On Students’ Ability to Resolve their own Tracing Errors through Code Execution

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

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

ABSTRACT

When students attempt to solve code-tracing problems, sometimes students make mistakes as they read code that get in the way of correctly solving the problem. In this paper, we explore the degree to which students can correct their errors by executing the provided code on a computer. Specifically, we performed a qualitative between-subjects think-aloud study to compare what kinds of errors students can resolve by just executing the code versus which they can resolve by using a line-by-line debugger.

From observing our participants, two factors appear to be necessary for them to independently resolve their errors. First, they need to be using a tool that provides visibility into the error itself. When using a tool that provided only the output of the code, our participants could only resolve dataflow-oriented errors. In contrast, when given the ability to step through the code, some of our participants could additionally resolve control-flow errors. Second, the error must affect the output. In all of the cases where students arrived at the correct answer in spite of having errors in their understanding of the code, none corrected their error independent of the tool they were using. Presumably, they were not forced to confront their error because of an incorrect answer. Finally, while necessary, these conditions appear not to be sufficient, as students still need to be able to correctly interpret the information that the tool provides.

1 INTRODUCTION

Tracing code is often seen as a precursor skill to abstracting or understanding the purpose of code, and as a precursor to writing code [22, 29, 30, 32, 33, 40, 43]. In a tracing question, students are given code and asked to mentally execute the code in their minds. Code tracing questions are widely used in introductory programming courses and are well-studied in the literature [10, 11, 31, 42, 45].

As part of a prior research study, we conducted a qualitative think-aloud study where students solved tracing questions [21]. After their first attempt at executing the code “in their minds,” they were allowed to execute the code with an interpreter where students had an opportunity to independently understand mistakes. Our participants exhibited several errors in their understanding of the code as they solved our tracing problems. They were consistently able to correct some errors by executing the code of the Tracing question but failed to correct other errors even after viewing the output from execution. While misconceptions in programming are well-known in the literature (e.g., [1, 3, 5, 6, 8, 9, 14, 15, 23–25, 27, 28, 34, 35, 37–39, 41, 42]), we were curious why only some errors were understood with the output of the code.

To further understand cases when the output of the code was not sufficient for students to independently address their errors, we conducted an additional set of think-aloud interviews where students were allowed to use line-by-line debuggers after their first attempt solving our tracing questions. The goal of our study is to understand how the debugger is (or is not) more helpful than the output alone for students to independently resolve their errors. Therefore, our research questions are:

- RQ1: What type(s) of tracing errors can students independently resolve with an interpreter?
- RQ2: What type(s) of tracing errors can students independently resolve only with a debugger?
- RQ3: Why can students resolve only some types of tracing errors independently?

As part of this paper, we present a number of errors that we observed. We do not claim that the errors are novel misconceptions, but to the best of our knowledge, there is no prior work that compares how students independently understand their tracing errors differently between debuggers and interpreters.

2 PRIOR WORK

Novice programmers Tracing coding, debugging code, and their potential misconceptions are well-studied in the literature. Below we will highlight the findings of some of these studies.

Alqadi et al. conducted a study quantitatively evaluating the performance of students independently fixing semantic errors within...
a given C++ program with and without access to code execution. Students struggled the most with fixing the "loop has no body" error (e.g., misplaced semicolon after for loop) without access to code execution, but struggled the least with this error with access to code execution. The researchers speculate this is due to the C++ compiler directly giving a warning on this type of error [2].

Novice programmers struggle more with finding bugs than fixing them [17, 26]. In the study of Fitzgerald et al., novices reported in post-session interviews that the easiest bugs to fix were those found by compilers and other tools. A few participants in their study 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 [17]. Certain bugs that are more closely tied to the logic of programs can be more difficult to find [17, 18]. For instance, errors related to variable assignments are difficult to find as variables are largely tied to the logic of programs [18]. Novice programmers often lacked systematic strategies toward locating bugs. For instance, novices opt to work arbitrarily with print statements and repeatedly apply random fixes. [44].

As tracing code is often regarded as a precursor to writing and abstracting code [22, 29, 30, 32, 33, 40, 43], tracing is a crucial skill that novice programmers must learn before they can reliably fix bugs. Sowloay et al. found that if higher-level strategies toward understanding a program fail, programmers resort to concrete line-by-line tracing [13]. Past qualitative think-aloud studies of tracing on paper found that students who correctly kept track of the values of variables as they change per line of code in tracing problems tend to perform better than students who do not [10, 11, 31, 42, 45].

Novice programmers’ struggles are very well-known in the literature. For example, Vainio et al. identified a variety of misconceptions students have when tracing code. For instance, students performed ‘single value tracing,’ where students kept track of only one variable in any program, and assigned the value of the most recently assigned variable just to that one variable [42]. Other potential misconceptions about variables include reversing the variable assignment direction, interpreting assignment as equality [39], interpreting assignments as copying instead of reference [41], interpreting assignments symmetrically like in mathematics [15], and so on [1, 12, 34, 38]. Misconceptions about other programming constructs are also well-studied in the literature (e.g., [3, 15, 25, 27, 39]), such as confusing return for print [3], returning outside functions [25], passing unevaluated expressions as parameters, and incorrectly interpreting Boolean expressions [39].

To the best of our knowledge, there is no prior work that compared how students independently understood their potential tracing misconceptions differently between debuggers and interpreters. The line-by-line step-through nature of debuggers may allow us to better understand how the output was insufficient for students to understand some of their errors, how errors resolvable by the output differ from unresolvable errors, and the visual information needed for students to independently understand their errors. In this paper, we explore how students understood their mistakes with access to code execution compared to line-by-line debuggers. The code given to our students had no bugs, but rather, our participants had errors in their understanding of the code. Our study is qualitative to better understand how and why the debugger and the output are helpful for students to independently understand their errors.

3 METHODS

We conducted a series of think-aloud interviews to observe how students (the participants) solved tracing questions with allowed access to an interpreter as compared to a line-by-line debugger. We followed the protocol of Ericsson et al. for conducting think-aloud interviews where we asked participants to verbalize only their thought process without translating it for our benefit to minimize fatigue and third factors [16]. If participants were silent for more than 2-3 minutes, they were reminded to think aloud.

The 31 participants were traditional aged undergraduate students (18 males, 13 females, age range 18-23) who had completed an introductory programming course in Python for non-Computer Science majors during the Spring 2021 and Fall 2020 semesters. With IRB permission, we recruited these participants through an email sent to the class roster. Each interview was approximately one hour and participants were compensated with a $15 gift card. For this paper, we focused our analysis on 10 of these interviews. The other interviews did not exhibit errors relevant to this work. We conducted the study as a between-groups design as a follow-up to our preliminary work on students tracing code [21]. The earlier 5 of these interviews were part of the preliminary work conducted after the Fall 2020 semester, where participants who have taken the course that semester were allowed access to an interpreter. The later 5 of these interviews were conducted after the Spring 2021 semester, where participants from both the Fall 2020 and Spring 2021 semesters were allowed access to a line-by-line debugger. All participants were given approximately equivalent sets of 11 tracing questions, questions order was varied.

The interviews were recorded over Zoom due to COVID-19, then transcribed and analyzed independently by two researchers, who met to discuss differences in interpretations. We inductively coded the data, accounting for 1) The students’ initial understanding and answer exhibiting error(s) before using the interpreter or debugger, 2) the students’ understanding of an error after using the interpreter or debugger, 3) if using a debugger, how the student stepped through the code back and forth line-by-line, 4) and how the student compared their initial answer to the output of the interpreter or state of the debugger. The two researchers independently identified themes from the codes and reconciled differences to produce a final list of themes.

An external coder re-coded the dataset to ascertain inter-rater reliability. Cohen’s k was used to measure the inter-rater reliability. For the computation, we examined whether raters had assigned the same codes to the participants’ quotes of their think-aloud before and after using the tool [36]. The codes were 1) whether a participant understood their error or did not address it (addressed, addressed incorrectly, unaddressed), 2) whether the participant’s answer demonstrated the error (yes or no), and 3) line(s) of code within the program that is most relevant to the error. Cohen’s k was found to be in high agreement (k = 0.86). TODO: Our coding scheme and tracing questions are publicly available at the following website: https://zilles.cs.illinois.edu
4 RESULTS: DEBUGGERS VS INTERPRETERS

We begin by describing three types of errors related to the participants’ understanding of the dataflow of the program. All participants were able to independently understand their dataflow-related errors using the interpreter (viewing the output of the code alone), and we describe how the output was sufficient in these cases. Next, we talk about a control-flow-related error about a participant confusing the `insert` function as replace, then the fourth element 13 will be missing (replaced by 8).

4.1 Errors Resolved with Interpreters

Our participants that used interpreters could reliably resolve dataflow-related errors that affect their initial answer (i.e., their answer is different from the correct output). We consider an error to be dataflow-related if participants perceive an incorrect value for at least one variable due to misunderstanding either the programming language semantics or constructs, as demonstrated by their think-aloud. The different values between the participants’ answers and the output gave direct hints of how the participants’ mental model of the code’s execution behavior regarding the language semantics is incorrect. We describe the dataflow-related errors below.

4.1.1 Confusing Insert with Replace in a List. The list.insert method inserts a new element into the list at a specified index, and if there is already an existing element at the specified index, then that element (and all later elements) gets shifted by one index. Some of our participants appeared to confuse the list.insert method with a "replace" method, resulting in the preexisting element at the specified index being missing from their answer (see Figure 1 for an example of the function and this error). Our participants seemed to easily understand the difference between their answers demonstrating this error and the correct output, where they seem to notice that the existing element was missing from their answer but present in the output. For example, participant 1 had this error, and after reading the output, they appeared to point out that 13 is still present in the list, then they correctly explained their error.

"You’re going to replace it. ... So this is the list ([3, 3, 7, 7, 8], initial answer), and I’m going to check it (runs code, output: [3, 3, 7, 7, 8, 13]). It includes (13) (pause) it inserts k before that (13)."

4.1.2 Confusing Index Numbers with Value at Index Within a List. Some of our participants appeared to confuse the value of a list element at a specified index (e.g., 11[2] in Figure 2) with the index number (e.g., 2). Understanding the difference between answers demonstrating this error and the correct output seemed easy for our participants, where they appeared to notice the difference between the index number(s) within the list of their initial answers compared to the value(s) at an index within the output list. Participant 2 made this mistake and after reading the correct output, they appeared to point out the element that was different from their initial answer, and then they correctly explained the error.

"Why is that 12 it should be ... 2 because that’s the second index? Oh, because, OK, so you’re inserting the second item of the second index into the second position, so that’s why it’s returning, so that’s why it’s 12 again."

4.1.3 Confusing Modifying Existing List with Creating a New List. Participant 3 was solving a tracing question (Figure 3) that involves a function invocation with a list as one of the input arguments. In the function, a new list is initialized, values are appended to the new list, and then that new list is returned. Participant 3, among some other participants, appeared to incorrectly treat the function as modifying the input list rather than appending values to a new list. What differentiates answers demonstrating this error from the correct answer are the extra element(s) originally from the input (e.g., 2 in this example) that are not present in the correct output. This difference in dataflow is shown directly in the output. Our participants who appeared to notice the element missing from the output (but present in their incorrect answer) universally understood this error after viewing the output alone. Participant 3 viewed the output, appeared to point out the statement that appends to a new list, and then correctly explained their error.

... 14 20 2. (initial answer) (runs code) Oh 14 20 ... Oh ...
We’re only ... appending. Oh yeah, that makes sense.
We’re not editing the actual list, we created a new list.
So then we’re only adding in those numbers. So yeah, just 14 20

4.2 Errors Resolved with Debuggers

Our participants that had access to interpreters seemed to struggle to resolve control-flow-related errors. We consider an error to be control-flow-related if participants followed an incorrect order of
4.2.1 Non-terminating & Multiple returns

Control flow. While debuggers can display the control flow behavior of return statements, some participants seemed unable to understand their non-terminating return error even after stepping through the return statement. For example, as participants 7 & 8 stepped through return statements, they appeared surprised to see that the program terminated. Then, they attempt to align their understanding with the correct output of the program by constructing an incorrect reason based on line(s) of code less relevant to the return statement.

Participant 7 incorrectly associated the output based on if/elif conditionals and indentation, rather than the control-flow behavior of the return statement.

Participant 7: (tracing Figure 4’s code) (Steps line-by-line until return “Y”) ... “Yep, so we return Y” (Tries to step through again, but the program is terminated) “That was it?” “Oh, I guess it’s like you only return ‘I’ if none of..."
these statements (prior if & elif conditionals) are True.
... “I guess the return would have to be like here” (gestures mouse at return “I” being unindented).

**Participant 8** incorrectly associated the output based on a line of code calling the `list.insert` method, appearing to think that the method can be called only once.

**Participant 8:**
*(tracing Figure 1’s code)*

(Initial Answer: `[2, 6, 7, 7, 9, 7, 10]`)
(Correct Answer: `[2, 6, 7, 7, 9, 10]`)

*(Steps through return 1i statement)*

“I don’t get why you wouldn’t put [an extra 7 before 10] (points at 10 in the output) ... (pause)”

“oh wait ... Is it just the first one where you insert it? (points at first element that is greater than k, the first instance the if is True)”

*(Steps back and forth repeatedly through last iteration then return 1i again)*

“Ok. I guess you’d only insert it once.”

Despite the debugger directly showing the control-flow behavior of the return statement, these participants still could not recognize the correct semantical meaning of the return statement. Rather, they made false assumptions about less relevant parts of the program.

### 5.2 Error Not Confronted because it does Not Affect the Output

In other cases, the participants’ errors did not cause them to get the wrong answer. This leads them to not notice the error even when stepping through the code with the debugger.

#### 5.2.1 Non-terminating returns, but the return statement is executed only once thus irrelevant to output

**Participant 9** was solving a problem nearly identical to Figure 1. They appeared to initially make a minor mistake of confusing the less than symbol (<) for greater than (>), and as a result, the if condition would be True for every element of the list but the last. By being unaware that return statements exit the function, the participant appeared to mentally execute the `list.insert` statement multiple times. They caught their mistake of swapping the less than symbol after stepping through the line if `k < li[1]`, and as a result, the if conditional is True only at the last element of the list, hiding the non-terminating return error from their next answer since the return is executed only once. This leads to them solving the problem correctly. Therefore, they were not forced to confront their error.

*(Initial answer: `[8, 3, 8, 3, 8, 7, 8, 7, 13]` demonstrating non-terminating return error and swapped `<)*

*(On debugger: stepped through if k < li[1])*

“k is less than (pause) oh yea it (<) was backwards.”

*(Corrects answer: `[3, 3, 7, 7, 8, 13] )* 

*(Then quickly steps through the line [return li] silently)*

Correct answer:

def f(a, p, r):
    if (a and p >= 79):
        return "B"
    elif (r < 100 and p >= 26):
        return "M"
    elif (r < 38):
        if (a):
            return "F"
        return "C"
    return "I"

print(f(True, 64, 6))

Figure 5: The participant seemed to treat the first if conditional as `False`, which is correct. However, their explanation was because ‘true is not a number’ due to the error shown in Figure 6. They proceeded to step through the line silently with the debugger until the return “‘Y’” statement, then arrive at the correct answer without addressing the error.

Correct:

Incorrect, confusing Boolean ‘and’ with English:

![Diagram showing the flow of the code](image)

Figure 6: Example of confusing the Boolean `and` operator with the English distributive ‘and.’ e.g., Saying “John and Bob did their homework” means “John did his homework” and “Bob did his homework,” but this does not apply to Boolean and in programming.

#### 5.2.2 Confusing Boolean and with English distributed ‘and,’ but the if conditional is `False` thus irrelevant to output

Some of our participants were unable to address a error related to the Boolean and operator after viewing the output (Figure 6). Participants seemed to think that the compound Boolean expression evaluated to False due to an error.

*(after reading output of code in Figure 4, attempting to interpret first if conditional)*

**Participant 10:** “This first if statement didn’t work I’m guessing because it’s comparing False and a number, but then it just moves down straight to this second elif statement.”

**Participant 2:** “This is not True clearly because you can’t have a ... Boolean and numerical. ... That can’t work.”

**Participant 4:** “The first condition doesn’t work because is False and 75, so this would be if False and 75 is greater than or equal to 25 ... Since it’s False here, what exactly does that mean?”

We observed a case where **Participant 7**’s error of the ‘and’ operator did not affect their answer, leading them to skip addressing the error with the debugger. They were solving a second version of the question in Figure 4, shown in Figure 5. They seemed to misinterpret the and operator on the second line of the function in Figure 5 ‘if (a and p >= 79):’. As they stepped through the code, they skipped over the line ‘if (a and p >= 79):’ since it
evaluated to False and were therefore not forced to confront the error.

*(Before using the debugger)*

"This first part [if (a and p >= 79):] is definitely not right because true is like not a number."

*(On debugger: quickly steps through line [if (a and p >= 79):] silently)*

6 DISCUSSION

Participants seemingly can independently resolve errors only if the tool being used provides visibility toward the logic of the error. Below we will revisit our research questions.

**RQ1:** What type(s) of tracing errors can students independently resolve with an interpreter? To answer RQ1, participants who used the interpreter to view the output of code could resolve only dataflow-related errors. The difference between the participants’ incorrect answer and the correct output explicitly revealed the differing mental representations of the dataflow from the error to the correct execution behavior and language semantics. In contrast, our participants were never successful in understanding control-flow-related errors by viewing the output alone. Even if the participants’ incorrect answers differed from the correct output, the output does not display the control-flow behavior of the program.

**RQ2:** What type(s) of tracing errors can students independently resolve only with a debugger? To answer RQ2, some participants were able to address control-flow-related errors with line-by-line debuggers which expose the control-flow behavior (order of execution) of the language directly.

**RQ3:** Why can students resolve only some types of tracing errors independently? To answer RQ3, participants never addressed control-flow errors in cases the error did not affect their answer. Since the error does not interfere with solving the tracing problem correctly, participants did not have to confront the error and did not seem to notice the error. Participants were also unable to address errors related to understanding compound Boolean expressions, even with debuggers. While debuggers provide visibility for the control-flow behavior of the program, it does not show how an expression is parsed sub-expression by sub-expression. Rather, line-by-line debuggers step through each line of code as a whole, immediately moving on to the next line. Thus, debuggers do not provide visibility toward errors related to understanding how expressions are parsed.

Participants who did not resolve errors always constructed a semantically incorrect explanation of the output of the program. Often, the explanation was related to line(s) of code that were not relevant to the error. This aligns with prior work on novice vs expert programmers’ behavior fixing bugs, where experts narrowed down to more relevant parts of the program pertaining to the bug [4, 7, 19].

7 LIMITATIONS

Like any study, this work has obvious limitations. The sample population is relatively small, as with most qualitative studies, due to the large amounts of data to analyze per interview and time commitments to coordinate interviews. Our participants are self-selected, which might mean they are among the higher performers in the course due to their self-confidence to participate in a study that involves skills learned from the course.

Also, as the study design is between-subjects, there may be external factors involving the skill level of individual participants when comparing participants using the interpreter to participants using the debugger. We did not get an opportunity to interview the same participants from the prior study (interpreter condition) with the second condition (debuggers). Both populations demonstrated errors similarly, giving us some confidence that they could be compared.

8 CONCLUSIONS & FUTURE WORK

In this paper, we explored how students independently resolved their errors when given access to an interpreter (i.e., view the output of code) compared to a line-by-line debugger on ‘find the output of code’ tracing questions. Our participants were only able to resolve errors independently if the error affected their initial answer and if the tool they use provides visibility into the logic of the error. We found that for dataflow-related errors, students were consistently able to resolve their errors with both interpreters and debuggers. For control-flow-related errors that affect students’ answers, only debuggers were helpful as the step-through nature of debuggers directly shows the control-flow behavior of programs. For errors that do not affect students’ answers, students bypassed addressing their errors.

Our results suggest that showing students the correct answer to a tracing question is not enough for students to understand their own tracing errors. Thus, it is potentially detrimental to give students answer keys to tracing questions even after solving the problem. Students may think they understood the output of tracing questions, but their thought-process might be for semantically incorrect reasons. Rather, the answer key needs to at least define the relevant programming language semantics, or students should be asked to explain their understanding of the correct output as part of the homework assignment. Offering students a debugging interface in homework assignments can be especially beneficial for students to learn the control-flow behavior of programs in an applied learning manner. However, debuggers alone are not necessarily sufficient toward understanding errors, as debuggers do not show how expressions are parsed sub-expression by sub-expression, and students must correctly interpret what they see. Tools like PythonTutor [20] may benefit from having a feature to step through lines of code sub-expression by sub-expression.

It may be interesting to explore and classify how high- or low-level are students’ explanations of programs as they step through the program with debuggers compared to interpreters, and possibly compared to neither. Although not discussed in this paper due to space limitations, we found certain cases where students using debuggers gave more concise, higher-level explanations of programs on the dataflow-related errors.

9 ACKNOWLEDGMENTS

This work was supported in part by Mohammed Hassan’s SURGE and graduate college fellowships at the University of Illinois.