



# Investigating the Utility of Self-explanation Through Translation Activities with a Code-Tracing Tutor

Maia Caughey and Kasia Muldner<sup>(✉)</sup>

Department of Cognitive Science, Carleton University, Ottawa, Canada  
{maiacaughey,kasiamuldner}@cunet.carleton.ca

**Abstract.** Code tracing is a foundational programming skill that involves simulating a program's execution line by line, tracking how variables change at each step. To code trace, students need to understand what a given program line means, which can be accomplished by translating it into plain English. Translation can be characterized as a form of self-explanation, a general learning mechanism that involves making inferences beyond the instructional materials. Our work investigates if this form of self-explanation improves learning from a code-tracing tutor we created using the CTAT framework. We created two versions of the tutor. In the experimental version, students were asked to translate lines of code while solving code-tracing problems. In the control condition students were only asked to code trace without translating. The two tutor versions were compared using a between-subjects study ( $N = 44$ ). The experimental group performed significantly better on translation and code-generation questions, but the control group performed significantly better on code-tracing questions. We discuss the implications of this finding for the design of tutors providing code-tracing support.

**Keywords:** Code Tracing Tutor · Self-Explanation · Translation from Python to English

## 1 Introduction

What skills do students need to be taught when learning to program? To date, the emphasis has been on program generation (i.e., teaching students how to write computer programs). To illustrate, the majority of work in the AIED community and/or computer science education focuses on supporting the process of program generation or aspects of it (e.g., [2, 8, 11]). Program generation is certainly a core skill, but it is only one of several competencies listed in theories of programming education [26]. A foundational skill proposed in Xie et al.'s framework [26] is code tracing. Code tracing involves simulating at a high level the steps a computer takes when it executes a computer program (including keeping track of variable values and flow of execution through the program). The high-level goal of our research is to design tutoring systems that help students

learn to code trace. As a step in this direction, here we describe two alternative designs of a tutoring system supporting code tracing and the corresponding evaluation. We begin with the related work.

### 1.1 Code Tracing: Related Work

Code tracing helps students learn the mechanics of program execution, which promotes understanding of the constructs making up the program [19]. Tracing on paper is positively correlated with programming performance [16, 17, 24] and there is experimental evidence that teaching students to code trace first helps them to subsequently write programs [3]. However, many students find code tracing challenging. Some report not knowing where to start [27]. Others report avoiding code tracing altogether, resorting to suboptimal methods instead [7]. When students do code trace, their traces are often incomplete [6, 9] and/or contain errors [9]. Thus, support for code tracing is needed.

One way to help students learn to code trace is with tutoring systems. In our prior work [13], we implemented CT-Tutor, which was designed to provide practice with code tracing through problems. To test the effect of scaffolding, we created two versions of the problem interface: (1) the high-scaffolding interface guided code tracing by requiring students to enter intermediate variable values and providing feedback on these entries; (2) the reduced-scaffolding interface only provided an open-ended scrap area. In both versions, for each problem CT-Tutor provided a corresponding example, shown either before the problem or after (based on condition). Students learned from the tutor but there was no significant effect of either scaffolding or example order. However, when the analysis included only students who learned from using the tutor, an interaction between scaffolding and example order emerged. When students were given the high-scaffolding interface, learning was highest when the example came *after* the problem, but the opposite was true for the reduced-scaffolding interface (more learning if the example came *before* the problem). CT-Tutor aimed to mirror the process of code tracing on paper. Another way to code trace is with a debugger (these are often included in program development environments). Nelson et al. [18] developed a tutoring system called PL-Tutor that like a traditional debugger showed the program state at each program-execution step, additionally prompting students to self-explain during various parts of the code-tracing process. PL-Tutor was evaluated by comparing learning from the tutor and Codecademy materials. There was no significant difference between the two conditions but students did learn from using PL-Tutor.

Other work focuses on evaluating debugging and/or program visualization tools. While access to debuggers can be beneficial [10, 20], these are traditionally designed for students with some programming experience, since they use technical terms (e.g., stacks, frames). A potential limitation of these tools is that they do all the code-tracing work, and so students may fail to learn how to do it on their own without the help of the tool.

Other work focuses on the design of code-tracing examples (rather than problems as we do in the present work). Hosseini et al. [12] created a tutoring system

that provided animations of code traces. The animations showed a visual trace of the program as well as the stack frame with values of variables. A second type of example included in the tutor was static, providing written explanations about code traces without animations. Students had access to both examples. Accessing animated examples was positively associated with higher course grades, while accessing static examples was associated with lower grades. Kumar [14] evaluated a tutoring system that presented a code-tracing example after an incorrect code-tracing solution was submitted. Students in the experimental group were prompted to self-explain the example; the control group did not receive prompts. There was no significant difference in gain scores between the prompted and unprompted groups, perhaps because tutor usage was not controlled (this study was done in a classroom context and students worked with the tutor on their own time, so some may have devoted little effort when answering the prompts).

As the summary above shows, obtaining significant effects related to code-tracing instructional manipulations is challenging, highlighting the need for more research. In general, existing support for code tracing focuses on showing the mechanics of program execution and state. However, in order to code trace a program, the programming language syntax and semantics need to be understood. This is not trivial for novices [21]. Accordingly, theories of programming instruction propose that translation is an essential component of learning to program [26]. Translation involves converting programming language syntax into an explanation of what the statement does in a human language. The translation grounds the code in more familiar language, which should free cognitive resources needed for performing the code trace. Translation can be characterized as the general mechanism of self-explanation [4], because it requires inferences over and beyond the instructional materials and because these inferences should be beneficial for performing the code trace.

Note that translation of individual programming statements or lines differs from describing what an entire program does. There is work on the latter aspect [17,25]. To illustrate, Whalley et al. [25] evaluated the reading comprehension skills of novice programmers. Participants were asked to explain what programs do at a high level. The prompts targeted entire programs or subsections that involved multiple program lines. In contrast, during code tracing, students read line-by-line to trace variable values and program behavior and so smaller program components are involved.


## 2 Current Study

To the best of our knowledge, work in AIED and beyond has not yet evaluated whether incorporating explicit support for line-by-line translations in a tutoring system promotes learning of code tracing. To fill this gap, we conducted a study to answer the following research question:

**RQ:** *In the context of a tutoring system supporting code tracing, does translation of programming language syntax into a human language (English in our work) prior to code tracing improve learning?*

Simulate the execution of this program by:  
 (1) entering a translation for each program line  
 (2) if there is an update to a variable related a line translated, update the Code Trace table  
 (3) finish updating the Code Trace table by entering the values of variables for each loop iteration.

Python Code:	Translations:	Code Trace:														
1 print("Number 1")	1 Prints the string "Number 1" to the screen	<table border="1"> <thead> <tr> <th rowspan="2">Description/ iteration:</th> <th colspan="2">Variables:</th> </tr> <tr> <th>val</th> <th>res</th> </tr> </thead> <tbody> <tr> <td>before loop</td> <td>3</td> <td>5</td> </tr> <tr> <td>first loop</td> <td>4</td> <td>6</td> </tr> <tr> <td>second loop</td> <td>3</td> <td>3</td> </tr> </tbody> </table>	Description/ iteration:	Variables:		val	res	before loop	3	5	first loop	4	6	second loop	3	3
Description/ iteration:	Variables:															
	val		res													
before loop	3		5													
first loop	4		6													
second loop	3		3													
2 val = 5	2 Assign the value 5 to the variable val															
3 res = 10	4 Assign the value 10 to the variable res															
4 while True:	Complete the lines of code inside the loop															
5 if res < 5:	Check if the variable res is less than 5															
6 break	Exit the loop															
7 val = val - 1	Lower the value of the variable val by 1															
8 res = res - val	Lower the value of the variable res by the value															
9 print(val, res)	Print the value of the variables val and res to the															



**Fig. 1.** The translation tutor interface with the full solution entered. The Python program to be code traced appears in the leftmost panel; the translations of the program from Python syntax to plain English are in the middle panel; the values of the program variables for a given loop iteration are in the right-most panel that contains the code-trace table. Notes: (1) the orange bubbles are for illustrative purposes and were not shown to students - see text for their description; (2) the solution to the translation, not shown here for space reasons, appears to the right of the code-trace table. (Color figure online)

To answer this question, we created two versions of an online tutoring system supporting code tracing of basic Python programs: (1) a translation tutor, and (2) a standard tutor. Both versions were created using CTAT [1]. CTAT is a tutor-building framework that facilitates tutor construction by providing tools to create the tutor interface and specify tutor behaviors. Both versions scaffolded the process of code tracing, but only the translation tutor required students to self-explain the meaning of the program being code traced.

### 2.1 Translation Tutor vs. Standard Tutor

The translation tutor presented one code-tracing problem per screen. Each screen included a brief Python program (Fig. 1, left), a translation panel used to enter plain English translations of the program (Fig. 1, middle), and a code-trace table used to input values of the program’s variables during program execution (Fig. 1, right). To solve the problem, for each program line, students had to first self-explain the line by translating it into plain English. The translation was required

before the code trace to encourage reflection about the meaning of that line (its semantics) and so reduce errors during code tracing. Once the explanation was provided, students had to code trace the program line by entering relevant variable value(s) into the corresponding input box(es) in the code-trace table.

Figure 1 shows the tutor interface with a completed problem. When a problem is first opened, all the translation and code-trace table input boxes are blank and locked (except for the first translation box next to the first program line, which is blank and unlocked). The input boxes are unlocked in sequence as entries are produced. To illustrate, for the problem in Fig. 1, a student has to enter the translation of the first program line into the corresponding input box (see label 1, Fig. 1), which unlocks the second translation input box (see label 2, Fig. 1). Once the second translation is produced (see translation related to label 2, Fig. 1), the corresponding code-trace table box is unlocked (see label 3, Fig. 1); this process continues, with the next translation and table input boxes unlocked after corresponding answers are generated (see labels 4 and 5, Fig. 1). Note that if a program includes a loop, as is the case for the program in Fig. 1, a translation of a line inside the loop body has to be produced only once (during the code trace of the first loop iteration).

The translation tutor provides immediate feedback for correctness on the entries in the code-trace table, by coloring them red or green for incorrect and correct entries, respectively. To guide solution generation, a correct table entry is required to unlock the next input box. For the translations, due to challenges with natural language parsing, immediate feedback is not provided, and any input submitted for a translation unlocks the next input box. Students can view a canonical translation solution after the entire code-trace problem is completed, by clicking on the purple box used to temporarily hide the translation solution (not shown in Fig. 1 for space reasons). At this point, students can revise their translations if they wish (all translation boxes are unlocked after the code trace-table is correctly completed).

In addition to feedback for correctness, the interface is designed to implicitly guide the code-tracing process in several ways, using tactics from our prior work [13]. First, as described above, the tutor requires students to enter the code trace step-by-step in the logical order dictated by the code-tracing process - cues are provided about this because the input boxes are locked (colored gray) until a step is correctly generated, at which point the color of the next input box changes to indicate it is unlocked and available for input. This design aims to encourage students to enter the entire solution step by step, motivated by the fact that when tracing on paper, students produce incomplete traces and/or skip steps when code tracing [6]. Second, because some students do not know which program elements to trace, the interface specifies the target variables to be traced (see code-trace table, Fig. 1).

The standard version of the tutor is identical to the translation version, except that its interface does not include the translation panel shown in Fig. 1 (middle). Thus, in the standard tutor, students only have to enter the code trace

using the code-trace table. All other scaffolding included in the translation tutor is also provided by the standard tutor.

## 2.2 Participants

The study participants were 44 individuals (37 female, 6 male, 1 demi-femme) recruited using a range of methods: (1) class announcements in a first-year university programming class for cognitive science majors, (2) the SONA online recruitment system available to students in a first-year university class that provided a broad overview of cognitive science, (3) social media advertising via university Facebook groups, and (4) word of mouth. Participants either received a 2% bonus course credit for completing the study, or \$25 compensation. To be eligible, participants either had to have no prior programming experience or at most one university-level programming course.

## 2.3 Materials

Both versions of the tutor were populated with four code-tracing problems. The same problems were used for the two versions; each problem showed a Python program with a while loop (e.g., see Fig. 1). A brief lesson was developed to provide an introduction and/or refresher to fundamental programming concepts. The lesson corresponded to a 20-min video showing a narrated slideshow. The lesson covered variable assignment, integer and string data types, basic conditional statements, and while loops.

A pretest and posttest were created to measure domain knowledge and learning. The tests were isomorphic, with the same number of questions and question content, but different variable names and values. There were three types of questions on the test, namely translation, code tracing, and code generation. Each test showed a series of six brief Python programs - for each program, students were asked to produce (1) a line-by-line translation of the program into plain English, and (2) a detailed code trace. The final question required the generation of a Python program. A grading rubric was developed, with the maximum number of points for each test equal to 44.5.

## 2.4 Experimental Design and Procedure

The study used a between-subjects design, with two conditions corresponding to the two tutor versions described in Sect. 2.1. Participants were assigned to the conditions in a round-robin fashion.

The study sessions were conducted individually through Zoom (one participant per session). The study took no more than two hours to complete. After informed consent was obtained, participants were given the link to the video lesson and were asked to watch it; they could take notes if they wished (20 min). After the lesson, participants completed a brief demographics questionnaire and the pretest (20 min. with a 5 min. grace period). Next, participants took a five-minute break. After the break, they were provided a link to the tutor (either

the translation tutor or the standard tutor). After logging into the tutor website, they were given a five-minute introduction to the tutor by the researcher, including a demonstration on how to use the tutor (this was scripted for consistency between sessions). The demonstration used a problem from the video lesson. Next, participants used the tutor to solve four code-tracing problems. Participants were instructed to complete the problems at their own pace and told they would have 40 min to do so (with a 10-min grace period). After the tutor phase, participants were provided with a link to the posttest (20 min. with a five-minute grace period).

### 3 Results

The primary goal was to analyze if translation activities during code tracing with a tutoring system improved learning. Learning was operationalized by change from pretest to posttest (details below). The tests were graded out of 44.5 points, using a pre-defined grading scheme. Recall there were three types of questions on the test, namely translation, code tracing, and code generation (the latter was the transfer question, as the lesson and tutor did not cover program generation). We conducted separate analyses for each question type because each involved distinct concepts and lumping them together had potential to obscure findings. Initially, there were 22 participants in each condition. For a given question type, we removed from the analysis participants at ceiling at pretest for that question type, so the degrees of freedom will vary slightly.

The descriptive statistics are shown in Table 1. Collapsing across condition and question type, participants did learn from using the tutor, as indicated by the significant improvement from pretest to posttest ( $M_{gain} = 5.52\%$ ,  $SD = 10.24$ ),  $t(43) = 3.57$ ,  $p < .001$ ,  $d = 0.54$ .

Prior to testing conditional effects, for each question type, we checked if there was a difference in the pretest scores between the two conditions using an independent samples t-test. Despite the assignment strategy, overall the translation-tutor group had slightly higher pretest scores, but this effect was not significant for any of the three question types (translation pretest scores:  $t(42) = 1.21$ ,  $p = .233$ ,  $d = 0.37$ ; code-tracing pretest scores:  $t(39) = 1.67$ ,  $p = .103$ ,  $d = 0.52$ ; code-generation pretest scores  $t(42) = 1.49$ ,  $p = .144$ ,  $d = 0.50$ ).

To measure learning, we used normalized gain [5], calculated as follows:

$$\frac{posttest(\%) - pretest(\%)}{100\% - pretest(\%)}$$

Normalized gain characterizes how much a student learned (based on their raw gain from pretest to posttest), relative to how much they could have learned (based on their pretest score). This enables a more fair comparison between groups, particularly in situations where there are differences in pretest scores. This was the case with our data, i.e., the pretest scores for all three question categories were higher for the translation tutor group, albeit not significantly so.

**Table 1.** Mean and standard deviation for pretest, posttest, and raw gain (posttest – pretest) scores (by percentage) for each question type and condition.

Question Type		Standard Tutor $n = 22$	Translation Tutor $n = 22$
		$M$ ( $SD$ )	$M$ ( $SD$ )
Translation	Pretest (%)	66.99 (23.60)	74.88 (19.49)
	Posttest (%)	63.04 (28.71)	78.71 (22.58)
	Raw Gain (post-pre) (%)	-3.95 (15.70)	3.83 (8.11)
Code Trace	Pretest (%)	50.78 (23.43)	63.91 (26.96)
	Posttest (%)	66.75 (31.24)	64.81 (27.82)
	Raw Gain (post-pre)(%)	15.97 (20.93)	0.90 (8.35)
Code Generation	Pretest (%)	23.49 (28.48)	28.95 (31.34)
	Posttest (%)	31.44 (34.69)	51.32 (39.11)
	Raw Gain (post-pre)(%)	7.96 (21.59)	22.37 (28.06)

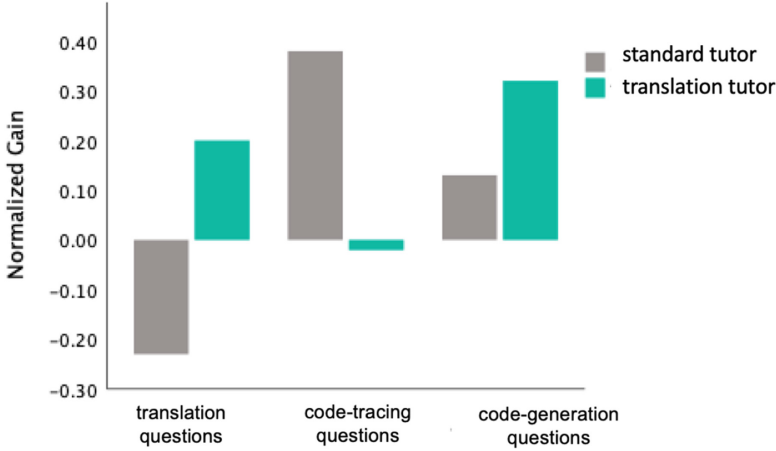
Normalized gain was calculated for each student separately. The average normalized gains for each question type and condition are shown in Fig. 2. A disadvantage for normalized gain is that it is arguably harder to interpret than raw gain (posttest - pretest). Thus, for the sake of completeness, we also report raw gain in Table 1. For the sake of parsimony we only report the inferential statistics for the normalized gain but the pattern of results for each question holds if raw gain is used as the dependent variable.

For the translation questions, the standard-tutor group performed slightly worse on the posttest than on the pretest as indicated by a negative normalized gain, while the translation-tutor group improved from pretest to posttest (see Fig. 2). This effect of tutor type on normalized gain was significant,  $t(31.2) = 2.4$ ,  $p = .025$   $d = 0.78$ ), with the translation-tutor group learning significantly more. The opposite pattern occurred for the code-trace questions, with only the standard-tutor group improving from pretest to posttest; the translation-tutor group had similar pretest and posttest scores (see Fig. 2). This effect of tutor type on normalized gain for the code-tracing scores was significant,  $t(39) = 2.5$ ,  $p = .016$ ,  $d = 0.8$ . For the code-generation transfer question, the translation-tutor group had higher normalized gain but not significantly so,  $t(39) = 1.2$ ,  $p = .251$ ,  $d = 0.37$  (but see below after outliers were removed).

The results show that the effect of tutor type depended on question type (more translation learning with the translation tutor but more code-tracing learning with the standard tutor). This interaction between tutor type and question type was significant as indicated by a mixed ANOVA with question type as the within-subjects factor and tutor type as the between-subjects factor,  $F(2, 74) = 6.57$ ,  $p = .005$ ,  $\eta_p^2 = .30$ .

To ensure the validity of our results, we checked for the presence of outliers. There were no outliers due to data errors but there were several in each condi-





**Fig. 2.** Average normalized gain for each condition and question type (note that normalized gain is calculated differently from the raw gain shown in Table 1)

tion due to normal variation<sup>1</sup>. If outliers are a normal consequence of inherent individual differences, the advocated approach is to re-run the analysis without them and report if the original patterns change. This is advocated over removing outliers that are not due to errors as it portrays a more holistic view of the results. The removal of the outliers did not change the significant effect of tutor type for the translation and code-trace questions (translation:  $t(41) = 2.09$ ,  $p = .043$ ,  $d = 0.64$ ; code-tracing  $t(33.31) = 3.07$ ,  $p = .04$ ,  $d = 0.92$ ). For the code-generation transfer question, the outliers were influential. After they were removed, the translation-tutor group improved significantly more on the transfer question than the standard-tutor group,  $t(27.43) = 2.72$ ,  $p = .011$ ,  $d = 0.88$ .

## 4 Discussion and Future Work

The present paper describes the design and evaluation of a tutoring system supporting code-tracing activities. Overall students did learn from interacting with the tutoring system but the analysis of normalized gain scores broken down by question type revealed a nuanced pattern regarding the effect of tutor type.

We begin with the positive results. Students working with the translation tutor had to generate translations of programming language syntax into plain English. These students had significantly higher normalized gain on translation test questions, compared to the standard-tutor group not required to produce translations. Since translation is a form of self-explanation, our findings replicate prior work showing the benefits of self-explaining [4]. We acknowledge that the

<sup>1</sup> There was 1 outlier in the standard-tutor data for translation scores, 4 outliers in the translation-tutor data for code-tracing scores, and outliers for the code-generation scores (3 in the standard-tutor data and 1 in the translation-tutor data).

translation-tutor group had practice on translation, but explicit support does not always result in significant effects (e.g., [13,14,18]) and so this finding is encouraging. Notably, the standard-tutor group got slightly worse in their translation performance from pretest to posttest. This may have been due to the time gap between the lesson, which included translation information, and the posttest. Due to this gap, the standard-tutor group may have forgotten translation concepts from the lesson. In contrast, the translation-tutor group received support for translation with the tutor, which likely reinforced lesson concepts.

The translation-tutor group also had significantly higher normalized gain on the code-generation transfer question (after outlier removal). This group was scaffolded to generate explanations from Python to plain English. Because these explanations may have made the program more meaningful, this could have facilitated learning of code schemas (i.e., algorithms), which are useful for code generation [22].

We now turn to the unexpected result related to learning of code tracing, namely that the standard tutor group had significantly higher code-tracing normalized gains than the translation-tutor group. In fact, the translation-tutor group did not improve on code-tracing outcomes from pretest to posttest (normalized gain was close to zero). This surprising, as this group had opportunities for code-tracing practice, which based on prior work should increase learning [15,18]. One possible explanation for this finding is that the translation tutor required translation at each step and this may have increased cognitive load [23], which interfered with the code-tracing process. In particular, translations added an extra feature students had to pay attention to, and cognitive load theory predicts that splitting a student's attention between features can hinder learning. Other factors to explain why the translation-tutor group had low code-tracing performance include the timing of translations, effort, and feedback. The timing of the translations may have not been ideal - perhaps if the tutor prompted for translations at a different point in the instructional sequence, rather than concurrently during the code-trace activity, learning would have occurred. Moreover, writing translations requires effort. This increased effort may have led to fatigue, leaving less resources for the code trace. Finally, the feedback for the translations was delayed until after the code trace due to logistics reasons and came in the form of a canonical solution showing the "ideal" translations. This format requires students to invest effort into processing the feedback (e.g., comparing their translations to ones in the canonical solution), something that they may have been unmotivated to do.

In conclusion, the answer to our research question on whether translation activities improve learning depends on the outcome variable of interest (translation, generation, code tracing). Requiring translations to be completed concurrently with a code-trace activity produced a modest but significant boost for normalized gain related to translation questions; translation also improved normalized gain related to code generation, once outliers were removed. However, translation hindered learning of code tracing. To address this issue, future work needs to investigate alternative designs, including other instructional orderings.

For instance, perhaps students need to master the skill of translation from a given programming language to their native human language before *any* code tracing occurs. Another avenue for future work relates to providing guidance for translations. In the present study feedback in the form of a canonical solutions was provided after the code trace - perhaps novices require feedback earlier. Work is also needed to generalize our findings. In our study, the majority of participants were female. This was partly a function of the fact that about a third of the participants came from a first year programming class required for all cognitive science majors in our program (even ones not focused on computer science), and this class has a higher proportion of female students. In general, however, programming classes are increasingly required for non-computer science majors, and so research is needed on how to best support students from varied backgrounds.

**Acknowledgements.** This research was funded by an NSERC discovery grant.

## References

1. Alevan, V., et al.: Example-tracing tutors: intelligent tutor development for non-programmers. *Int. J. Artif. Intell. Educ.* **26**(1), 224–269 (2016)
2. Anderson, J.R., Conrad, F.G., Corbett, A.T.: Skill acquisition and the Lisp tutor. *Cogn. Sci.* **13**(4), 467–505 (1989)
3. Bayman, P., Mayer, R.E.: Using conceptual models to teach basic computer programming. *J. Educ. Psychol.* **80**(3), 291 (1988)
4. Chi, M.T., Bassok, M., Lewis, M.W., Reimann, P., Glaser, R.: Self-explanations: how students study and use examples in learning to solve problems. *Cogn. Sci.* **13**(2), 145–182 (1989)
5. Coletta, V., Steinert, J.: Why normalized gain should continue to be used in analyzing preinstruction and postinstruction scores on concept inventories. *Phys. Rev. Phys. Educ. Res.* **16** (2020)
6. Cunningham, K., Blanchard, S., Ericson, B., Guzdial, M.: Using tracing and sketching to solve programming problems: replicating and extending an analysis of what students draw, pp. 164–172 (2017)
7. Cunningham, K., Ke, S., Guzdial, M., Ericson, B.: 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, ITiCSE 2019*, pp. 37–43. Association for Computing Machinery, New York (2019)
8. Fabric, G.V., Mitrovic, A., Neshatian, K.: Evaluation of parsons problems with menu-based self-explanation prompts in a mobile python tutor. *Int. J. Artif. Intell. Educ.* **29** (2019)
9. Fitzgerald, S., Simon, B., Thomas, L.: Strategies that students use to trace code: an analysis based in grounded theory. In: *Proceedings of the First International Workshop on Computing Education Research, ICER 2005*, pp. 69–80 (2005)
10. Hoffswell, J., Satyanarayan, A., Heer, J.: Augmenting code with in situ visualizations to aid program understanding. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018*, pp. 1–12 (2018)
11. Hosseini, R., et al.: Improving engagement in program construction examples for learning Python programming. *Int. J. Artif. Intell. Educ.* **30**, 299–336 (2020)

12. Hosseini, R., Sirkiä, T., Guerra, J., Brusilovsky, P., Malmi, L.: Animated examples as practice content in a Java programming course. In: Proceedings of the Technical Symposium on Computing Science Education, SIGCSE 2016, pp. 540–545 (2016)
13. Jennings, J., Muldner, K.: When does scaffolding provide too much assistance? A code-tracing tutor investigation. *Int. J. Artif. Intell. Educ.* **31**, 784–819 (2020)
14. Kumar, A.N.: An evaluation of self-explanation in a programming tutor. In: Trausan-Matu, S., Boyer, K.E., Crosby, M., Panourgia, K. (eds.) ITS 2014. LNCS, vol. 8474, pp. 248–253. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-07221-0\\_30](https://doi.org/10.1007/978-3-319-07221-0_30)
15. Lee, B., Muldner, K.: Instructional video design: investigating the impact of monologue- and dialogue-style presentations, pp. 1–12 (2020)
16. Lister, R., Fidge, C., Teague, D.: Further evidence of a relationship between explaining, tracing and writing skills in introductory programming, pp. 161–165 (2009)
17. Lopez, M., Whalley, J., Robbins, P., Lister, R.: Relationships between reading, tracing and writing skills in introductory programming, pp. 101–112 (2008)
18. Nelson, G.L., Xie, B., Ko, A.J.: Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in CS1. In: Proceedings of the ACM Conference on International Computing Education Research, pp. 2–11 (2017)
19. Perkins, D.N., Hancock, C., Hobbs, R., Martin, F., Simmons, R.: Conditions of learning in novice programmers. *J. Educ. Comput. Res.* **2**(1), 37–55 (1986)
20. Pérez-Schofield, J.B.G., García-Rivera, M., Ortin, F., Lado, M.J.: Learning memory management with C-Sim: a C-based visual tool. *Comput. Appl. Eng. Educ.* **27**(5), 1217–1235 (2019)
21. Qian, Y., Lehman, J.: Students’ misconceptions and other difficulties in introductory programming: a literature review. *ACM Trans. Comput. Educ.* **18**(1) (2017)
22. Soloway, E., Ehrlich, K.: Empirical studies of programming knowledge. *IEEE Trans. Softw. Eng.* **10**(5), 595–609 (1984)
23. Sweller, J.: Cognitive load theory. *Psychol. Learn. Motiv.* **55**, 37–76 (2011)
24. Venables, A., Tan, G., Lister, R.: A closer look at tracing, explaining and code writing skills in the novice programmer. In: Proceedings of the ACM Conference on International Computing Education Research, pp. 117–128 (2009)
25. Whalley, J.L., et al.: An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. In: Proceedings of the 8th Australasian Conference on Computing Education, pp. 243–252 (2006)
26. Xie, B., et al.: A theory of instruction for introductory programming skills. *Comput. Sci. Educ.* **29**(2–3), 205–253 (2019)
27. Xie, B., Nelson, G.L., Ko, A.J.: An explicit strategy to scaffold novice program tracing. In: Proceedings of the ACM Technical Symposium on Computer Science Education, pp. 344–349 (2018)