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 misunderstandings 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 misunderstandings. 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.

CCS CONCEPTS

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

KEYWORDS

tracing, debuggers, errors

ACM Reference Format:
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 V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499400

SIGCSE 2022, March 3–5, 2022, Providence, RI, USA

ACM ISBN 978-1-4503-9070-5/22/03…$15.00
https://doi.org/10.1145/3478431.3499400

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 to determine one or more output values. Code tracing questions are widely used in introductory programming courses and are well-studied in the literature [10, 11, 31, 42, 45].

In a prior research study, we obeserved students solving tracing questions both with and without access to an interpreter [21]. When they had access to the interpreter, many students would first solve the problem without the interpreter and then check their answer using the interpreter. On occasion, our participants exhibited misunderstanding of the code as they solved the problems in their heads. Some of these misunderstandings could be characterized as misconceptions (e.g., [1, 3, 5, 6, 8, 9, 14, 15, 23–25, 27, 28, 34, 35, 37–39, 41, 42]), while others might be more accurately considered mis-readings of the code or merely failures to correctly understand the code.

What sparked our interest was that they appeared consistently able to correct some kinds of these misunderstandings by observing the output of an execution of the given code but not others. 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 a line-by-line debugger after first attempting to solve the tracing questions in their head. The goal this study is to understand how the debugger is (or is not) more helpful than the output alone for students to independently resolve their misunderstandings. Specifically, 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 we are not aware of their presentation in previous publications.

2 PRIOR WORK

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

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. Soloway 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], `return`ing outside functions [25], passing unevaluated expressions as parameters, and incorrectly interpreting `Boolean` expressions [39].

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 [17]. 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. 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 non-systematically with `print` statements and repeatedly apply random fixes. [44].

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 most closely related work is Alqadi et al.'s study of students independently fixing semantic errors within a given C++ program with and without access to code execution [2]. 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.

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 Fall 2020 and Spring 2021 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; question 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, documenting 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.

To ascertain inter-rater reliability [36], an external (non-author) coder re-coded the quotes from the participants' think-alouds using the first author's codes. The codes were: 1) whether a participant addressed their error (addressed, addressed incorrectly, unaddressed), 2) whether the participant's answer demonstrated the error (yes or no), and 3) identifying the line(s) of code within the program that are most relevant to the error. Cohen's κ was used as a measure of the inter-rater reliability. The external coder's codes and the first author's codes had high agreement ($\kappa = 0.86$). Our coding scheme and tracing questions are publicly available at the following website: http://hdl.handle.net/2142/112810

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 (i.e., from 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 `return` statements that participants could not understand by using the interpreter, but could sometimes understand by using a line-by-line debugger. We describe how the debugger was more helpful than just the output. Finally, we talk about two cases where both the debugger and the interpreter were insufficient.

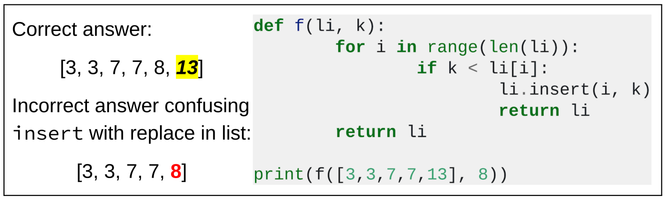

Figure 1: This function `inserts` 8 at the fourth index, but if a participant confuses the `insert` function as replace, then the fourth element 13 will be missing (replaced by 8).

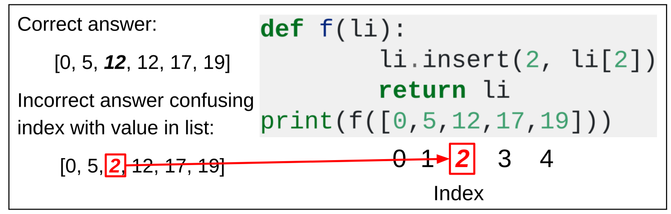

Figure 2: This function inserts 12 at the second index, but if a participant confuses the index 2 for the value 12, then they will insert 2 instead.

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 in spite of correctly tracing the sequence of lines of code. 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., `li[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 12 again."

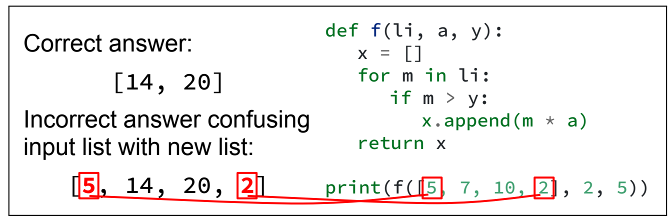

Figure 3: The extra elements 2 and 5 from the incorrect answer is originally from the input list, but since 2 and 5 are not greater than 5, it should not be appended to the list at all. Thus, there is confusion between modifying the current input list vs appending to the new list `x`.

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. Ok 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 execution due to a misunderstanding related to a construct that alters the order of execution (e.g., `return` terminates a function).

Correct answer:

"Y"

Incorrect answer with non-terminating return:

"Y", "I"

```
def f(a, p, r):
   if (a and p >= 25):
      return "B"
   elif (r < 150 and p >= 25):
      return "Y"
   elif (r < 50):
      if (a):
         return "F"
      return "C"
   return "I"

print(f(False, 75, 132))
```

Figure 4: Since the `return "I"` statement is outside any conditional, participants demonstrating the non-terminating `return` error appeared to believe that the `string "I"` is always `return`ed regardless of prior `returns` executing.

Control-flow errors can affect the output, but often in indirect ways. Upon viewing just the output, participants made false assumptions about the programming language semantics in an attempt to explain the output. Students making false assumptions about semantics and syntax is well documented in the literature (e.g., [24]). In some cases, however, participants that used line-by-line debuggers seemed to resolve this type of error as soon as they stepped through relevant control flow.

4.2.1 Non-terminating & Multiple `returns`*.* In a tracing problem with multiple conditionally executed `return` statements (Figure 4), many participants appeared unaware of the control flow behavior of the `return` statement. These participants seemed to believe that the function would `return` multiple times, `return`ing multiple values (the strings "`Y`" and "`I`"). After viewing the interpreters' output (only the string "`Y`"), all of these participants incorrectly attributed their mistake to a misunderstanding related to `if/elif` semantics rather than the `return` statement.

> **Participant 2:** "Now I understand that it doesn't even go to `return 'I'` so that means something about my knowledge about `elif` is wrong. It must be that it's one of the `if` statements. And then that's it. It exits 'cause it just finds out one of the conditions that satisfies it and that's it."
> **Participant 10:** "This first if statement didn't work and this last elif statement, didn't work, so the only thing that did work is this elif statement in the middle, which is why we return Y."

While all participants did not seem to understand this error with the output alone, some participants understood the error using a debugger. Stepping through the code shows that the `return` statement exits the function as soon as the `return` statement is executed. Once these participants stepped through the `return` statement, they appeared to have an "Aha!" moment and gave a correct explanation of their error (that `returns` exit functions). In these cases, the debugger explicitly displayed the control-flow behavior of `return` statements, rendering debuggers more useful than the output alone.

> (Once stepped through `return` statement)
> **Participant 6:** "Oh ok. ... You would return it, so the program would end right there."
>
> **Participant 5:** "Huh? Oh! ... [Integrated Development Environment Name] literally gives you the ... OK, because this is return so it would break off the loop." (question in Figure 1)

5 RESULTS: ERROR NOT RESOLVED WITH DEBUGGERS OR INTERPRETERS

In this section, we describe two cases where participants appeared unable to resolve their errors with neither debuggers nor interpreters. The first case is when the participants made false assumptions about line(s) of code not relevant to the error even after using a debugger. The second case is when the error does not affect the participants' answer, causing them to bypass addressing the error. We observed the first case with the non-terminating `return` error. We observed the second case with two errors: 1) the non-terminating `return` error and 2) a new error related to compound `Boolean` expressions with the `and` operator.

5.1 Debuggers Alone are Not Enough Without the Right Interpretation

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 5's code)*
> (Initial Answer: `[2, 6, 7, 7, 9, 7, 10]`)
> (Correct Output: `[2, 6, 7, 7, 9, 10]`)
> *(Steps through* `return li` *statement)*
> "I don't get why you wouldn't put [an extra 7 before 10] (points at 10 in the output) ... (pause)"

Correct answer:

`[2, 6, 7, 7, 9, 10]`

Incorrect answer with non-terminating `return`:

`[2, 6, 7, 7, 9, 7, 10]`

```
def f(li, k):
    for i in range(len(li)):
        if k < li[i]:
            li.insert(i, k)
            return li
    return li

print(f([2, 6, 7, 9, 10], 7))
```

Figure 5: The participant seemed to believe that the `return` statement can be executed more than once, leading them to append an extra element 7.

> "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 li` 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 and is irrelevant to output. **Participant 9** was solving a problem nearly identical to Figure 1. They appeared to initially make the 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[i]`, 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[i]`*)*
> "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)*

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.

Correct:

`(False and 75 >= 25)` → `(False and True)` → `False`

Incorrect, confusing Boolean 'and' with English:

`(False and 75 >= 25)` → `(False >= 25 and 75 >= 25)`

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.

Correct answer:

"Y"

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

print(f(True, 64, 6))
```

Figure 7: 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.

> *(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 7. They seemed to misinterpret the `and` operator on the second line of the function in Figure 7 '`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 into 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? Participants who used the interpreter to view the output of code resolved 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? 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? 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 used 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. 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.

REFERENCES

[1] Alireza Ahadi, Raymond Lister, and Donna Teague. 2014. Falling Behind Early and Staying Behind When Learning to Program.. In *PPIG*, Vol. 14.
[2] Basma S Alqadi and Jonathan I Maletic. 2017. An empirical study of debugging patterns among novices programmers. In *Proceedings of the 2017 ACM SIGCSE*

technical symposium on computer science education. 15–20.

[3] Austin Cory Bart, Teomara Rutherford, and James Skripchuk. 2020. Evaluating an Instrumented Python CS1 Course.. In *CSEDM@ EDM*.

[4] Roman Bednarik. 2012. Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *International Journal of Human-Computer Studies* 70, 2 (2012), 143–155.

[5] Yorah Bosse and Marco Aurélio Gerosa. 2017. Why is programming so difficult to learn? Patterns of Difficulties Related to Programming Learning Mid-Stage. *ACM SIGSOFT Software Engineering Notes* 41, 6 (2017), 1–6.

[6] Yorah Bosse, David F Redmiles, and Marco Gerosa. 2019. Connections and Influences Among Topics of Learning How to Program. In *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–8.

[7] 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.

[8] Ricardo Caceffo, Steve Wolfman, Kellogg S Booth, and Rodolfo Azevedo. 2016. Developing a computer science concept inventory for introductory programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 364–369.

[9] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. 2014. Identifying challenging CS1 concepts in a large problem dataset. In *Proceedings of the 45th ACM technical symposium on Computer science education*. 695–700.

[10] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using tracing and sketching to solve programming problems: replicating and extending an analysis of what students draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. 164–172.

[11] Kathryn Cunningham, Shannon Ke, Mark Guzdial, and Barbara Ericson. 2019. Novice rationales for sketching and tracing, and how they try to avoid it. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 37–43.

[12] Saeed Dehnadi. 2009. *A cognitive study of learning to program in introductory programming courses*. Ph.D. Dissertation. Middlesex University.

[13] 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.

[14] Dimitrios Doukakis, Maria Grigoriadou, and Grammatiki Tsaganou. 2007. Understanding the programming variable concept with animated interactive analogies. In *Proceedings of the The 8th Hellenic European Research on Computer Mathematics & Its Applications Conference (HERCMA'07)*.

[15] Benedict Du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.

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

[17] 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.

[18] John D Gould. 1975. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies* 7, 2 (1975), 151–182.

[19] 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.

[20] Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 579–584.

[21] 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.

[22] 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.

[23] Cruz Izu and Peter Dinh. 2018. Can Novice Programmers Write C Functions?. In *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. IEEE, 965–970.

[24] Lisa C Kaczmarczyk, Elizabeth R Petrick, J Philip East, and Geoffrey L Herman. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*. ACM, 107–111.

[25] Maria Kallia and Sue Sentance. 2019. Learning to use functions: The relationship between misconceptions and self-efficacy. In *Proceedings of the 50th ACM technical symposium on computer science education*. 752–758.

[26] Irvin R Katz and John R Anderson. 1987. Debugging: An analysis of bug-location strategies. *Human-Computer Interaction* 3, 4 (1987), 351–399.

[27] Cazembe Kennedy and Eileen T Kraemer. 2018. What Are They Thinking? Eliciting Student Reasoning About Troublesome Concepts in Introductory Computer Science. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. 1–10.

[28] Cazembe Kennedy and Eileen T Kraemer. 2019. Qualitative observations of student reasoning: Coding in the wild. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 224–230.

[29] 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.

[30] 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.

[31] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin* 36, 4 (2004), 119–150.

[32] 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.

[33] 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.

[34] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. 2007. Investigating the viability of mental models held by novice programmers. In *Proceedings of the 38th SIGCSE technical symposium on computer science education*. 499–503.

[35] Davin McCall and Michael Kölling. 2014. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. IEEE, 1–8.

[36] Cliodhna O'Connor and Helene Joffe. 2020. Intercoder reliability in qualitative research: debates and practical guidelines. *International Journal of Qualitative Methods* 19 (2020), 1609406919899220.

[37] Wolfgang Paul and Jan Vahrenhold. 2013. Hunting high and low: Instruments to detect misconceptions related to algorithms and data structures. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 29–34.

[38] Takayuki Sekiya and Kazunori Yamaguchi. 2013. Tracing quiz set to identify novices' programming misconceptions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*. 87–95.

[39] Teemu Sirkiä and Juha Sorva. 2012. Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*. 19–28.

[40] 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.

[41] Donna Teague and Raymond Lister. 2014. Longitudinal think aloud study of a novice programmer. In *Conferences in Research and Practice in Information Technology Series*.

[42] Vesa Vainio and Jorma Sajaniemi. 2007. Factors in novice programmers' poor tracing skills. *ACM SIGCSE Bulletin* 39, 3 (2007), 236–240.

[43] 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.

[44] 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.

[45] Benjamin Xie, Greg L Nelson, and Andrew J Ko. 2018. An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 344–349.