



On Teaching Novices Computational Thinking by Utilizing Large Language Models Within Assessments

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

Paul Denny
University of Auckland
Auckland, New Zealand
p.denny@auckland.ac.nz

Yuxuan Chen
University of Illinois
Urbana, Illinois, USA
yuxuan19@illinois.edu

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

Abstract

Novice programmers often struggle to develop computational thinking (CT) skills in introductory programming courses. This study investigates the use of Large Language Models (LLMs) to provide scalable, strategy-driven feedback to teach CT. Through think-aloud interviews with 17 students solving code comprehension and writing tasks, we found that LLMs effectively guided decomposition and program development tool usage. Challenges included students seeking direct answers or pasting feedback without considering suggested strategies. We discuss how instructors should integrate LLMs into assessments to support students' learning of CT.

CCS Concepts

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

Keywords

Large Language Models, code comprehension, debuggers, execution

ACM Reference Format:

Mohammed Hassan, Yuxuan Chen, Paul Denny, and Craig Zilles. 2025. On Teaching Novices Computational Thinking by Utilizing Large Language Models Within Assessments. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025)*, February 26-March 1, 2025, Pittsburgh, PA, USA. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3641554.3701906>

1 Introduction

Understanding and writing code are essential skills [16], yet novices often struggle to develop them. A multi-national study by the Leeds working group found that most novices fail to acquire these abilities by the end of an introductory programming course [15]. They “lack the ability to take a problem description, decompose it into sub-problems and implement them, then assemble the pieces into a complete solution,” skills central to computational thinking.

Prior research has identified effective code comprehension teaching strategies for novices [7–9, 21]. These techniques have shown

potential for novices to learn and apply them independently over time [21], including decomposing code into sub-goals [21], tracing (mentally executing code), disproving intermediate hypotheses to verify understanding, leading to actions like choosing diverse inputs [9], understanding relationships between variables [21], and using debuggers [8, 20] and code execution tools [7].

In this study, we explore integrating these teaching strategies into a Large Language Model (LLM)-based assistant within assessments and evaluate how they can generalize toward teaching computational thinking skills. Two key motivations drive this approach: First, leveraging LLMs allows us to deploy these strategies at scale in large introductory programming courses. Second, by providing assistance directly within students' working interface, we aim to encourage them to use the provided tools rather than seeking external resources that could be used for cheating.

While prior work explored auto-generating code explanations using GPT [30], this tells students what the code does rather than helping them develop code understanding strategies. When given answers, students may focus on *what* the answer is rather than *how* to obtain it [3]. Code comprehension skills are especially crucial when evaluating the correctness of code generated by LLMs. Therefore, we investigate how students use an LLM-based chatbot that aims to teach computational thinking skills by guiding them to use debuggers and code execution tools embedded on assessment pages. The research question is: *How do novices use an LLM-based assessment tool to help them learn computational thinking skills?*

To answer this, we conducted qualitative think-aloud interviews with 17 introductory programming students solving ‘Explain in Plain English’ (EiPE) and code writing questions, using an LLM-based chatbot present on the question pages (Figure 1), alongside a PythonTutor-like [6] debugger and code execution tools.

2 Related Work

2.1 Computational Thinking

Computational thinking (CT) is the ability to solve problems effectively and efficiently by developing reusable solutions [31]. It involves several interrelated skills: Decomposition (breaking down complex problems into manageable parts), creating algorithms (designing solutions that are efficient, logical, ordered, generalizable, and reusable), debugging (identifying and fixing errors through iterative refinement), abstraction (understanding core actions or



This work is licensed under a Creative Commons Attribution International 4.0 License.

SIGCSE TS 2025, February 26-March 1, 2025, Pittsburgh, PA, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0531-1/25/02
<https://doi.org/10.1145/3641554.3701906>

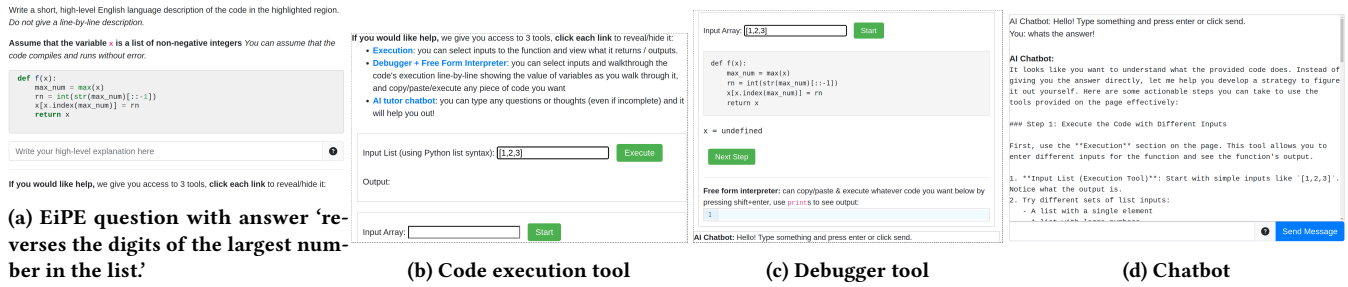


Figure 1: Question assessment page, scrolling from left to right.

concepts within an algorithm), and pattern recognition (identifying similarities and recurring structures across problems). These foundational skills make CT a transferable and valuable approach for addressing diverse problem-solving tasks across disciplines.

2.2 Metacognition

Metacognition is awareness of one's own understanding and problem-solving progress [25, 26], involving self-regulation and self-monitoring [18]. Self-regulation includes planning, monitoring, and adjusting strategies as needed; self-monitoring involves reflecting on strategies used. Novices often lack metacognition [18], shown by reluctance to abandon incorrect solutions, misunderstanding problems [4], and rushing tasks. Scaffolding these skills through reflective prompts and self-assessment can help [4, 19, 28].

However, the effectiveness of metacognitive scaffolding depends on student engagement; active participation is crucial for success [28]. Hauswirth et al. discussed challenges in metacognitive calibration, where students may engage superficially with self-assessment tasks, mechanically rating skills without deeper consideration just to complete the task, which suggests that the design of the tools must promote deeper engagement [10]. Wang et al. emphasized the importance of instructors in arousing students' interest and motivation in learning metacognition to promote engagement [36].

Our prior work [21] found that utilizing Karl Popper's theory of refutation [24] can help novices develop metacognitive skills. Popper posits that knowledge advances by formulating hypotheses and attempting to disprove them. Encouraging students to challenge their intermediate understanding addresses cognitive biases like confirmation bias, the tendency to seek only confirming evidence while disregarding contradictory evidence, which limits understanding. By seeking evidence that challenges their hypotheses, students consider problems more holistically and thoroughly evaluate their problem-solving strategies, leading to actions like choosing more diverse inputs when executing code.

2.3 Programs & Code Comprehension

Programs consist of intrinsic actions, the commands a computer executes, and extrinsic purposes, its broader goals, concepts, and societal meanings [2]. The Block Model [11] classifies code comprehension as understanding syntactical structure, execution behavior (algorithmic structure), and intent (purpose), emphasizing that understanding intent (why it's written) differs from how it executes.

Code comprehension is typically regarded as a mix of top-down (hypothesis-driven) and bottom-up (decomposition or trace-first) processes, depending on the programmer's familiarity with the

code [13, 17, 23, 35]. In the top-down model proposed by Soloway et al., programmers recognize familiar code patterns or 'programming plans' [32]. Programmers can be familiar with concepts, plans, or strategies at differing levels of knowledge: declarative (knowing what something is), procedural (knowing how to perform it), and strategic (knowing when, where, and how to use it). Superficial memorization can lead to a lack of these advanced knowledge types.

2.4 Dialogue-based Tutoring

Conversational intelligent tutoring systems (ITSs) emulate one-to-one human tutoring via text dialogue [29] and can be as effective as human tutors [34]. They have been applied in fields like algebra, physics [33], and code writing [1]. ITSs encourage students to provide information in as much detail as possible to diagnose their thought processes, using dialogue moves like hints and assertions [29]. Allowing students to guide the conversation aligns with constructivism, enabling them to construct knowledge by applying prior knowledge. Tailoring problem-solving paths adjusts to student's zone of proximal development. These systems often use non-deterministic mechanisms like Hidden Markov Models.

Prior work has found that while dialogues in tutoring systems help learners understand material better than static text, increasing the frequency of dialogue interactions does not necessarily improve learning outcomes [12]. Kopp et al. [12] and Nückles et al. [22] found that the most effective approach is a mixed strategy combining full dialogue interactions with limited ones. They speculate that this balanced method maintains the benefits of interactive learning while preventing overload and reducing the risk of boredom or decreased motivation from constant prompting. This suggests there may be an optimal amount of dialogue for certain learning tasks. Most prior research on dialogue-based tutoring systems relies on log data and quantitative methods, not providing insights into when and why students may benefit more from different dialogue modes.

3 Method

We conducted think-aloud interviews with 17 undergraduates (10 men, 7 women) who completed an introductory Python programming course for non-technical majors in Fall 2023. With IRB permission, we recruited them via course roster emails. Each interview was about one hour, where participants solved 10 Python EiPE questions and 5 code-writing questions using a chatbot, a PythonTutor-style debugger, and an execution tool. Following Ericsson et al.'s protocol [5], we asked participants to verbalize their thoughts without translating them for our benefit to minimize fatigue, reminding them to think-aloud if they were silent for more than 2–3 minutes. After

Table 1: Success rates. “Incorrect with chatbot help” includes instances that needed interviewer intervention for the participant to solve correctly

| | Number of Questions |
|--------------------------------|---------------------|
| Correct with no help | 21 |
| Correct with chatbot help | 14 |
| Incorrect with chatbot help | 5 |
| Incorrect, did not use chatbot | 0 |

each problem, the interviewer asked retrospective questions about the chatbot. Participants received a \$35 gift card as compensation.

We recorded the interviews over Zoom, then transcribed and independently analyzed them by two researchers, who met to discuss interpretive differences. We inductively coded the data, focusing on interactions with the chatbot and how student strategies differed throughout the assessment. We applied the constant comparative method to identify emerging themes and used reflexive coding to reflect on our assumptions during analysis.

We guided the GPT-4o-based chatbot behavior using an extensive prompt, similar to approaches in other LLM-based tutors [14]. An excerpt of the prompt (truncated due to space limitations) follows: “Be an intelligent tutor helping the student solve this Python problem. Avoid giving code-specific hints; instead, provide hints for independent understanding of the code. Ask questions like ‘what’s the relationship between `var_a` and `var_b`?’ Give hints on effectively using the given tools. Suggest that students comprehend and execute specific code sub-parts. You can (re)write relevant portions of the code and tell them to copy/execute it. Encourage them to disprove intermediate assumptions and choose diverse inputs using code execution or debuggers.”

We also explained to the LLM what code execution and debuggers were. The chatbot interprets a screenshot of the page with each student message to understand the given problem and the tools. In Table 1, we show the participants’ overall correct rates. In the next sections, we report on instances of the chatbot helping students solve problems, followed by unsuccessful instances.

4 Results: Code Reading Successful Interactions

This section describes instances where the chatbot effectively helped students solve EiPE problems. Some participants felt it was more helpful than traditional ChatGPT, which often provides (sometimes incorrect) answers without helping them understand their problem-solving mistakes, as summarized by participant 16 below:

Participant 16: *It was helpful and easy to understand. It would tell you if you’re in the right direction or not. It would be like, ‘Oh, you’re missing something.’ So I would go back and look through it again to know what I was doing wrong. I thought the AI would use more complex language, but it clearly broke down what I was doing, what I was missing. Regular GPT’s feedback isn’t helpful; it doesn’t go step by step. It just tells you the answer, and sometimes it’s wrong, but I feel like this one was trying to direct you*

In the next sections, we describe different types of assistance that led students to solve EiPE problems correctly. It guided them on how to make better use of debuggers and execution by providing advice like breaking down lines of code, choosing more diverse inputs, defining general syntax, and offering hints that prompted students to ask more specific questions about their misunderstandings.

4.1 Suggestion to use Debuggers & Execution

When students ask the chatbot for answers (e.g., “What does the code do?”), it suggests using the debugger or execution tool.

Participant 6: *I can’t figure out the function so I was going to ask the chatbot to explain [it]. But then the chatbot told me to use the execution [tool]. ... It was a helpful reminder*

4.2 Guidance on How to Use Tools

When students do not effectively use tools, the chatbot provides guidance on how to use the tools by defining individual syntactical elements present within lines of code and suggesting to choose a diverse set of inputs to test each individual element. For example, participant 15 struggled because they chose unhelpful inputs, choosing single-digit inputs that appear the same when reversed.

Participant 15: *I want to try inputting (inputs single-digit numbers for code in Figure 1a). Still return the same [input number as output]. Now I want to ask the chatbot because I’ve tried several [inputs] and they always return the same thing. (asks chatbot ‘rm = int(str(max_num)[: -1]), explain it’).*

The chatbot responds by suggesting to break down the expression and using the execution tool to choose inputs to test individual parts of the expression, suggesting multi-digit integer inputs to be able to observe and understand the reverse slicing operation.

Chatbot: *To understand what the expression ‘rm = int(str(max_num)[: -1])’ is doing, break it down and examine the steps involved. Start by exploring the individual operations being performed. Make use of the Interpreter tool to input the individual parts of the expression and see what they output. For example, you could start by setting ‘max_num’ to a known value, like ‘123’, and then converting it to a string with ‘str(max_num)’ in the interpreter to see the effect. Next, you could take that string result and apply the slicing operation ‘[: -1]’ to see what happens.*

Participant 15: *That chatbot is very helpful. This sentence here is very important because it mentioned the slicing operation. So I changed the test case to a bigger number. So the slicing will apply, and I know what it’s doing here. (answers correctly). At first, I didn’t use a big number, just a single digit number as an input and always return the same thing. But it mentioned slicing. And I just changed it to a larger number, like three digits, like 155 and execute code. Then I can find something different.*

4.3 Signs of Learning to Double-Check Understanding Using Tools Independently

Some participants express that, in later questions, they learned to use the tools more diligently independent of the chatbot after its repeated suggestions.

Participant 1: *The chatbot broke the habit of me going through the debugger once and then giving up. It taught me to look through it again. Just because you do it once and you might still be confused, maybe you’re missing something and you need to go through again*

The participant describes breaking down the code independently and recalling syntax definitions it previously suggested:

I just slowed down and... broke it down even more in my mind with the steps that [the debugger] was giving. And the modulo example from earlier, I already knew. I asked the chatbot earlier what this

did, so I didn't need to... ask it again. I learned from what I did the last time and I'm like, okay, let me just go through and do this again and go even more step-by-step. My brain automatically knew, like, okay, you're stuck. Just look through it closer and do it again

For example, in a later question and without chatbot assistance, participant 1 repeatedly re-uses the debugger trying more inputs:

Participant 1: (using debugger) I kind of got what the end result is and about what it is. But let's just do an easy list again. ... Let's change that to 80. Let me retry this. (answers correctly).

4.4 Guiding Students to Ask Better Questions

In some cases, when students asked vague questions, the chatbot provided suggestions that led them to ask more specific questions to target their specific misunderstandings. For example, after the chatbot suggested to break down a line of code, some students asked more specific questions targeting their initial misunderstanding:

Participant 6: (asks chatbot 'explain this line of code `rn = int(str(max_num)[::-1])`' for question in Figure 1a)

Chatbot: It's essential to start by breaking down the line into each of its components. Investigate what each part does individually. Additionally, use the Interpreter, to create examples with different data and observe the outcome.

Participant 6: looking at the second line, it takes the variable `max_num`, turns it into a string, and then back to an integer. Trying to figure out the indexing because I don't remember what double colon means. Maybe I can ask. (asks chatbot 'What does `::` mean')

After breaking down the line of code, participant 6 asked a follow-up question that is more specific and directly targets their initial uncertainty about the double colon syntax.

Chatbot: In Python, the colon (`:`) is used in slicing to indicate the start and end positions for the slice. When you see `::`, this is a slice operation that includes the entire sequence. Furthermore, when it's used with a number after it, like `::-1`, it represents step size. You can run small tests with different sequences and slice configurations in the Interpreter provided to you to see different outcomes and understand how it behaves.

Participant 6: It finds the position that it's in the list, the maximum-then reverses it, and then it returns the list (answers correctly). The chatbot was helpful in explaining what the double colon means.

5 Results: Code Writing Successful Interactions

We highlight instances where the chatbot assisted with code-writing problems. Participants found it useful for decomposing problems and implementing each step.

Participant 8: It was helpful for writing code since it leads you to the correct thinking process if you break apart the question and ask (the chatbot) each part of the question how to do it. It guides me on how to create code. Without it, it would have been very hard for me

5.1 Breaking up Problem Statement to Concepts

The chatbot helps students break problem statements into meaningful sub-functions with clear purposes. For example, participant 11 was solving a problem that asks whether a given timestamp is a perfect square whose square root is prime. They began by typing their thought process to the chatbot:

Participant 11: (Types to chatbot: "Returns true if number's perfect square and its square root's prime else false.")

Chatbot: Tackle the problem by breaking it down focusing on two tasks: determining if a number is a perfect square and if its square root is prime.

Perfect Square: Write a function that returns 'True' if an integer is a perfect square. Use 'math.sqrt' and check if the result is an integer. **Prime Number:** Write a function that returns 'True' if an integer is prime. Check divisibility up to its square root.

In your main function 'is_prime_square(day, month, year)':

1. Zero-pad day and month to maintain the correct format.
2. Concatenate 'day', 'month', and 'year' into a single number.
3. Check if this concatenated number is a perfect square.
4. If it is, check if its square root is prime.

Test each helper function individually before combining them.

The chatbot describe how to break up the problem into a set of distinct concepts each in its own helper function, and how to piece these helper functions together. The participant is then successful:

Participant 11: So this is a separate function. We'll change this to number equals. Then compare to ... (implements correctly)

5.2 Breaking up Concept to Logical Steps

When students asked it for help on implementing concepts, it provided step-by-step instructions on how to implement it:

Participant 8: I'm going to first ask, because I'm not sure how I can start this problem if I'm not comfortable with prime numbers. How can I check if a number is prime?

Chatbot: To verify if a number is prime, you'll need to check that it has no divisors other than 1 and itself. One way to do this is to try dividing the number by all smaller numbers greater than 1. If none of them divides it evenly, then the number is prime. It's a good practice to write some test cases with known prime and non-prime numbers to check your implementation.

Participant 8: Oh got it (writes code implementing loop for prime).

5.3 Writing Syntax of Constructs Guidance

In cases participants struggled in writing syntax, the chatbot provides guidance in the form of generic code examples of common syntactical constructs. As an example, participant 11 struggled to implement checking prime numbers:

Participant 11: How to loop through possible divisors?

Chatbot: Here's a basic loop function:

```
for i in range(start, end):
    # Perform actions within the loop using the variable 'i'
```

Consider: What's the loop's purpose? What are the start and end conditions? What actions occur inside the loop? Write your loop and test it with different values to ensure it works as intended.

It explains how to use loops, then they correctly implement it:

Participant 11: For i in range 2, integer times times 0.5 (writes correct solution)

5.4 Tailored Sub-goals Leveraging APIs

In response to incomplete or incorrect solutions, the chatbot provides feedback and suggests sub-goals to implement to correct their

solution. For example, participant 11 realized their solution didn't retain the remaining list elements after sorting the first k elements:

Participant 11: Wait, where did it go?

Chatbot: To sort the first k elements of array `arr` in ascending order while keeping the rest of the array unchanged, you can't modify the `first_k_elements` variable that you've extracted from the array. Currently, your code is only slicing the first k elements and storing them in `first_k_elements`, but you're not sorting them. Since the goal is to sort those elements, you'll want to apply a sorting method to `first_k_elements` before returning the modified array.

The chatbot explained the issue and provided steps to correct it:

Chatbot: Here's a strategy to follow:

1. Take `first_k_elements` which you've correctly sliced from `arr`.
2. Sort `first_k_elements` using a sorting method. Python lists have a method called `.sort()` that you can use, which will sort lists in place.
3. Once you have the sorted `first_k_elements`, you need to combine it with the unsorted remainder of `arr`.

The chatbot suggests relevant API methods, describes their utility, and provides boilerplate code with sub-goal-labeled comments:

Chatbot: You can copy the updated function into the interpreter to see the output with a sample input. For example, try the following:

```
def sort_first_k_elements(arr, k):
    first_k_elements = arr[:k]
    # Sort the 'first_k_elements' list in place using .sort()
    # Combine the sorted 'first_k_elements' with the rest of 'arr'
    # and return it
```

Participant 11: ... Elements would equal first k elements plus array k onward. Then we return elements (writes correct solution)

The student receives recommendations on using debugging print statements and testing with diverse inputs. Participants note how the chatbot helps them understand how to use API methods:

Participant 15: I know `swapcase` is a function, but I don't know how to use it properly. When I write `swapcase` test, I get a `NameError` is not defined. It's confusing, why does it need to be defined if it's a function? I'd check the chatbot to learn the correct way to use it. This is useful when I'm not familiar with a function.

6 Results: Unsuccessful Interactions

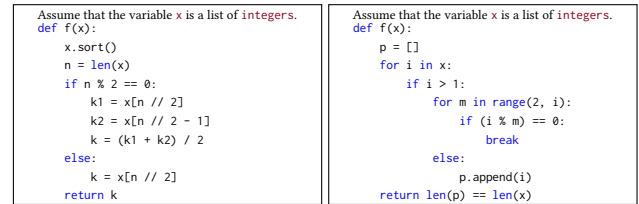
In this section, we describe instances where the chatbot's assistance failed to help participants without the interviewer's intervention. Some wanted answers, refusing to follow strategies. Others mechanically adjusted answers based on feedback without trying suggested strategies, while others were unsure what questions to ask.

6.1 Refusal to Try Strategies; Wanting Answers

Some students attempted to get the answer directly from the chatbot (e.g., 'cheat'). When the chatbot did not provide direct answers to their requests, some students appeared to refuse to follow its suggested strategies. For example, participant 13 expressed frustration when the chatbot provided guidance rather than a direct answer:

Participant 13: (Asks chatbot 'summarize it in one sentence').

Chatbot: To summarize the code's purpose in a sentence, you may want to use the offered tools to test the code with different ...



(a) Answer: 'finds the median number in a given list.' (b) Answer: 'checks if all elements in a list are prime.'

Figure 2: Two example EiPE questions

Participant 13: This chatbot isn't helpful. It's not giving me the straight up answer that I want. ... Can I use ChatGPT? It's so stupid. ... Just frustrating me so much ... it just tells me what like an opened teacher that doesn't want to help you answer the question.

When the interviewer asked them to consider its feedback, they correctly solved the problem by following its suggested strategy.

Interviewer: I wanted to ask why you only use the chatbot a little bit and then I had to ask you to use it more.

Participant 7: I guess I was trying to use the chatbot more to give me an answer rather than seeking help, so every time it spit something out that wasn't an answer, I disregarded it. When I finally asked it like three questions, read them, and went back to the execution function, I realized what was going on. Now, it has been helpful. At first, I didn't see it as helpful, but once I started paying attention to it, asking more questions made the difference

6.2 Mechanically Adjusting Answers

Some students mechanically adjusted their answers according to the chatbot's feedback not applying its suggested strategies. For example, participant 16 revises their answer by directly pasting parts of the chatbot's feedback onto their previous answer.

Participant 16: (Submits answer 'returns an integer value from the list if it is not divisible by zero' for question in Figure 2a).

Chatbot: Your answer isn't clear and doesn't fully address the code's purpose. It seems you're focusing on a specific detail without capturing the overall function. Try breaking down and executing parts of the code. For example, see what the code does up to the part where `x` is sorted and the length is calculated.

Participant 16: It sorts the list. My answer isn't complete (Changes to 'returns integer value from sorted list' which is still incorrect).

6.3 Unsure How to Ask Complete Questions

Some participants who were stuck on solving problems expressed that they were not sure what questions to ask the chatbot. For example, participant 14 demonstrates this uncertainty:

Participant 14: I'm not sure. it's confusing (Figure 3).

Interviewer: Maybe try asking the chatbot questions.

Participant 14: What kind of questions in this case?

When the interviewer asked them to write their immediate thoughts directly to the chatbot, it provided better guidance:

Interviewer: Can you ask that thought you told me to the chatbot?

Participant 14: (asks chatbot 'Why would someone want to do this? What is the mathematical problem to solve?')

Chatbot: To understand the mathematical problem the code solves, you can start by examining the relationships between the variables

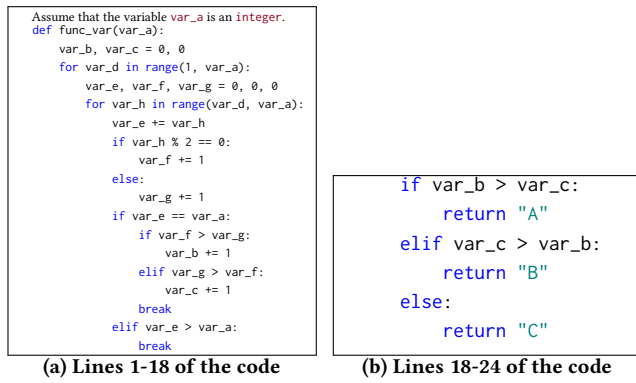


Figure 3: An EiPE question with answer ‘checks whether there are more even or odd consecutive sums from 1 to var_a.’

and the conditions under which they change. Use the Debugger tool to step through the code with different input values for ‘var_a’ and observe how the variables ‘var_b’, ‘var_c’, ‘var_e’, ‘var_f’, and ‘var_g’ are updated. This will help you identify the pattern and purpose behind the calculations.

They solve the problem correctly after following its advice to explain the relationship between variables:

Participant 14: *it’s counting the odds in- it’s trying to see how many of the numbers in the sequence- how many odds or even numbers are greater in a sequence? ... How the sum relates to the input and influences the A, B, or C. So it is summing the sequence ... of numbers (answers correctly)*

These participants appear unaware that they can type their incomplete thoughts or actions they have taken directly to the chatbot, feeling the need to ask only complete questions:

Interviewer: *What parts of the chatbot did you find unhelpful? ... What kind of hints did you wish it gave you?*

Participant 14: *I feel like, I wouldn’t ask- ... I don’t know, I don’t pinpoint ideas with the chatbot. I don’t tell the chatbot about what I’m thinking. I would ask more concrete questions. ... It was weird asking just about what I was thinking.*

Their unawareness of expressing their thoughts or problem-solving process directly to the chatbot can limit its ability to offer targeted assistance on their incorrect problem-solving process.

7 Discussion & Conclusions

Overall, we found the chatbot effective in guiding students to practice computational thinking and metacognition. For code reading, it helped students decompose complex code, define individual actions (abstraction) using general definitions of syntax and API methods, use common development tools, and select input test cases to double-check their understanding, skills crucial for debugging. By guiding students to break programs into sub-actions, understand each component’s role, and describe relationships, it facilitated pattern recognition and helped piece together the overall purpose. These strategies foster transferable skills that students can apply broadly in programming and problem-solving.

For code writing, the chatbot guided students by decomposing concepts into logical algorithmic steps in natural language, helping them begin implementation. It clarified which syntax and API

methods to use and how to use them in general. In response to incorrect or incomplete answers, it tailored sub-goals to guide them in fixing their solutions. By guiding them on one sub-goal at a time, it promoted iterative refinement.

An interesting finding is that the chatbot guided students to decompose problem statements by first identifying distinct, purposeful concepts before implementation. This mirrors the software engineering principle of high cohesion [27], where each sub-function handles a single, well-defined task. By isolating each sub-purpose, students created code with distinct sub-functions, promoting modularity, readability, reusability, and maintainability. By considering the purpose behind each component, the chatbot aligned their problem-solving strategies with professional practices. This underscores the importance of starting decomposition with purposeful concept identification for cohesive, effective solutions.

For students who struggled more with code writing, the chatbot guided them on how to write syntax by providing code examples and general guidance on using constructs and API methods. This suggests that it met students’ needs based on their prior knowledge, whether they were familiar with particular syntax or the high-level algorithms and logic pertaining to the concept. By operating within their zone of proximal development, it offered support that was appropriately challenging without overwhelming them.

While the chatbot was effective in most cases, some resisted using the suggested strategies, expecting direct answers due to prior experiences with ChatGPT. Others mechanically pasted suggestions without applying them. Future work should explore refining the chatbot interface to better promote productive engagement.

Some students believed they could only ask the chatbot complete, formulated questions, leading to uncertainty about what to ask. When they shared their thoughts with the chatbot per interviewer suggestion, they received better assistance. This suggests that chatbots and instructors should encourage students to share their problem-solving actions and general thoughts, including frustrations, so the chatbot can provide better assistance.

We observed that when the chatbot suggested breaking down lines of code to individual syntactical elements, it led students to ask more specific questions to address their misunderstandings. Similar to prior work on ITSs, this suggests we should design chatbots to elicit more information from learners through decomposition, allowing students to recognize which parts they do not understand and focus on those specific aspects. This may indicate that students are expressing metacognitive behaviors [25, 26], gaining awareness of what they do (not) know about the problem.

Lastly, some students independently applied the strategies previously suggested by the chatbot in later questions. These students exhibited metacognitive behaviors, such as learning to double-check their understanding using debuggers and execution on their own.

8 Limitations

Our data may not be representative of all students. The sample size is small, like most qualitative studies, due to the time-intensive nature of coordinating interviews. Participants were self-selected, potentially reflecting higher self-confidence and performance. The short duration of the study limits conclusions about long-term learning effects, which is an exciting area for future research.

References

- [1] John R Anderson, Albert T Corbett, Kenneth R Koedinger, and Ray Pelletier. 1995. Cognitive tutors: Lessons learned. *The journal of the learning sciences* 4, 2 (1995), 167–207.
- [2] Andrej Brodnik, Andrew Csizmadia, Gerald Futschek, Lidija Kralj, Violetta Lonati, Peter Micheuz, and Mattia Monga. 2021. Programming for All: Understanding the Nature of Programs. *arXiv preprint arXiv:2111.04887* (2021).
- [3] Paul Denny, Stephen MacNeil, Jaromir Savelka, Leo Porter, and Andrew Luxton-Reilly. 2024. Desirable Characteristics for AI Teaching Assistants in Programming Education. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*. 408–414.
- [4] Paul Denny, James Prather, Brett A. Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research (Koli, Finland) (Koli Calling '19)*. Association for Computing Machinery, New York, NY, USA, Article 11, 10 pages. <https://doi.org/10.1145/3364510.3366170>
- [5] K. Anders Ericsson and Herbert A. Simon. 1980. Verbal reports as data. *Psychological Review* 87 (1980), 215 – 251.
- [6] 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.
- [7] 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.
- [8] Mohammed Hassan and Craig Zilles. 2022. On Students’ Ability to Resolve their own Tracing Errors through Code Execution. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. 251–257.
- [9] Mohammed Hassan and Craig Zilles. 2023. On Students’ Usage of Tracing for Understanding Code. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 129–136.
- [10] Matthias Hauswirth and Andrea Adamoli. 2017. Metacognitive calibration when learning to program. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*. 50–59.
- [11] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, et al. 2019. Fostering program comprehension in novice programmers-learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. 27–52.
- [12] Kristopher J Kopp, M Anne Britt, Keith Millis, and Arthur C Graesser. 2012. Improving the efficiency of dialogue in tutoring. *Learning and Instruction* 22, 5 (2012), 320–330.
- [13] S Letovsky. 1986. Cognitive Processes in Program Comprehension: First Workshop. E. Soloway and S. Iyengar eds.
- [14] Mark Liffiton, Brad E Sheese, Jaromir Savelka, and Paul Denny. 2024. Code-Help: Using Large Language Models with Guardrails for Scalable Support in Programming Classes. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research (Koli, Finland) (Koli Calling '23)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. <https://doi.org/10.1145/3631802.3631830>
- [15] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin* 36, 4 (2004), 119–150.
- [16] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (Bologna, Italy) (ITICSE '06)*. ACM, New York, NY, USA, 118–122. <https://doi.org/10.1145/1140124.1140157>
- [17] David C Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. 1987. Mental models and software maintenance. *Journal of Systems and Software* 7, 4 (1987), 341–355.
- [18] Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2022. Metacognition and Self-Regulation in Programming Education: Theories and Exemplars of Use. *ACM Trans. Comput. Educ.* 22, 4, Article 39 (sep 2022), 31 pages. <https://doi.org/10.1145/3487050>
- [19] Tilman Michaeli and Ralf Romeike. 2019. Improving debugging skills in the classroom: The effects of teaching a systematic debugging process. In *Proceedings of the 14th workshop in primary and secondary computing education*. 1–7.
- [20] Mohammed Hassan, Grace Zeng, and Craig Zilles. 2024. Evaluating How Novices Utilize Debuggers and Code Execution to Understand Code. In *Proceedings of the 20th ACM Conference on International Computing Education Research (ICER'24)*, *Research Papers Track*. <https://doi.org/10.1145/3632620.3671126>
- [21] Mohammed Hassan, Kathryn Cunningham, and Craig Zilles. 2023. Evaluating Beacons, the Role of Variables, Tracing, and Abstract Tracing for Teaching Novices to Understand Program Intent. In *Proceedings of the 19th ACM Conference on International Computing Education Research (ICER'23)*, *Research Papers Track*. <https://doi.org/10.1145/3568813.3600140>
- [22] Matthias Nückles, Sandra Hübner, Sandra Dümer, and Alexander Renkl. 2010. Expertise reversal effects in writing-to-learn. *Instructional Science* 38, 3 (2010), 237–258.
- [23] Michael P O’Brien, Jim Buckley, and Teresa M Shaft. 2004. Expectation-based, inference-based, and bottom-up software comprehension. *Journal of Software Maintenance and Evolution: Research and Practice* 16, 6 (2004), 427–447.
- [24] Karl Popper. 2005. *The logic of scientific discovery*. Routledge.
- [25] James Prather, Brett A. Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. 2020. What Do We Think We Think We Are Doing? Metacognition and Self-Regulation in Programming. In *Proceedings of the 2020 ACM Conference on International Computing Education Research (Virtual Event, New Zealand) (ICER '20)*. Association for Computing Machinery, New York, NY, USA, 2–13. <https://doi.org/10.1145/3372782.3406263>
- [26] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 41–50.
- [27] G Priyalakshmi and R Latha. 2018. Evaluation of software reusability based on coupling and cohesion. *International Journal of Software Engineering and Knowledge Engineering* 28, 10 (2018), 1455–1485.
- [28] Siti Nurulain Mohd Rum and Maizatul Akmar Ismail. 2017. Metocognitive support accelerates computer assisted learning for novice programmers. *Journal of Educational Technology & Society* 20, 3 (2017), 170–181.
- [29] Vasile Rus, Sidney D’Mello, Xiangen Hu, and Arthur Graesser. 2013. Recent advances in conversational intelligent tutoring systems. *AI magazine* 34, 3 (2013), 42–54.
- [30] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 27–43.
- [31] Valerie J Shute, Chen Sun, and Jodi Asbell-Clarke. 2017. Demystifying computational thinking. *Educational research review* 22 (2017), 142–158.
- [32] Elliot Soloway, Beth Adelson, and Kate Ehrlich. 1988. Knowledge and processes in the comprehension of computer programs. *The nature of expertise* (1988), 129–152.
- [33] Kurt VanLehn. 2006. The behavior of tutoring systems. *International journal of artificial intelligence in education* 16, 3 (2006), 227–265.
- [34] Kurt VanLehn. 2011. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational psychologist* 46, 4 (2011), 197–221.
- [35] Anneliese Von Mayrhauser and A Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55.
- [36] Ye Wang. 2019. Study of Metacognitive Strategies’ Impacts on C Language Programming Instruction. In *2nd International Conference on Contemporary Education, Social Sciences and Ecological Studies (CESSES 2019)*. Atlantis Press, 112–116.