [27].

As for understanding *intent*, an example of such a task could be selecting suitable variable names within a program. This process involves understanding the *functional* relationship between (purposeful) sub-goals within the code. These relationships come together to establish the program's overarching goal or *intent* of the program. Choosing an appropriate name for the whole program pertains to understanding the macrostructure of the program's *intent* [27].

3 METHOD

We apply a mixed-methods approach to understand the relative effectiveness of execution and debuggers on novice programming students' ability to understand code and how they choose between the two tools across a range of Explain in Plain English (EiPE) questions. We describe the method for the randomized control trial in Section 3.1 and the qualitative interviews in Section 3.2.

The studies were conducted at the University of Illinois at Urbana-Champaign. The subjects were enrolled in a introductory Python programming course taught for non-CS, non-engineering, undergraduate students. Data was collected with informed consent under the guidance of the university's IRB.

We used the same set of EiPE questions in both parts of the study. They are included as Figures 3-12. These questions were designed to be relatively challenging for this student population, so as to necessitate tool usage.

3.1 Quantitative Data Collection

We conducted a randomized control trial in the Fall 2023 semester. As part of a mid-term computerized exam [59] toward the end of the semester, students in the course were asked to solve an EiPE question, and on the question page they were provided access to a tool (either execution or debugger). The EiPE problem was a random selection from the first eight of the EiPE questions shown, and which tool they were given was also randomized. To evaluate tool

Assume that the variable x is an integer.

```
def f19(x):
    o = 0
    while x > 0:
        if (x % 10) % 2 == 0:
            o += 1
        x //= 10
    return o
```

Figure 3: An EiPE question with the high-level description of "counts the number of even digits present in a given input number."

Assume that the variable x is a list of integers.

```
def f(x):
    for i in range(len(x)):
        if x[i] < 0:
            x[i] = -x[i]
    return x
```

Figure 4: An EiPE question with the high-level description of "replaces all numbers in a given list with their absolute value."

Assume that the variable x is a list of integers.

```
def f(x):
    max_num = max(x)
    rn = int(str(max_num)[::-1])
    x[x.index(max_num)] = rn
    return x
```

Figure 5: An EiPE question with the high-level description of "reverses the digits of the largest number in a list."

effectiveness (rather than students' choice of tool), students were randomly assigned to *only one* of the two tools. Practice exams for this exam included both types of questions (i.e., ones with execution and ones with a debugger).

The answers to the EiPE questions were graded for correctness, ambiguity, and being high-level [5]. In addition, the computerized exam recorded additional information about the question, including: question duration, inputs selected, and the lines stepped through using the debugger.

3.2 Qualitative Data Collection

We conducted 18 1-hour think aloud sessions. The think-alouds were conducted over Zoom, screen-recorded, and transcribed verbatim. Students who had completed the introductory programming class during the Fall 2023 semester were recruited by email and compensated with a $35 gift card for their time. Participants solved

Assume that the variable x is a list of integers.

```
def f(x):
    x.sort()
    n = len(x)
    if n % 2 == 0:
        k1 = x[n // 2]
        k2 = x[n // 2 - 1]
        k = (k1 + k2) / 2
    else:
        k = x[n // 2]
    return k
```

Figure 6: An EiPE question with the high-level description of "finds the median number in a given list."

Assume that the variable x is a list of integers.

```
def Rf17(x):
    for w in x:
        if w % 2 == 0:
            return w
    return -1
```

Figure 7: An EiPE question with the high-level description of "finds the first even number present in a list of numbers."

Assume that the variable k is a list of integers.

```
def f(k):
    s = 0
    o = len(k)
    while s < o:
        if k[s] < 0:
            k.remove(k[s])
            o -= 1
            s -= 1
        s += 1
    k.sort()
    return k
```

Figure 8: An EiPE question with the high-level description of "sorts list in increasing order removing negative elements."

a series of 'Explain in Plain English' (EiPE) questions. At the beginning of each interview, the participant was asked to fill out a survey self-rating their prior familiarity with PythonTutor and code execution on a 6-point scale, to understand if their prior familiarity with the tool may effect how they choose to use it. As shown in Figures 13 and 14, most participants indicated they were at least somewhat familiar with the tools but not highly experienced.

To understand what factors influence novices' choice of one tool over another, and under what circumstances they make these

Assume that the variable x is a list of integers.

```
def f(x):
    p = []
    for i in x:
        if i > 1:
            for m in range(2, i):
                if (i % m) == 0:
                    break
            else:
                p.append(i)
    return len(p) == len(x)
```

Figure 9: An EiPE question with the high-level description of "checks if all elements in a list are prime numbers."

Assume that the variable x is a list of integers.

```
def f(x):
    result = []
    for k in x:
        if k > 0:
            m = k * 2
            if m % 3 == 0:
                result.append(m)
    return result
```

Figure 10: An EiPE question with the high-level description of "doubles each positive number that is divisible by 3."

decisions, participants were offered access to both PythonTutor and code execution tools. This design choice mirrors real-world coding scenarios where developers have access to various tools and must decide which best suits their immediate needs. These problems were administered through a web page where each problem page has PythonTutor and code execution embedded (see Figure 1), where students can choose inputs and select an 'execute' or 'visualize' button. The number of inputs and lines of code executed using each tool were recorded for each task, as well as time spent using each tool.

As participants were solving the problems, they were left to independently work on the problems with access to both tools. They were prompted to switch tools only if the student struggled or repeatedly incorrectly solved a problem, making no progress for more than about 5 minutes using a tool. If they struggled for more than 5 minutes and were not using a tool, they were asked to select a tool. After the student has completed attempting the problem (i.e., gave an explanation of the code and wrote it in the provided answer box), the interviewer asked the student retrospective questions. These questions were:

- If they switched between tools, they were asked why they chose to switch between tools.
- If they selected one tool and not the other, they were asked why they choose one specific tool.

Assume that the variable lst is a list of integers.

```
def func_a(lst):
    lst_b = [0] * len(lst)
    var_c = 0
    var_d = len(lst) - 1

    for x in range(var_d):
        lst_b[x] = abs(lst[x] - lst[x + 1])
        if lst_b[x] > var_c:
            var_c = lst_b[x]

    var_e = 0
    for y in range(len(lst_b)):
        if lst_b[y] == var_c:
            var_e += 1

    return var_c, var_e
```

Figure 11: An EiPE question with the high-level description of "returns the largest difference between two adjacent elements and the number of times this difference occurs."

Students were asked to verbalize their raw thought processes and decisions as they work through the programming tasks. Our think-alouds followed the protocol of Ericsson et al. for recording unstructured verbalizations [12]. Their approach aims to minimize the extra cognitive effort required to verbalize thought processes to help prevent fatigue from impacting student performance. Participants were only asked to say what was currently on their mind as they were solving the problems and were not asked to explain nor interpret their thought process for our benefit [12].

3.3 Qualitative Analysis Method

The goal of our analysis was to conduct thematic analysis, identifying, analyzing, and reporting themes within the think-aloud session transcripts. The initial stage of analysis was inductive open coding of the think-aloud session transcripts. Special attention was paid to moments of struggle, decision points regarding tool use, and expressions of confidence or frustration. The think-aloud interviews were analyzed in conjunction with video footage. This process involves reading through the transcripts line by line to identify and label common themes as they naturally emerge from the data. Codes representing similar concepts were grouped into categories, forming the basis for identifying broader themes related to tool preference, problem-solving strategies, and code comprehension. The constant comparative method was applied throughout the coding process. This involved continuously comparing new codes to existing codes and categories to refine and elaborate on the emerging themes. We applied reflexivity, where the researchers' biases were examined in relation to the data. Reflexive coding involved maintaining a journal to document our thoughts, assumptions, and decision-making processes throughout the analysis. First two authors first coded

Assume that the variable var_a is an integer.

```
def func_var(var_a):
    var_b = 0
    var_c = 0

    for var_d in range(1, var_a):
        var_e = 0
        var_f, var_g = 0, 0

        for var_h in range(var_d, var_a):
            var_e += var_h
            if var_h % 2 == 0:
                var_f += 1
            else:
                var_g += 1

            if var_e == var_a:
                if var_f > var_g:
                    var_b += 1
                elif var_g > var_f:
                    var_c += 1
                break
            elif var_e > var_a:
                break

    if var_b > var_c:
        return "A"
    elif var_c > var_b:
        return "B"
    else:
        return "C"
```

Figure 12: An EiPE question with the high-level description of "checks whether there are more even or odd consecutive sums from 1 to var_a."

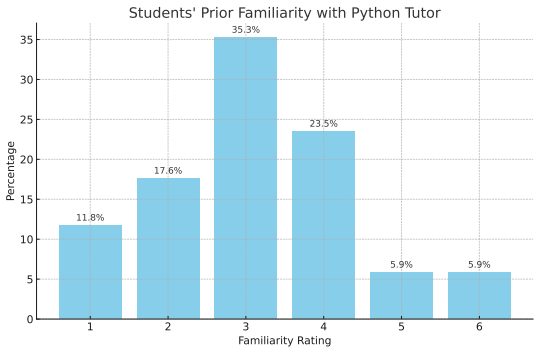


Figure 13: Think-aloud participants' self-reported familiarity with PythonTutor.

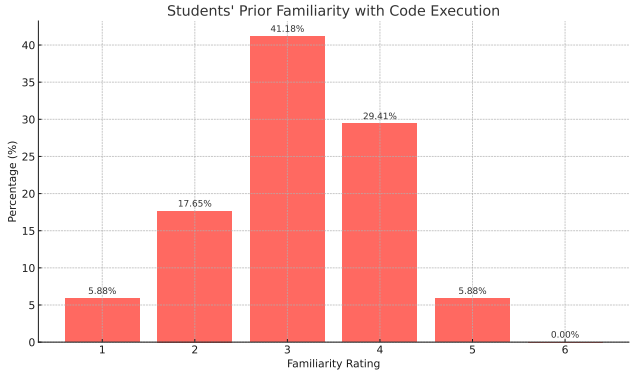


Figure 14: Think-aloud participants' self-reported familiarity with Code Execution.

the data independently, and then met to compare and discuss their codes and interpretations, resolving disagreements.

3.4 Limitations

Our data may not be fully representative of the broader student population. The qualitative portion of our study involved a relatively small sample size due to the extensive data analysis required for each interview and the significant time commitments for coordinating and conducting these interviews. Participants in the qualitative segment were self-selected, which may indicate a higher-than-average level of self-confidence as they chose to participate in the study, potentially skewing the results.

The average exam score for the participants in the qualitative portion of the study (N = 18) was approximately 84.07%, while the average score for the quantitative portion (N = 421) was around 82.2%. This similarity in average exam scores suggests that, in terms of programming ability, the interviewed participants may be representative of the larger participant pool involved in the quantitative analysis.

However, the self-selection bias and small sample size in the qualitative study may limit the generalizability of our findings. The higher self-confidence of these participants could influence their engagement and performance, possibly affecting the study's outcomes. Consequently, these factors should be considered when interpreting the results and their applicability to the general student population.

Our results may be limited to PythonTutor's unique visualization. Our findings may be specific to PythonTutor and may not fully apply to other debugging tools. Future work should explore the effectiveness of different debugging tools and visualizations in various IDEs to provide a more comprehensive understanding of how these tools help students understand code.

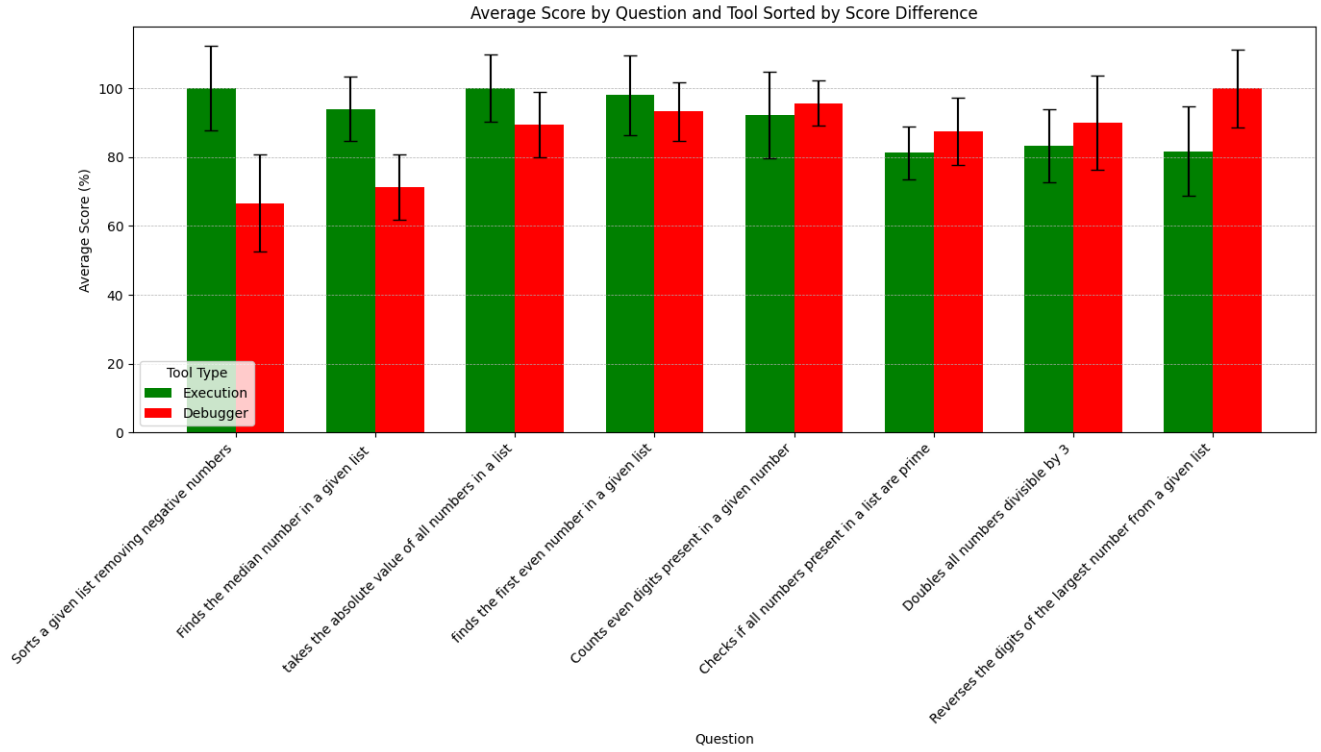


Figure 15: Bar plot showing average performance on question by tool. Going from left to right are questions in the Figures 8, 6, 4, 7, 3, 9, 10, 5

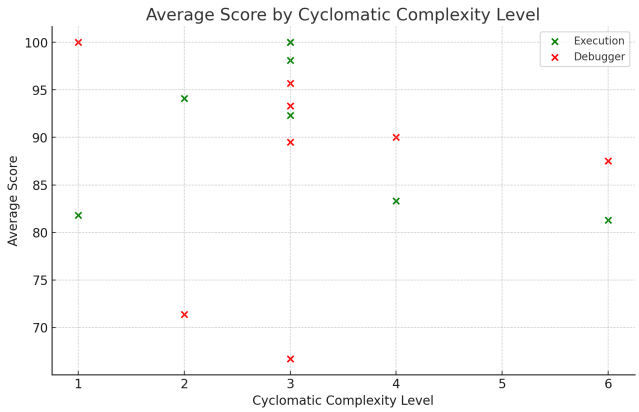


Figure 16: Scatter plot showing average performance on question by cyclomatic complexity level.

4 QUANTITATIVE RESULTS (RQ1): PERFORMANCE

4.1 Students often do better with code execution

The average performance of students solving EiPE questions with access to code execution was 44.9% (Std. Error 1.2), while for debuggers was 30.1% (Std. Error 1.7). Overall, students performed better with code execution to a statistically significant degree ($p = 0.00523 < 0.05$).

Interestingly, the average number of inputs that students tried for code execution was 3.63 and for debuggers was 2.55. A t-test finds this difference between conditions to be significant ($p = 0.02 < 0.05$). This may imply that overall, the act of trying more inputs may be more important towards solving EiPE questions and easier to do with code execution than a debugger.

4.2 Debuggers becomes more useful as code complexity increases

If we look into the success rates on the various problems (Figure 15), we see that the relative performance of the two tools differs between the codes. Code execution is statistically significantly better on the first two, leftmost problems, and there is a continuum of not statistically significant differences that bias towards execution to those that bias toward the debugger.

Manually inspecting these codes, we generally observe a tendency that, as cyclomatic complexity increases, students assigned to debuggers start to outperform students assigned to code execution. Cyclomatic complexity measures the number of distinct execution paths through a program. Figure 16 plots the student performance of each tool vs. cyclomatic complexity, and it can be seen that the two most complex are among the ones where students do better with the debugger. The notable exception to this trend

$$\text{score\_on\_question} = \text{average\_exam\_score} + \text{cyclomatic\_complexity} \times \text{debugger\_tool}$$

Figure 17: The logistic regression formula

Table 1: Logistic Regression Model: Interaction Between Tools and Code Complexity Level

Coefficient	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-4.34	1.32	-3.30	0.00096 ***
average_exam_score	0.0629	0.0143	4.39	1.13e-05 ***
cyclomatic_complexity	-0.413	0.133	-3.110	0.00187 **
debugger_tool	-1.75	0.614	-2.85	0.00444 **
cyclomatic_complexity:debugger_tool	0.365	0.180	2.03	0.0423 *

is the problem with the lowest cyclomatic complexity, shown in Figure 5, which also has higher performance with the debugger. This particular question has a large amount of nested syntactical expressions, potentially indicating that debuggers may be more useful in such cases, as well. Overall, this graph suggests that as cyclomatic complexity increases, debuggers may be more useful.

To explore this trend, we ran the logistic regression shown in Figure 17. The dependent variable was whether the student solved the problem correctly (binary correct/incorrect). This output was predicted from three independent variables: 1) the tool type (debugger_tool) which is encoded as 1 (debugger) and 0 (execution), 2) the cyclomatic complexity (an integer between 1 and 6, inclusive), and 3) the student's average exam score across the whole semester (a number from 0 to 100), to control for student ability. We include an interaction term between tool type and cyclomatic complexity to confirm the above finding.

The logistic regression model demonstrated moderate to good fit and predictive ability. The McFadden's pseudo R-squared values was approximately 0.12, indicating an acceptable level of explanatory power. The Area Under the Curve (AUC) for the Receiver Operating Characteristic (ROC) curve was observed to be around 0.74, suggesting that the model have a good ability to discriminate between students who performed well and those who did not. The Variance Inflation Factor (VIF) values were within acceptable limits (below 2.5), indicating that multicollinearity was not a significant concern in this model. Furthermore, the model successfully converged, which supports the reliability of the estimated coefficients. Overall, these statistics suggest that the model fits the data well without over-fitting.

The results of the regression (shown in Table 1) indicate that while debuggers are generally less helpful than code execution, as shown by a significant negative coefficient ($p = 0.004^{**}$), and that cyclomatic complexity generally reduces students success, debuggers' effectiveness increases with the cyclomatic complexity of the code. This is shown by the positive and statistically significant interaction between cyclomatic complexity and the use of debuggers ($p = 0.042 < 0.05$), suggesting that for more complex problems, debuggers may offer more of an advantage.

5 QUALITATIVE RESULTS (RQ2): TOOL SELECTION

Overall, we observe that students generally choose a tool that helps them correctly understand code. In 72% of the problems attempted in the think-alouds, students correctly solved the question with the first tool that they chose.

Students generally expressed reasoning about their choice of tool that correlates to the findings in Section 4. They chose code execution on simpler and familiar code and the debugger for more complex code.

> **Participant 5:** If it's a simple code, you don't even need to go into the visualizations, you're just wasting your time.
> **Participant 5:** But if it's a more complex code that you're having a harder time understanding, then yeah, sure. Go into using the visualization tool

In this section, we go into greater detail to explain students tool choices. In Section 6, we highlight cases where students used both tools together, and, in Section 7, we present cases where students were unsuccessful solving problems based on how they selected between the tools.

5.1 Students prefer execution on code they're familiar with to double check understanding

Some students used code execution in the case the code was familiar to them. These students expressed that they needed to confirm their understanding of the code and decided to use code execution as a quick way to double check their understanding.

Participant 2 described how they recall seeing a similar piece of code in their programming class and, thus, recognized its purpose easily:

> **Interviewer:** I wanted to ask you why you chose the execute tool this time.
> **Participant 2:** I was more confident with what I was doing... but I just wanted to double-check that my answer was lining up, so I was less concerned with the process and more concerned with the general answer... I've seen this question on a test before, so I kind of remember the general premise of how it's

> done. I just was– I just wanted to make sure I was correct. (Figure 7)

The participant seems to express that they did not need a tool to demonstrate how the code works and rather just wanted to double-check their understanding by viewing outputs. The detail-oriented nature of PythonTutor was not necessary because they already understood the code by recalling what they learned in class. Code execution was sufficient to check their answer.

Participant 5 used code execution in a similar way:

> **Participant 5:** Let me just double check something. We'll say 5, 3, -1. Yeah, that's what I thought.
> **Interviewer:** So what made you choose the execute tool this time immediately?
> **Participant 5:** It was pretty obvious to me because I was like, so it was a for loop. I got that. I knew that straight away. And then I knew if it's an even number, because if it's divisible by two, and then there's no remainder, so it's even return 'W'... I feel like I've seen something similar to this. (Figure 7)

Participant 5 demonstrated a self-awareness that they know and understand this code, but was cautious to double-check their understanding. Participant 6 additionally described how this can more often be the case on what they believe are simpler pieces of code.

> **Participant 6:** Just one for loop and just like a singular list. It's more easy to just use execute since I can follow it by reading the code.

In these cases, where the code is simple enough for the participant to recognize an input-output pattern along with tracing the code by themselves, the act of viewing input-output pairs using code execution alone was sufficient. When participants were able to recognize input-output patterns, often on simpler problems, the detail-oriented nature of PythonTutor was not necessary; trying inputs on code execution was sufficient.

Participant 5 expressed that, in cases that they are independently capable of understanding the code by reading it, they may find PythonTutor distracting and execution preferable.

> **Participant 5:** I feel like, with the whole like visualization, it gets a little confusing for me. I just want to see what the inputs are and then what the result is. And then I can do, I guess, like, make my own inferences about what's going on between. It's easier to understand using the execute tool, because it's less confusing. Like, visualizations can be a good thing and a bad thing. it could be a good thing if I'm inputting these values, and I'm still not understanding what's going on. Otherwise, with execute, I can make my own inferences, which also enhances my ability to read code.

Some participants have expressed that they may prefer code execution if they are able to independently infer the purpose of the code.

5.2 Students prefer the debugger on unfamiliar and complex code

Participants describe preferring to select PythonTutor over execution in situations the code looks too complex to them, appearing to not be familiar with the code.

> **Participant 2:** I'm not sure really what the whole goal of this is. It just seems like a complicated function. But let me look through the visualizer [PythonTutor]. I just– I couldn't use the [execution tool] because it would overwhelm me too much. (Figure 6)

When they believe the code is complex (e.g., has nested loops), the code execution tool is often insufficient towards helping them understand the code. Therefore, they prefer selecting the debugger.

> **Participant 2:** [On code execution] I thought I could just try all those numbers to see maybe oh 'all these numbers are divisible by four' or something simple like that but that didn't really work out. So then I went to [PythonTutor], because I wanted to look at if I was missing something, which I ended up; I was missing something because it kept; it was reiterating at the beginning of the loop, which I didn't realize because there were two nested loops within each other so that's where I think having [PythonTutor] helped me, because it made me look at where it iterated instead of just the end result (Figure 12)

The participant expressed how the execution tool did not make clear the nested loop behavior of the code, which is why they preferred the debugger. They appear to demonstrate a self-awareness that they do not understand how inputs lead to outputs, and therefore chose the debugger to observe this process.

Participant 5 described the challenges of making sense of outputs that appeared random or unintuitive, therefore preferring PythonTutor in those specific situations:

> **Participant 5:** I would use [PythonTutor] 100%, if I was not understanding- Let's say I put in these numbers, right? But it gave me, like, some whack numbers, like, some weird numbers... then I would come into using PythonTutor and really, like, hone in on, like, what is going on, what I'm missing, why it's the values that came out that it came out to be and you know, what's really going on in the situation. (Figure 8)

In the examples below, these participants described entering several inputs on the code execution tool but not seeing any clear pattern from input-output pairs. Then, when they used the debugger, they were successful in understanding the code.

> **Participant 6:** I tried out arbitrary, random numbers, and it would just continue giving me the output 'False'. That wasn't very helpful, because I don't know what the code is doing to my numbers to return False. (Figure 9)
> **Participant 8:** Execution is just an output. I don't know what's going on behind the scenes, so going into PythonTutor allows me to see that. On [code execution]... I know that I'm not seeing the pattern. So I just went into PythonTutor. (Figure 5)

Participants 6 and 8 describe that they struggled to understand how their chosen inputs led to and related to the function's outputs when using code execution alone, therefore needing the debugger. Participants expressed that code execution did not help them understand code that did not have an obvious input-output pattern to them. Therefore, they selected the debugger to understand how the operations done on inputs lead to outputs.

Overall, participants described how they struggled to recognize an input-output pattern, especially on more complex pieces of code with less predictable patterns, and that PythonTutor proved to be the more effective tool to help them understand the code. Unlike direct code execution, which offers limited insight into how inputs lead to outputs, PythonTutor shows this information by visualizing each step of the code's execution process.

Additionally, participants express how they prefer using the debugger on code with poor, similar variable names. They describe how in this situation, it is more difficult to mentally keep track of the value of each variable without using the debugger.

> **Interviewer:** So why do you feel like when there's three or more variables being updated that you need the visualizer?
> **Participant 2:** Just because they all get mixed up, especially when you're dealing with two nested loops where they're both updating Also, they all sound very similar. Var-E, Var-D, Var-B, stuff like that. That makes a difference because it's hard to say out loud. Like if Var-B was equal to pizza and Var-C was equal to light bulb, then it might be easier to just think like, oh, pizza is this and light bulb is this. But Var-B and Var-C, they all just sound so similar. when those variables are all named the same thing, I need an extra tool to help me show how the variables are being updated. (Figure 12)

The participant expresses how execution alone does not make clear how these variables update, or i.e., how inputs lead to outputs.

5.3 Students independently switch from using execution to the debugger when the input-output pattern is initially unclear

Often, participants attempt to use code execution first in an attempt to understand the code quickly, but struggle to recognize any input-output pattern, then independently decide to switch to the debugger to understand how inputs lead to outputs.

> **Interviewer:** So I noticed that first you chose the code execution tool, and then after that you switched to the visualizer. What made you do that?
> **Participant 3:** Well, I wanted to start with whatever's quickest first [to] see if that'll help me understand, and since it didn't, then I went into the more thorough thing.
> **Interviewer:** Why was the execute tool not helpful enough?
> **Participant 3:** So since execute doesn't, like, show the work, I wasn't able to see list P. But since I was able to see list P within visualize, I saw what was getting put into the list. And that helped me see, 'oh, it's only the prime numbers that are getting put in the list.' (Figure 9)

Participant 3 demonstrates self-awareness that they needed more information about the code's execution process, thus independently selecting the debugger after failing to understand it using code execution alone. They describe that PythonTutor's visualization of the elements being appended to a new list guided them to the correct purpose, whereas a simple input and output tool would not have been able to effectively show this process.

5.4 Students prefer the debugger when they're self-aware/identify a specific part of the code they're not familiar with

Participants also prefer to use PythonTutor in the case they generally understand how the code works, but would like to address a specific misunderstanding or understand a specific line(s) of code that they are not familiar with.

> **Participant 5:** I had to go use PythonTutor and really crack down on, like, what I wasn't understanding

The participant below states how they are unfamiliar with how the code executes one specific line (`return -1`), but is otherwise demonstrating an understanding of the rest of the code. Thus, they use the debugger to step through the specific line they do not understand.

> **Participant 4:** From what I learned, it's asking if w is an even number, and if it is an even number, then it returns that value. But I will go ahead and use the visual tool because I am confused on the return minus one part. This is why I really used the PythonTutor, to see if it was an odd number will it return negative one. (Figure 7)
> **Interviewer:** ... what lead you to choosing PythonTutor?
> **Participant 4:** So here I knew that, well, I knew what was going on. I just had to verify one last thing, the return negative one. So the PythonTutor helped me verify that. The [debugger] was more useful when I was stuck on one specific part of code.

They appeared to demonstrate a self-awareness of what they correctly understood and what they did not understand about the code, thus choosing PythonTutor to address their specific unfamiliarity or their specific misunderstanding.

6 STRATEGIES SUCCESSFUL PARTICIPANTS USED UTILIZING BOTH TOOLS

In addition, we saw some instances where students strategically switched between the tools to efficiently understand code using both tools.

6.1 Students switch from the debugger to execution to initially break-down & familiarize with code then confirm understanding with more inputs

In the first case, shown below, the participant expresses how they prefer to use the debugger initially to familiarize themselves with how the code works, then, afterwards, use code execution to confirm their understanding.

They express how using execution at the beginning can be overwhelming for them and why they would prefer to use the debugger at first.

> **Interviewer:** Is there anything else you wanted to also share about the PythonTutor versus the executor?
> **Participant 2:** At the beginning, I just, I couldn't use the [execution tool's] bird's eye view because it would overwhelm me too much.
>
> [PythonTutor] was a good kind of kickstarter to get me thinking in code instead of just thinking, 'oh my gosh look at all these numbers and letters.' That thing in the side where it shows you what all the variables are at that point in time definitely helped me ... simplify it to a step-by-step thing instead of trying to analyze it all at once, breaking it down to it into its lowest parts line by line. So, thinking code, like, just to make it easier for me to understand.

Then afterwards, they used code execution to confirm their understanding after sufficiently familiarizing themselves from using the debugger.

> ... But then once I get to the end, the [code execution tool] bird's eye view is more helpful because I look at where did I start? What did I input? What variables came from that? And then what did I end with? I ended with A because of this and then this. And I look at all of it holistically instead of the line by line like I did at the beginning.

The debugger helps students become familiar with the code, then, as described earlier in the paper, they use code execution to then confirm their understanding with more inputs. This may demonstrate they are setting expectations of how inputs lead to outputs, thus making it easier to test, confirm, and revise later hypotheses using code execution with more inputs.

6.2 Top-down reasoning: Students switch from execute to the debugger to initially identify all possible distinct input-output behaviors then understand them

We observed a complementary case where they prefer to have a 'general overall scope' first with code execution.

> **Participant 6:** I started with the execute tool because I wanted to get like a general overall scope of, like, what the function would output just to, like, familiarize myself with the function. (Figure 3)

They express how they use code execution initially to identify all possible distinct input-output behaviors, then use the debugger to help them understand these behaviors.

> I feel like having a bunch of different outputs first shows what the code does to alter the inputted value. And if I have that information, I'm able to understand what to input for it to visualize for different scenarios.

Then, they dived down into the details with the debugger to address specific uncertainties in their understanding.

> But after I did the execute function a couple of times, it was hard for me to find an exact pattern because I had confusion on line four. It's a statement. And so to clear up my confusion, I wanted to use the visualize tool to look deeper into line four, the line I was confused with, and see a step-by-step way of what changes line four would make to my answer.

They express how using the debugger initially may be overwhelming for them, as they do not have a clear sense of what inputs would be useful to help them understand more distinct behaviors of the code. Knowing what output to expect makes the process of understanding PythonTutor's step-through easier.

> Because if I went straight to visualize, and I just put in things, I don't know what to expect for different values until I go through every single step again.

With code execution, they could identify what inputs lead to distinct, unique execution behaviors, then understand those distinct behaviors using the debugger.

> I use execute first to tell me if they have different outputs. And then I can use that information to know that I only have to visualize the ones that have different outputs. It kind of cuts down what is– like, cuts down all the unnecessary things to test. And then only test the necessary things. The ones that have different outputs. To understand what the code is doing, I want to find different inputs that create different outputs. And if I realize that there are some inputs that create the same output, then it doesn't makes sense to visualize them both, since they are, like, the same. So I wouldn't do a repeat. Instead, I will try to visualize different scenarios, since those help me understand what the code does best.

The advantages of this top-down-like strategy is that they can 1) understand what output to expect as they walkthrough PythonTutor, making the process of understanding the walkthrough easier, and 2) identify all distinct, different input-output execution behaviors quickly before testing them in the debugger.

7 UNSUCCESSFUL STRATEGIES

Here we describe reasons participants were unable to correctly understand code despite using the tools. Generally, unsuccessful participants did not (carefully) use PythonTutor to step through lines of code that they did not initially understand, where viewing the output from code execution alone was insufficient to help them address their misunderstanding. When the interviewer prompted

them to carefully step through parts of the program they misunderstood, they were often successful in understanding the code.

7.1 Preferring execution on extremely complex code that is too tedious to walkthrough, failing to recognize input-output patterns

On severely complex pieces of code that contain too many steps to walkthrough, a few participants preferred to use code execution as they perceived the excessive number of steps to be 'too distracting' and 'overwhelming to understand.' As a result, they answer it incorrectly as they failed to understand how inputs relate to outputs from execution alone.

> **Participant 8:** (Answers incorrectly using execution)
> **Interviewer:** Why did you choose execution?
> **Participant 8:** Honestly, just for time's sake because I did see it was like 27 steps. (Figure 11)

When the interviewer prompted them to use PythonTutor, they are then able to correctly understand the code.

> **Interviewer:** Could you try that question again but using PythonTutor?
> **Participant 8:** ... I see how these two [differences equal] nine, And the output is 9 and 1. I guess it's just, like, the highest... Times, like, highest difference between numbers (answers correctly).
> **Interviewer:** So which tool did you think was more helpful in this case?
> **Participant 8:** PythonTutor because it was a lot more clear in the lists ... It was a good visualizer because it would take a long time for me just to hand generate the different lists, well, the second list, and then the final output. So PythonTutor is able to do that all for me. This is the first list and I see like there's two lists. There's list underscore b and then there's just list. So I was able to see, like, every single number in every single list and the final output. So it's very convenient ... For code execution, I think... It just shows the output, it doesn't really explain it.

They described how they realized that PythonTutor was a more useful tool upon attempting to use it, and how viewing the output of execution alone was insufficient.

7.2 Participants failing to notice & address misunderstandings using execution, thus not independently selecting PythonTutor

Occasionally, participants believed they had correctly understood the code and thus chose code execution to double check their understanding. However, these participants did not fully understand its purpose from viewing outputs alone. These participants appeared to not independently realize that they did not correctly understand the code, thus not selecting PythonTutor. When the interviewer asked these participants to use PythonTutor, they realized their misunderstanding and corrected their answer.

For example, Participant 18 believed that they had sufficiently understood the code and chose code execution to check their answer. They expressed how they did not feel the need to use PythonTutor.

> **Interviewer:** So, why do you choose code execution immediately?
> **Participant 18:** Well, I definitely thought I had a much better understanding of this problem. And so I kind of figured that from looking at it, it has something to do with, like, being divisible by three. So because I felt like I kind of knew what was going on, I didn't really need to see it worked out, I could just kind of, like, punch in some numbers. (Figure 10)
> **Interviewer:** But then, like, how did your thought process change when I told you to use PythonTutor?
> **Participant 18:** Well, I realized because, like, I wasn't really getting to the answer that was needed; that I needed to take a deeper look into it and try and get a better understanding of what was going on... Earlier I was definitely just kind of overlooking; like, I realized that the numbers were being multiplied by two. So, that was something that the visualization was, like, I need to be paying attention to that.

Reflecting on their thought process after using PythonTutor, they explain how PythonTutor helped them catch details of the code they initially missed when using code execution:

> **Interviewer:** How do you revise your explanation?
> **Participant 18:** I would change it from what I have now to function is filtering the list X and is returning multiples of 3.
> **Interviewer:** What tool do you think was more useful in this problem?
> **Participant 18:** Definitely PythonTutor, just because I got kind of close, but I was still kind of missing the visualization of what was going on. Because as soon as I plugged in the numbers on the PythonTutor, I realized that it was filtering through [the list] rather than just returning the multiples of three.

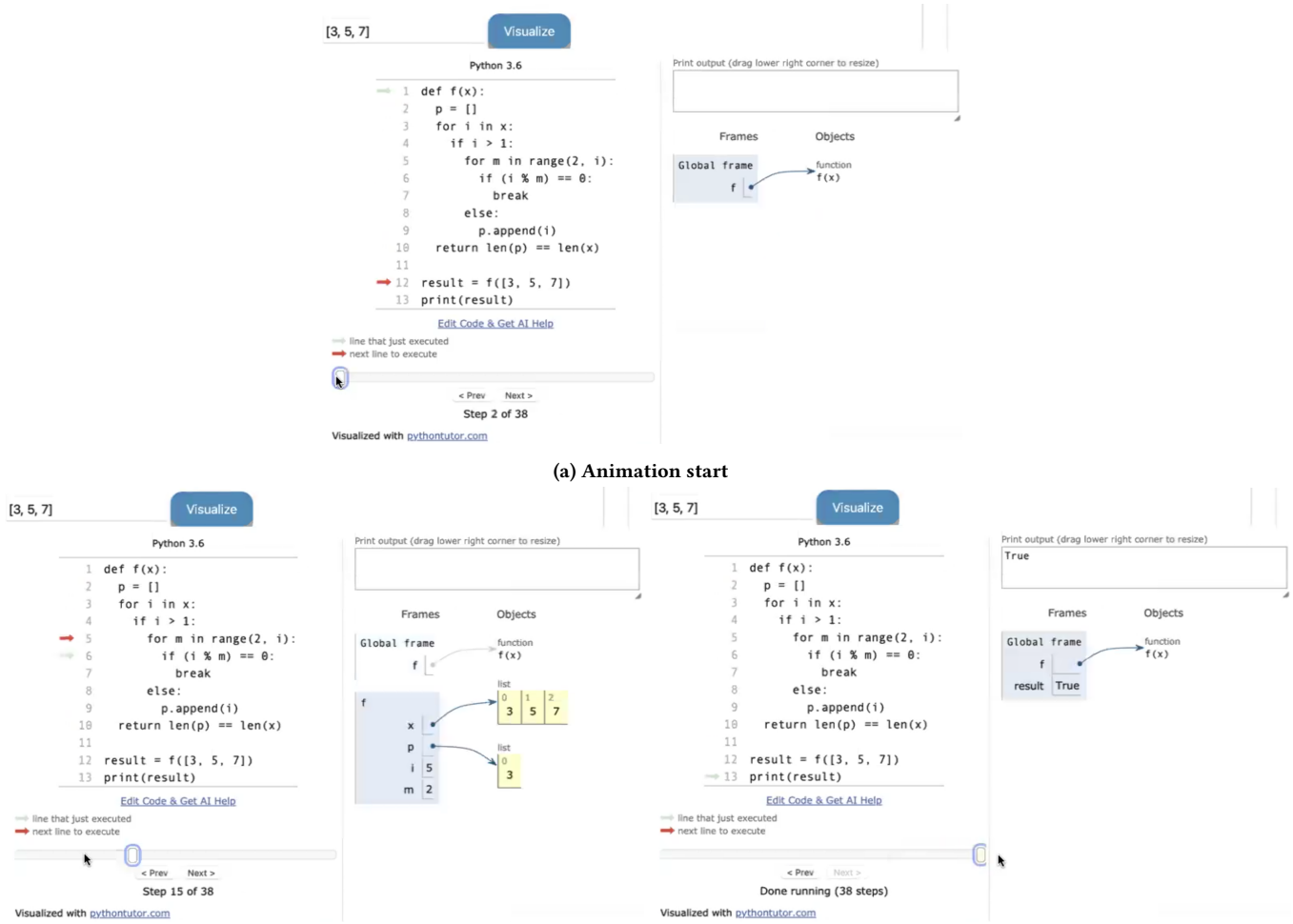
Before using PythonTutor, the participant did not fully understand the function's process of filtering through the list and misinterpreted the code as returning values rather than a list. Only when they were asked to use PythonTutor did they realize this list-filtering process. Code execution alone was insufficient for them to realize that they misunderstood this process.

7.3 On PythonTutor, they do not step through complex loops carefully & not entirely line-by-line, skipping around too quickly

Some participants using PythonTutor would skip through the entire code, or entire nested loops quickly, not stepping through the entire program carefully one step at a time (e.g., Figure 18). As a result, they appear confused about why the value of variables was present in the visualization.

For example, Participant 10 below uses PythonTutor, and skips immediately to the end of execution, observing just what is returned and then explains the code incorrectly.

> **Participant 10:** (skips quickly in PythonTutor, Figure 18) Returns false. Then if I change that to 7, returns



(a) Animation start

(b) 2nd step

(c) 3rd step

Figure 18: Example of participant skipping quickly through PythonTutor and failing to understand the code as a result.

> true (answers incorrectly: "returns true if all values are odd") (Figure 9)

When asked to try again, they step through the entire program carefully, line-by-line, follow and interpret the step-by-step visualization, then answer the question correctly.

> **Participant 10:** (steps through, carefully, clicking 'next' step by step) So we defined a function which we got was 3. We make a new empty list. And then for every object that is greater, which is true, so it goes on to n in the range of 2 to i. And the first range would be 2. So if this is even, the function breaks. Else it appends it to the new list. Let's try with a different number. So we start with 7 and then we go to 3. I will list it below here. We see if it's... Prime value if all values are prime.

They describe how the value of variables change, then answer the question correctly.

Overall, these participants may lack the independent self-awareness that would have aided in honing in on specific sections of code that they did not initially understand. In these cases, these participants needed prompting to be more cautious to step through lines of code they specifically misunderstood.

7.4 On PythonTutor, they do not attempt to explain relationships between variables

Another reason participants fail to understand PythonTutor's visualization is because they attempt to describe the value of each variable in isolation, not attempting to compare and relate variables that depend on each other (e.g., variables which are assigned to some mathematical combination of other variables).

For example, participant 3 tried multiple inputs and walking through the code in PythonTutor, expressing confusion.

> **Participant 3:** I'm confused at what this pattern is. Should be a 5 at 13. No. I really don't understand what

> this is. 8? Yet, more values is... A longer list is not going to help me. I just don't understand the code.

When the interviewer prompted them to try to explain the relationships between variables, they begin to understand PythonTutor's visualizations.

> **Interviewer:** try to relate all the parts together. Try to compare, let's say the list B to list and compare that to the actual line of code there and see if that helps at all. ... The problem I'm seeing here is that you're not kind of, like, relating the parts together. ... You're not relating those two variables together
> **Participant 3:** So we got list B range variable D and I get D equals list X minus X plus 1. Minus the absolute value of X plus 1. Wait a second. Hold on. So it's just the differences between these two. Is it not? ... Now it counts the amount of zeros, so the output will be 2. No, it's only 1. I don't understand why it's only 1. Oh, because it changes? No. Yeah, it becomes the greatest one. Okay, I see. I understand now. It becomes the greatest difference. So variable C is the greatest difference from list. And the whole function returns the greatest difference in the list. And, then, however many of the two numbers next to each other equal the greatest difference in the list.
> **Interviewer:** How did you go about figuring this out now?
> **Participant3:** So what you said pretty much helped me comparing list B to list. Because I wasn't really sure what list B was counting. And then once I started looking at lists. right here, list B, and then looking at lists, I was thinking of just list B here. Once I started looking at the difference, I started noticing. That's basically it. So I think I got that one.

Upon prompting, they explain the purpose of the mathematical expressions present in variable assignments, which is that it finds consecutive differences, then relate it to the find the maximum pattern.

8 DISCUSSION & CONCLUSION

Similar to prior work on metacognitive strategies and tool scaffolds, the effectiveness of PythonTutor's scaffolds in our study appeared heavily influenced by the students' active engagement with the tool. Prior work on metacognitive scaffolding show that it only improves learning outcomes if students actively engage and respond to presented scaffolds [23, 26, 47, 54]. Additionally, prior work on visualization tools found that higher engagement with visualizations significantly improves individual learning outcomes [6, 9, 17].

This aligns with our findings, where participants who stepped through the code *slowly* and through *all* lines using PythonTutor succeeded in understanding the code, while those who skipped steps or did not attempt to understand variable relationships (not carefully analyzing its visualization) struggled. In conclusion, we should encourage students to interact thoughtfully with these tools, asking them to reflect on their understanding, whether they are carefully stepping through the code when needed, and whether they are carefully analyzing PythonTutor's visualization.

In the sections that follow, we discuss findings and present four recommendations of how they could impact instruction, in Sections 8.1-8.4. In Section 8.5, we conclude by suggesting the shape of an integrated approach for teaching students to independently understand code.

8.1 Teach Students to Leverage Commonly Available Tools to Understand Code

Often our participants were successful understanding code when given access to code execution and debuggers. This demonstrates that even novices are much more likely to be successful independently understanding code when given access to these widely available tools. This means that in introductory programming classes, instructors should be attentive towards teaching their students to utilize these tools not only within the course but additionally remind them to use these tools independently beyond the class. Such instruction can ensure that our students can learn to independently read, comprehend, and learn about programs in the long run.

8.2 Teach Students to Cautiously Double Check their Understanding

While most of our interviewed participants were successful understanding code using the tools, we occasionally had participants who failed to understand code despite being given access to the tools. These participants often failed because they did not carefully attempt to double check their understanding of the code. These participants often attempted to understand the code in the quickest way possible using execution, appearing to lack a willingness to step through the code carefully using debuggers.

Some unsuccessful participants appeared overconfident in their incorrect understanding of the code, not feeling the need to double check using the debugger. This may be a sign of participants lacking metacognition or failing to accurately assess their understanding and skills [15]. Our successful participants, on the other hand, appeared cautious to double check their understanding, either targeting to understand specific lines of code using PythonTutor or their understanding of the output using code execution.

Participants who lacked the willingness to carefully step through the code may have not been motivated to walk through this tedious process, often expressing they wanted to quickly understand the code and mentioning they did not want to go through over e.g., 100 steps to understand it. Ko et al. [30] observed that students can be intimidated by debuggers because of their overwhelming interfaces and the lack of instruction on how to effectively use them. While a few of our participants also seemed overwhelmed and initially intimidated by the tedious process of using a debugger, they often quickly got acquainted with and successfully understood the code upon prompting from the interviewer to step through the code carefully using PythonTutor.

8.3 Teach Students to Carefully Step Through Nested Loops

We find that while code execution alone was generally sufficient for students to successfully understand code, debuggers became

increasingly more effective on code with higher cyclomatic complexity levels, indicating that debuggers become more useful on code with more nested loops. This result may be less surprising as code with nested loops tend to have less obvious input-output patterns to our participants. This indicates that we should teach novices to be extra cautious to carefully step through code with nested loops as viewing the output alone may not make the execution behavior of nested loops apparent.

This finding may be supported by the Cognitive Complexity of Computer Programs (CCCP) framework [11], which indicates that code that consists of a greater number of programming plans (common programming patterns) present, particularly when operations done by different patterns are interleaved, tend to be more complex to comprehend. Therefore, debuggers, the more detail-oriented tool, become more necessary for helping students understand these complex patterns. This may indicate that we should teach students to recognize code that appears complex (e.g., nested loops) and to be self-aware of when they should take a more careful, step-by-step approach toward comprehending such complex loops.

8.4 Teach Students to Familiarize with Code then Target Specific Misunderstandings

We additionally observe students strategically utilizing both tools to help them understand our most complex programs, such as initially familiarizing with the code with either PythonTutor or code execution and then completing their understanding using the other tool. Expert programmers are known to apply more bottom-up oriented strategies for understanding code they are less familiar with [45], decomposing code into chunks, tracing, and grouping chunks together. On the other hand, for code they recognize, they apply more top-down oriented strategies, creating and confirming initial hypotheses of what the code achieves. We may view our students' problem solving process in a similar way, striking a balance between both efficient and detail-oriented methods to understand code.

They begin with establishing some understanding of the code's execution behavior, then further solidify their understanding to recognize its intent. Future work should investigate teaching strategies to help students both mitigate cognitive overwhelm of complex programs but still be capable of understanding code they do not recognize, especially as novices are very likely to encounter code they do not recognize.

8.5 We can Teach Novices Techniques to Comprehend Programs While they Lack Advanced Schemas, Potentially Developing Schemas During the Process

One of the reasons experts excel at understanding code is because they have extensive knowledge of programming plans (schemas) from prior experience, allowing them to efficiently recognize familiar programs [49]. Novices, on the other hand, lack or have fragile knowledge of programming plans, failing to recognize and apply them across differing syntactical implementations [55], leading them to struggle to understand code. As we found in this paper,

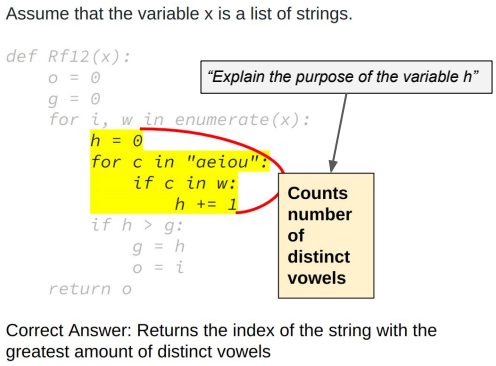


Figure 19: The role of variables intervention get students to identify sub-goals to understand within programs, in the form of lines of code pertaining to variable operations.

```
(pre-)   (first)   (second)        (third)
c = 0    c = 1     c = 2           c = 3
x = 0    x = S1    x = S1 + S2     x = S1 + S2 + S3
return S1 + S2 + S3 / 3
```

Student: is it an **average** of the values?

Figure 20: The abstract tracing intervention help students understand relationship between variables, by noting down values as uncollapsed symbols.

we could teach novices to make use of debuggers and code execution to understand and learn about unfamiliar programs while they currently lack expert-level schemas that can take a long time to develop, potentially allowing them to develop schemas independently throughout the process.

While it is beneficial to teach students common programming patterns/plans, we should additionally teach them strategies to understand programs that they are not familiar with. Across popular platforms such as GitHub, as well as within numerous industrial companies, there exists a vast expanse of domains and topics. This diversity leads to an almost infinite array of programming plans/patterns [7]. Thus novices are likely to encounter new programming patterns that have not been included in their coursework. We end this paper by proposing a starting framework of interventions beneficial for both novices and educators towards helping novices, who lack advanced plan knowledge, to comprehend programs and potentially learn new plans in the process.

We have discovered that debuggers may be more useful than code execution in situations the program is more complex and when the student is less familiar with the code, implying that more detailed, line-by-line methods of comprehension may be more useful in situations the code is unfamiliar and more complex. This aligns with prior code comprehension research on how experts perform a line-by-line, concrete trace of code they do not recognize [10], and prior eye-tracking studies where eyes transitions more rapidly on more complex, unfamiliar programs [29, 48]. Thus, we should instruct novices to understand unfamiliar programs using debuggers.

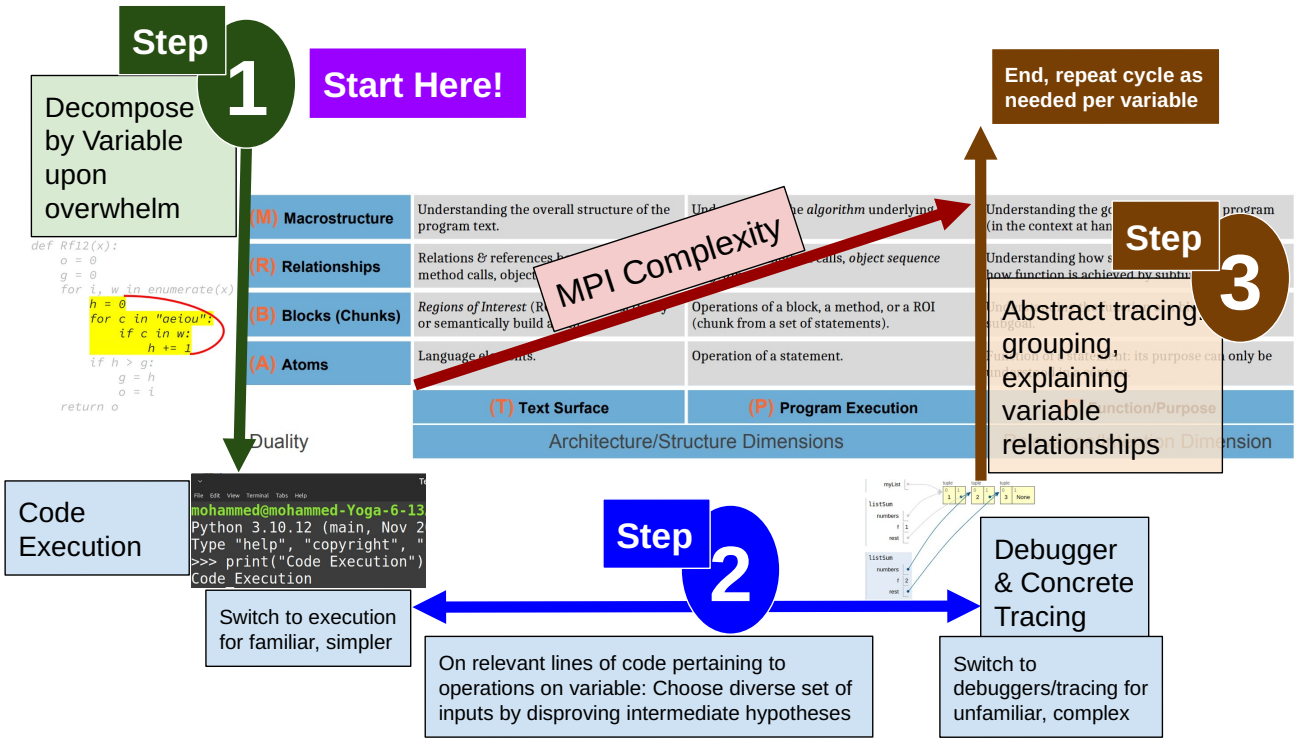


Figure 21: Proposed framework of code comprehension interventions for novices, grounded in the block model [27] and the Cognitive Complexity of Computer Programs (CCCP) framework [11]. MPI stands for Maximal Plan Interactivity [11], which is a part of the CCCP, a code complexity metric based on the number of programming plans present and their level of interleaving, and the programmer's level of familiarity with the code. Debuggers may help novices understand code with higher MPI levels (unfamiliar and complex). The role of variables intervention allows for novices to focus on smaller atoms/chunks to mitigate overwhelm caused by complex programs, and can work their way up to understand relationships between smaller chunks.

As we found in this paper, utilizing debuggers on large, complex programs can be overwhelming. Experts are known to decompose programs, comprehend individual segments, and relate segments together in a bottom-up fashion in situations the program is unfamiliar to mitigate cognitive overwhelm [45]. In our prior work [19–22], we have identified a set of interventions that students can learn to independently apply to decompose code and explain relationship between program segments:

- Role of Variables [19]: Understand programs one variable at a time to mitigate cognitive overwhelm, possible sub-goal labeling technique (Figure 19).
- Abstract Tracing [19]: Understand high-level relationships between variables in cases concrete tracing alone is insufficient (Figure 20).
- Concrete Tracing [19, 20, 22]: Mental line-by-line execution with inputs, assuming a sufficiently diverse set of inputs selected.

As shown in our proposed framework in Figure 21, novices should start by breaking down the code into smaller chunks or atoms, which can mitigate cognitive overwhelm caused by overly complex programs they don't recognize (step 1). Then, for these smaller chunks or atoms, novices can use code execution if these atoms seem familiar and simple. If the chunks are complex or unfamiliar, they should use debuggers. They can strategically switch between these methods based on their comfort level and the complexity of the code (step 2). After understanding individual segments, novices should attempt to explain the broader purpose of the comprehended segment, relating it to other dependent or related variables (step 3). Afterwards, repeat these three steps as needed on other variables, grouping together relationships to build a more comprehensive understanding of the code.

Students should reflect on their familiarity with the program to decide whether to use execution or debuggers, practicing self-monitoring as part of metacognition [37]. Additionally, they should rigorously verify their understanding by attempting to disprove their intermediate hypotheses about code segments, which is another form of self-monitoring [19]. This approach aligns with constructivist teaching principles [1], where students reflect on how their prior knowledge and experience interact with the current problem they are solving. These strategies can inform future teaching practices and be easily integrated into Intelligent Tutoring Systems, such as those utilizing Large Language Models.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. DUE 21-21424 and Mohammed Hassan's SURGE fellowship at the University of Illinois.

REFERENCES

[1] M Alam. 2017. Constructivism and the classroom curriculum. *The International Journal of Indian Psychology* 5, 1 (2017), 1–29.

[2] Pasquale Ardimento, Mario Luca Bernardi, Marta Cimitile, and Giuseppe De Ruvo. 2019. Reusing bugged source code to support novice programmers in debugging tasks. *ACM Transactions on Computing Education (TOCE)* 20, 1 (2019), 1–24.

[3] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies* 18, 6 (1983), 543–554.

[4] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 255–265.

[5] Binglin Chen, Sushmita Azad, Rajarshi Haldar, Matthew West, and Craig Zilles. 2020. A Validated Scoring Rubric for Explain-in-Plain-English Questions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE)*.

[6] Samantha S Cohen, Jens Madsen, Gad Touchan, Denise Robles, Stella FA Lima, Simon Henin, and Lucas C Parra. 2018. Neural engagement with online educational videos predicts learning performance for individual students. *Neurobiology of learning and memory* 155 (2018), 60–64.

[7] Paolo Dello Vicario and Valentina Tortolini. 2021. Evaluating a programming topic using GitHub data: what we can learn about machine learning. *International Journal of Web Information Systems* 17, 1 (2021), 54–64.

[8] Paul Denny, James Prather, Brett A Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A closer look at metacognitive scaffolding: Solving test cases before programming. In *Proceedings of the 19th Koli Calling international conference on computing education research*. 1–10.

[9] Siti Rosminah MD Derus and Ahmad Zamzuri Mohamad Ali. 2015. Utilizing program visualization in learning hardware programming: Effects of engagement level. In *2015 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 491–496.

[10] Françoise Détienne and Elliot Soloway. 1990. An empirically-derived control structure for the process of program understanding. *International Journal of Man-Machine Studies* 33, 3 (1990), 323–342.

[11] Rodrigo Duran, Juha Sorva, and Sofia Leite. 2018. Towards an analysis of program complexity from a cognitive perspective. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 21–30.

[12] K. Anders Ericsson and Herbert A. Simon. 1980. Verbal reports as data. *Psychological Review* 87 (1980), 215 – 251.

[13] Sue Fitzgerald, Gary Lewandowski, Renee McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116.

[14] Sue Fitzgerald, Renée McCauley, Brian Hanks, Laurie Murphy, Beth Simon, and Carol Zander. 2009. Debugging from the student perspective. *IEEE Transactions on Education* 53, 3 (2009), 390–396.

[15] John H Flavell. 1979. Metacognition and cognitive monitoring: A new area of cognitive–developmental inquiry. *American psychologist* 34, 10 (1979), 906.

[16] Max Fowler, David H Smith IV, Mohammed Hassan, Seth Poulsen, Matthew West, and Craig Zilles. 2022. Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study. *Computer Science Education* (2022), 1–29.

[17] Scott Grissom, Myles F McNally, and Tom Naps. 2003. Algorithm visualization in CS education: comparing levels of student engagement. In *Proceedings of the 2003 ACM symposium on Software visualization*. 87–94.

[18] Leo Gugerty and Gary Olson. 1986. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 171–174.

[19] Mohammed Hassan, Kathryn Cunningham, and Craig Zilles. 2023. Evaluating Beacons, the Role of Variables, Tracing, and Abstract Tracing for Teaching Novices to Understand Program Intent. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*. 329–343.

[20] Mohammed Hassan and Craig Zilles. 2021. Exploring 'reverse-tracing' Questions as a Means of Assessing the Tracing Skill on Computer-based CS 1 Exams. In *Proceedings of the 17th ACM Conference on International Computing Education Research*. 115–126.

[21] Mohammed Hassan and Craig Zilles. 2022. On Students' Ability to Resolve their own Tracing Errors through Code Execution. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. 251–257.

[22] Mohammed Hassan and Craig Zilles. 2023. On Students' Usage of Tracing for Understanding Code. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 129–136.

[23] Matthias Hauswirth and Andrea Adamoli. 2017. Metacognitive calibration when learning to program. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*. 50–59.

[24] Matthew Heinsen Egan and Chris McDonald. 2021. An evaluation of SeeC: a tool designed to assist novice C programmers with program understanding and debugging. *Computer Science Education* 31, 3 (2021), 340–373.

[25] Nanna Suryana Herman, Sazilah Binti Salam, Edi Noersasongko, et al. 2011. A study of tracing and writing performance of novice students in introductory programming. In *International Conference on Software Engineering and Computer Systems*. Springer, 557–570.

[26] Alison Hull and Benedict du Boulay. 2015. Motivational and metacognitive feedback in SQL-Tutor. *Computer Science Education* 25, 2 (2015), 238–256.

[27] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, et al. 2019. Fostering program comprehension in novice programmers-learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. 27–52.

[28] Éva Kállay. 2012. Learning strategies and metacognitive awareness as predictors of academic achievement in a sample of Romanian second-year students. *Cognition, Brain, Behavior* 16, 3 (2012), 369.

[29] Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. 2021. Through (tracking) their eyes: Abstraction and complexity in program comprehension. *ACM Transactions on Computing Education (TOCE)* 22, 2 (2021), 1–33.

[30] Minhyuk Ko, Dibyendu Brinto Bose, Hemayet Ahmed Chowdhury, Mohammed Seyam, and Chris Brown. 2023. Exploring the Barriers and Factors that Influence Debugger Usage for Students. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 168–172.

[31] Amruth N Kumar. 2013. A study of the influence of code-tracing problems on code-writing skills. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 183–188.

[32] Amruth N Kumar. 2015. Solving code-tracing problems and its effect on code-writing skills pertaining to program semantics. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. 314–319.

[33] RF Lister. 2007. The neglected middle novice programmer: Reading and writing without abstracting. *National Advisory Committee on Computing Qualifications* (2007).

[34] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further Evidence of a Relationship Between Explaining, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education* (Paris, France) *(ITiCSE '09)*. ACM, New York, NY, USA, 161–165. https://doi.org/10.1145/1562877.1562930

[35] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Acm sigcse bulletin* 41, 3 (2009), 161–165.

[36] Dastyni Loksa, Amy J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. 2016. Programming, problem solving, and self-awareness: Effects of explicit guidance. In *Proceedings of the 2016 CHI conference on human factors in computing systems*. 1449–1461.

[37] Dastyni Loksa, Lauren Margulieux, Brett A Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2022. Metacognition and self-regulation in programming education: Theories and exemplars of use. *ACM Transactions on Computing Education (TOCE)* 22, 4 (2022), 1–31.

[38] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research*. ACM, 101–112.

[39] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*. 101–112.

[40] Lauri Malmi, Ian Utting, and Andrew J Ko. 2019. Tools and environments. (2019).

[41] Tilman Michaeli and Ralf Romeike. 2019. Improving debugging skills in the classroom: The effects of teaching a systematic debugging process. In *Proceedings of the 14th workshop in primary and secondary computing education*. 1–7.

[42] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin* 40, 1 (2008), 163–167.

[43] Greg L Nelson, Benjamin Xie, and Amy J Ko. 2017. 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*. 2–11.

[44] Nancy Pennington. 1987. Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop, 1987*. 100–113.

[45] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology* 19, 3 (1987), 295–341.
[46] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 41–50.
[47] Siti Nurulain Mohd Rum and Maizatul Akmar Ismail. 2017. Metocognitive support accelerates computer assisted learning for novice programmers. *Journal of Educational Technology & Society* 20, 3 (2017), 170–181.
[48] José Aldo Silva Da Costa and Rohit Gheyi. 2023. Evaluating the Code Comprehension of Novices with Eye Tracking. In *Proceedings of the XXII Brazilian Symposium on Software Quality*. 332–341.
[49] Elliot Soloway. 1986. Learning to program= learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (1986), 850–858.
[50] Elliot Soloway, Beth Adelson, and Kate Ehrlich. 1988. Knowledge and processes in the comprehension of computer programs. *The nature of expertise* (1988), 129–152.
[51] Donna Teague. 2015. Neo-Piagetian theory and the novice programmer. *Diss. Queensland University of Technology* (2015).
[52] Donna Teague, Malcolm Corney, Alireza Ahadi, and Raymond Lister. 2013. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In *Proceedings of the 15th Australasian Computing Education Conference [Conferences in Research and Practice in Information Technology, Volume 136]*. Australian Computer Society, 87–95.
[53] Anne Venables, Grace Tan, and Raymond Lister. 2009. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the fifth international workshop on Computing education research workshop*. 117–128.
[54] Ye Wang. 2019. Study of Metacognitive Strategies' Impacts on C Language Programming Instruction. In *2nd International Conference on Contemporary Education, Social Sciences and Ecological Studies (CESSES 2019)*. Atlantis Press, 112–116.
[55] Renske Weeda, Sjaak Smetsers, and Erik Barendsen. 2023. Unraveling novices' code composition difficulties. *Computer Science Education* (2023), 1–28.
[56] Jacqueline Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P K Ajith Kumar, and Christine Prasad. 2006. An Australasian study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Eighth Australasian Computing Education Conference (ACE2006)* (2006).
[57] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. Novice Reflections on Debugging. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 73–79.
[58] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Andrew J Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253.
[59] Craig Zilles, Matthew West, Geoffrey Herman, and Timothy Bretl. 2019. Every university should have a computer-based testing facility. In *Proceedings of the 11th International Conference on Computer Supported Education (CSEDU)*.