Hints Help Finding and Fixing Bugs Differently in Python and Text-based Program Representations

Ruchit Rawal[†§], Victor-Alexandru Pădurean[†], Sven Apel[¶], Adish Singla[†], Mariya Toneva[†]
[†]Max Planck Institute for Software Systems, Saarbrücken, Germany
[¶]Saarland University, Saarbrücken, Germany
[§]Corresponding Author
rawalruchit22@gmail.com, vpadurea@mpi-sws.org, apel@cs.uni-saarland.de, adishs@mpi-sws.org, mtoneva@mpi-sws.org

***Abstract*—With the recent advances in AI programming assistants such as GitHub Copilot, programming is not limited to classical programming languages anymore–programming tasks can also be expressed and solved by end-users in natural text. Despite the availability of this new programming modality, users still face difficulties with algorithmic understanding and program debugging. One promising approach to support end-users is to provide hints to help them find and fix bugs while forming and improving their programming capabilities. While it is plausible that hints can help, it is unclear which type of hint is helpful and how this depends on program representations (classic source code or a textual representation) and the user's capability of understanding the algorithmic task. To understand the role of hints in this space, we conduct a large-scale crowd-sourced study involving 753 participants investigating the effect of three types of hints (test cases, conceptual, and detailed), across two program representations (Python and text-based), and two groups of users (with clear understanding or confusion about the algorithmic task). We find that the program representation (Python vs. text) has a significant influence on the users' accuracy at finding and fixing bugs. Surprisingly, users are more accurate at finding and fixing bugs when they see the program in natural text. Hints are generally helpful in improving accuracy, but different hints help differently depending on the program representation and the user's understanding of the algorithmic task. These findings have implications for designing next-generation programming tools that provide personalized support to users, for example, by adapting the programming modality and providing hints with respect to the user's skill level and understanding.**

***Index Terms*—program comprehension, debugging, programming modalities, hints, crowd-sourced study**

I. Introduction

Recent advances in generative AI, in particular, foundation models trained on text and source code, have the potential to make programming more accessible. Most notably, tools such as GitHub Copilot [1] and ChatGPT [2] enable end-users to solve programming tasks through different modalities, including natural language text and pseudo-code specifications. Thus, programming is not limited to classical programming languages such as Python or C anymore, and programming problems can be expressed and solved by end-users in natural text. This shift is particularly significant in the evolving landscape of software engineering, where new and diverse user groups are no longer relegated to peripheral roles but are increasingly taking a central role in developing applications – many of which are now being created outside traditional IT departments by employees with limited or no technical development skills [3]. While these tools have enabled new forms of programming modalities, users still require algorithmic thinking and debugging skills to solve their programming problems [4]–[7]. Thus, there is a need to develop tools that can assist users with algorithmic understanding as well as finding and fixing bugs.

A series of recent works have explored how generative AI can be leveraged to support users with various forms of programming hints to help them find and fix bugs, while also forming and improving their programming capabilities [8], [9]. On the one hand, several works have proposed techniques that can provide tutor-style natural language hints [10]–[13]; on the other hand, they also investigated how to design informative test cases for program comprehension and debugging [14], [15]. However, the role and merits of hints have been studied only for a classical programming setting, and it is unclear whether and how the helpfulness of hints depends on the program representation. Given the fact that AI programming assistants broaden the population of users, it is further open whether and how hints should be adapted to a user's skill level and degree of understanding of the algorithmic task.

In this paper, we study the interplay of hint types and program representations regarding their usefulness for end-users. More concretely, we investigate the effect of three types of hints (test cases, conceptual, and detailed), across two program representations (Python and text-based), and two groups of users (with clear understanding or confusion about the algorithmic task). We center our study around the following three research questions:

RQ1: How does the program representation affect a user's ability to find and fix bugs, and how does this depend on the user's understanding of the task?

RQ2: What is the utility of a hint on the user's ability to find and fix bugs across different program representations, and how does this depend on the user's understanding of the task?

RQ3: What types of hint are more suitable for different program representations, and how does this depend on the user's understanding of the task?

To answer these research questions, we conducted a large-scale, crowd-sourced study involving 753 participants. In this

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00192

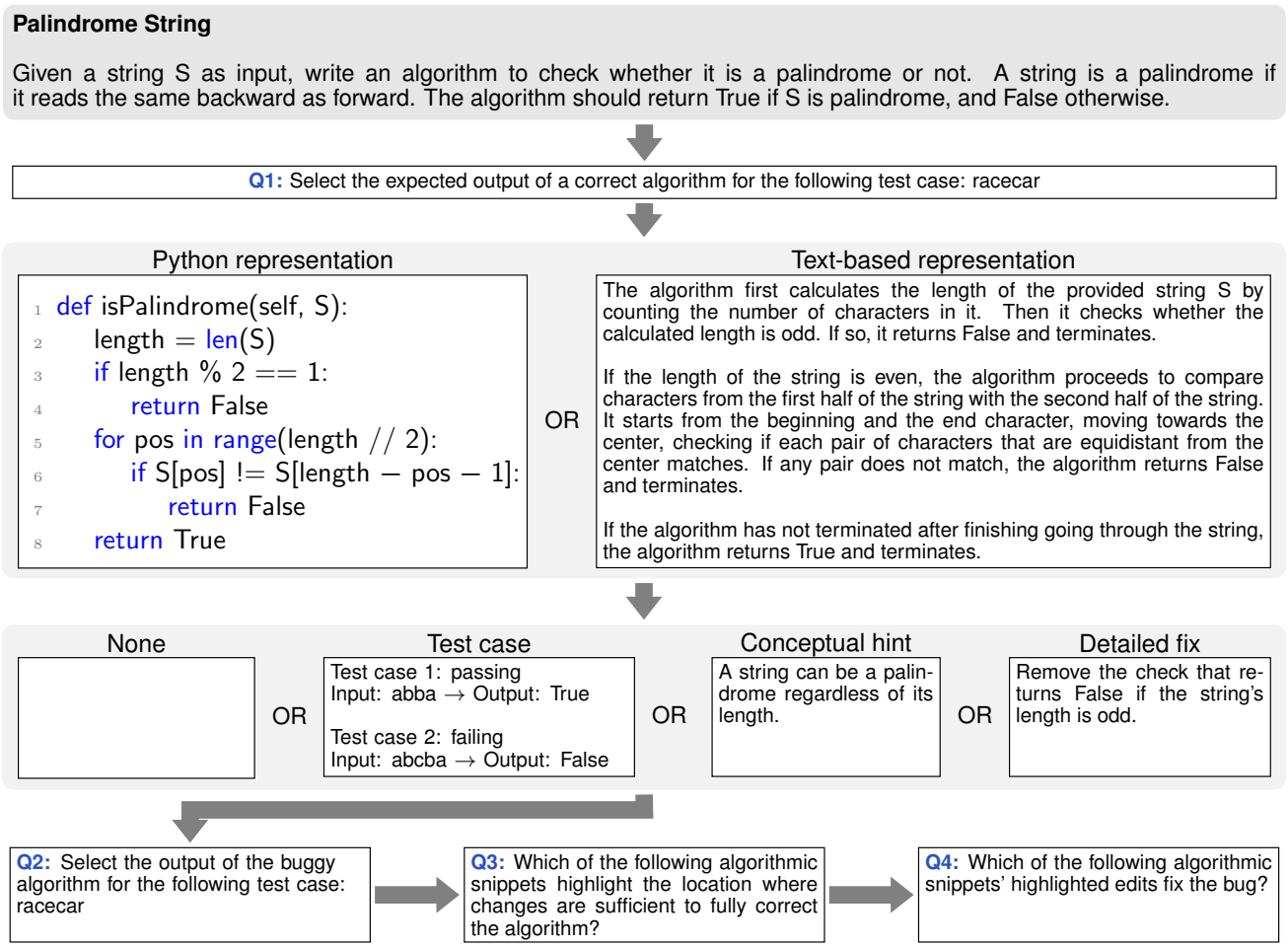

Fig. 1: An illustrative example from the study showcasing an algorithmic task. After showing a task, the user is asked to answer a question related to understanding of the task (Q1). Afterward, the user is shown a buggy program (in Python or text-based representation), possibly along with a hint. Then, the user is asked to answer questions related to bug understanding (Q2), bug finding (Q3), and bug fixing (Q4). These questions are posed as multiple-choice questions–the answer options are not shown in the figure for brevity.

study, a participating user is presented with an algorithmic task along with a buggy program (in Python or text-based representation) and is asked to find and fix bug(s) in the provided program, possibly with the help of a hint. We illustrate the experiment flow in Figure 1. We measure the utility of a hint primarily in terms of a user's ability to successfully find/fix bugs through a set of multiple-choice questions; we also consider an increase in speed for accurate responses as a secondary indicator of helpfulness. We summarize the takeaways for each Research Question in Figure 2. Our results for RQ1 show that the text-based program representation leads to better program debugging when considering users who demonstrate a clear understanding of the algorithmic task. Regarding RQ2, we found that hints for text representations do not appear to be helpful in improving user accuracy. On the other hand, hints for Python representations offer several advantages: (1) they improve user accuracy regardless of whether users have clear understanding or are confused; (2) they help reduce the accuracy gap between Python and text representations for users with a clear understanding; (3) they also reduce the accuracy gap between users who have clear understanding and who are confused about the algorithmic task. Finally, when investigating the interplay of program representation and hint types for RQ3, we found that detailed fixes are generally the most helpful across all representation modalities and user types, and conceptual hints are particularly beneficial for users with a confusion about the algorithm task when working with Python representations.

Our results have implications for designing next-generation programming tools that can provide personalized support to users that is adapted to their experience and understanding. For instance, our results indicate that users with a clear understanding of the algorithmic task can benefit from text-based program representation. Moreover, our results showcase how user accuracy can be improved by providing different types of hints and adapting them according to the programming

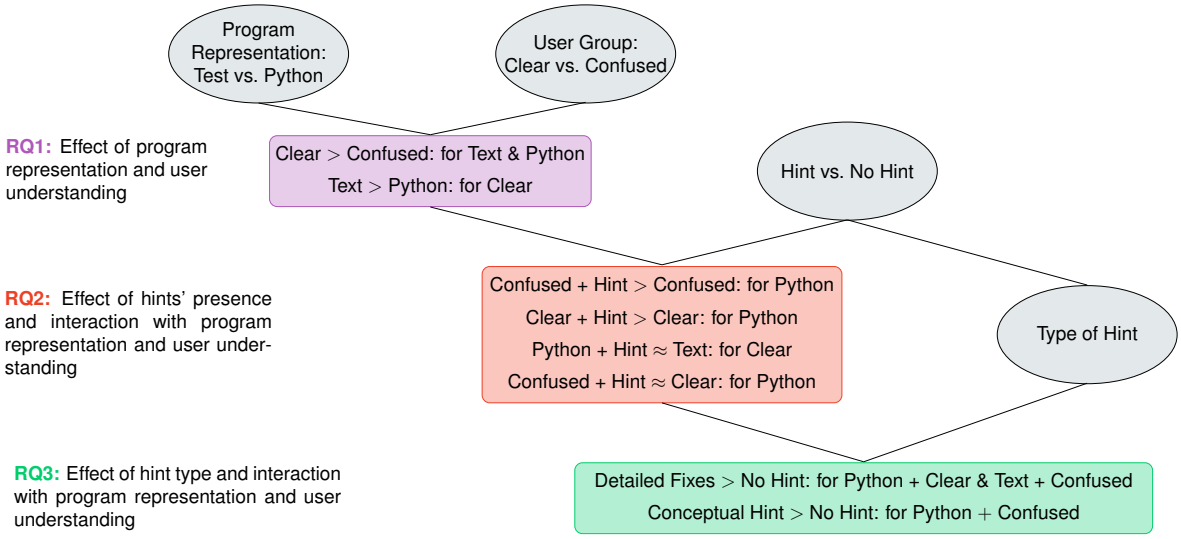

Fig. 2: Visual summary of main findings to our Research Questions (RQs). Circular nodes represent main factors of variation (Program Representation, User Group, Hint Presence, & Hint Type). Rectangular blocks show key takeaways, color-coded by research question (RQ1: purple, RQ2: red, RQ3: green). Connecting lines show how factors combine to address different RQs.

modality and the user's understanding of the task.

Our main contributions are as follows:

- We analyze the helpfulness of various hint types for two different program representations.
- We conduct a large-scale study to investigate how different hint types and modalities affect a user's ability to find and fix bugs in a program.
- We examine how hint types and program modalities can be adapted to a user's understanding.
- We formulate a set of explicit hypotheses that are meant to inform further work in this field.
- We provide all raw data and analysis scripts: https://github.com/bridge-ai-neuro/HintsCodeText

II. Related Work

A. Text vs. code for program comprehension

Several studies have examined the cognitive processes that support a programmer's ability to understand programs written in code or natural language, finding shared [16], [17] and distinct [18], [19] cognitive mechanisms, and differences in reading strategies [20]. Most previous studies have focused on passive comprehension paradigms where the participants were asked to simply read the program. A notable exception is Karas et al. [21], who studied the functional connectivity in the brain during the writing of program code and prose. However, this study used different problems in the code and prose conditions, which makes it difficult to directly contrast the results. Our work complements these previous findings and focuses on the effect of program representation on finding and fixing bugs.

B. Difference in program understanding for different users

Most research on program comprehension does not consider inter-personal differences of programmers nor differences in their skills, levels of understanding the problem at hand, and backgrounds. In fact, most studies work with students as study subjects. Notably, a line of research concentrates specifically on the effect of programming experience on program comprehension. For example, Uesbeck et al. [22] suggest that lambda expressions may hinder program comprehension, but only for novices, not experienced programmers. Stefik and Siebert [23] found that syntactic constructs that are difficult for novices may guide teachers in choosing the appropriate language to start with. Burkhardt et al. [24] studied the effect of expertise on program comprehension. They found that the models experts build from a program and task differ from the models novices build, indicating different cognitive processes or mental modelling strategies involved. Vessey [25] studied the difference between experts and novices in debugging tasks. She found that experts adopt a holistic system view (and use breadth-first search), whereas novices do not (using depth-first search). In the same vein, Détienne [26] notes that experts incorporate object-oriented and functional relationships in their reasoning, whereas novices focus on objects only. In a family of experiments, Dieste et al. [27] found that years of experience are a suboptimal predictor of programmer performance, academic background and specialized knowledge of task-related aspects are better predictors. In contrast to these previous works, we define groups of users based on their understanding of the specific algorithmic task, which we measure empirically in the beginning of the same experiment.

C. Supporting users with various forms of programming hints

A variety of assistive techniques have been considered in the literature that support users with programming hints with the goal of helping them find and fix bugs. Prior to recent developments in generative AI, automated techniques primarily focused on hints presented in the form of bug fixes because of challenges in automatically generating high-quality natural

language hints [28]–[31]. Another line of research investigated crowd-sourcing approaches to obtain hints provided by other learners or tutors [31]–[33]. Recent developments in generative AI have led to a surge in automated techniques that provide tutor-style natural language hints, for example, by providing conceptual hints without revealing details about fixes, thereby considering aspects of forming user's programming capabilities [10], [13]. Moreover, automated generative techniques have been proposed that provide effective error messages for syntactical errors [11], [34] or design informative test cases for a given buggy code [14], [15]. However, these works have considered hints only for classical programming settings, and it is unclear how the helpfulness of hints depends on programming modality, representations, and a user's skill level and capability to understand the problem at hand. Our work complements these works and focuses on understanding the interplay of hint types and program representations regarding their usefulness for end-users.

III. Methodology

A. Experiment Design

To investigate how different program representations and hint conditions affect participants' ability to understand, identify, and fix bugs, we conducted a large-scale crowd-sourced study with a total of 753 participants. We visualize the design flow of the study in Figure 1. The study featured two primary types of program-representation conditions: text-based description and Python code. For each program representation type, there are four possible hint-conditions–no hint, test cases hint, conceptual hint, or detailed fix hint–resulting in a total of eight distinct conditions. Each participant was randomly assigned to one of these eight conditions. Within their assigned condition, participants were presented with two algorithmic tasks, each focusing on one of five different problems (see Section III-C for details about the problems).

Participants took an average of 15.76 minutes to complete the survey. We recorded participants' responses and response times for each question, as well as the time they spent on each step of the survey, including reading and understanding content on pages like the program page and the hint page. Before beginning the study, we asked participants to fill out a short demographics survey that included questions about their experience with programming, self-rated Python programming skills (on a scale of 1 to 10), and familiarity with English reading comprehension skills (on a scale of 1 to 10). Our participant pool predominantly consisted of individuals from English-speaking countries such as the USA, who rated themselves very highly on English reading comprehension. Regarding the programming questions, the participant pool was skewed towards individuals with less self-reported programming experience. We found the self-reported measures of programming experience and skills unreliable, as there was no strong relationship between these measures and the participants' understanding of the algorithmic task (i.e. accuracy on Q1); therefore, we decided not to use these self-reported measures for the main analyses, and instead group participants by the data-derived measure of Q1 accuracy, which indicates whether the participant correctly understood the problem description (see Sec. III-G for more information about how this grouping was used in our analyses).

B. Utility Metrics

We focus on two metrics to quantify the utility of program representation and the provided hints for program understanding and debugging:

1) **Accuracy:** We assess participants' performance by computing the average accuracy over multiple-choice questions Q2, Q3, and Q4, as they cover different aspects of successfully debugging a program. The range of average accuracy ranges from 0 to 1, with theoretical chance accuracy of 0.305 (Q2: 4 options, Q3: 3 options, Q4: 3 options).
2) **Time Taken:** We measure the average response time of participants when answering questions Q2, Q3, and Q4 correctly. This includes the time taken to read the question, review the answer choices, and submit the response. This metric allows us to determine whether accuracy gains come at the cost of efficiency (inversely proportional to time taken) or in addition to it.

By evaluating both accuracy and time taken, we gain a holistic understanding of how program representations and hints affect program debugging [35], [36].

C. Stimulus Design

Since we target end-users, we use 5 basic computer science algorithmic tasks ranging in difficulty, commonly used in CS1 education, programming websites, and literature [12], [37]–[40]. The problem titles with brief descriptions are as follows:

- "Sum Positive Values" – calculate the sum of the positive values in the input list A.
- "Count NonNegatives and Negatives" – check whether there are more non-negative values than negative values in an input list A.
- "Print Average Rainfall" – print the average of non-negative integers representing daily rainfall amounts in the input list A.
- "Palindrome String" – check whether the input string S is a palindrome.
- "Fibonacci to N" – print the list of numbers in the Fibonacci sequence till the input number N.

We have 5 instances of buggy programs for each algorithmic task to ensure that our approach generalizes beyond specific implementations of the task, leading to a total of 25 program instances. Figure 1 shows an example of one instance of a buggy algorithm for the "Palindrome String" problem. We present both the Python and text-based representations for this instance. The bug is that the algorithm mistakenly treats all odd-length strings as non-palindromes. The figure also shows the three types of hints for this instance. We provide examples of the other algorithmic tasks on our GitHub repository. We provide details on how we constructed the Python and text representations of the programs, and the hints below:

1) Python Representation-Condition: To obtain the buggy programs in Python representation for "Palindrome String" and "Fibonacci to N" problems, we took buggy Python attempts from recent literature used to benchmark generative AI models [12], [13] – these are adapted from attempts publicly available on the platform geeksforgeeks.com [37]. Next, to obtain the buggy programs for "Sum Positive Values", "Count NonNegatives and Negatives", and "Print Average Rainfall", we manually plant bugs starting from the correct Python solution, based on the bugs we have encountered in our experience of working with students. As it is unclear how syntactic bugs can be reflected in text-based representations, we opt to include solely buggy codes with semantic bugs. Specifically, we first identified key sub-objectives necessary for correctly solving specific algorithmic tasks. For example, in the "Print Average Rainfall" task, previous research [38] has highlighted several essential objectives: handling negative inputs (Negative), summing the inputs (Sum), determining the number of inputs (Count), addressing cases with zero inputs (DivZero), and calculating the average (Average). We then designed bugs that cause the program to fail in one or two of these sub-objectives. This approach ensures that the bugs target specific functional aspects of the task, leading to meaningful failures. Examples of other such bugs include wrongly initializing counter or accumulator variables, starting to iterate from index 1 instead of index 0, using wrong comparison logic in conditionals, and so on. The buggy codes we include can be fixed with a few localized changes.

2) Text Representation-Condition: For each Python program, we carefully craft a corresponding text-based representation that describes the Python program in natural language without using any programming concepts such as "variables", "loops", etc. These descriptions underwent several rounds of internal iterations to ensure accuracy, clarity, and faithfulness to the original Python code.

3) Hint-Conditions: As previously discussed, each participant is assigned one of four hint conditions: no hint, test cases hint, conceptual hint, and detailed fix hint. The hint conditions are carefully chosen to target different aspects of overall bug understanding, being grounded in providing support to students in programming education [12], [14], [15]. The test case hint includes two input-output pairs, one representing a success case and the other a failure case, with minimal differences in their input. Conceptual hints highlight the underlying issue present in the program without suggesting specific changes, while detailed fixes focus solely on the necessary changes without explaining the underlying issue. Importantly, these hints are independent of the program representation and are applicable to both text-based and Python representations for a given problem type. The authors dedicated multiple iterations to handcrafting these hints, ensuring their clarity and applicability across both representations.

D. Multiple-Choice Questions:

Our study consisted of four multiple-choice questions per problem type to assess participants' understanding and evaluate the utility of hints. More specifically,

Q1. **[Understanding of problem description]** *Q. Select the expected output of a correct algorithm for the following test case: . . . ?* This question had four options, with one correct answer and three distractors, resulting in chance accuracy of 0.25.

Q2. **[Understanding output of buggy program]** *Q. Select the output of the buggy algorithm for the following test case: . . . ?* This question also had four options, with one correct answer and three distractors, resulting in chance accuracy of 0.25.

Q3. **[Ability to localize bug]** *Q. Which of the following algorithmic snippets highlight the location where changes are sufficient to fully correct the algorithm?* This question consisted of three options, each highlighting different portions of the program, with one correct answer and two distractors, resulting in chance accuracy of 0.33.

Q4. **[Ability to fix bug]** *Q. Which of the following algorithmic snippets' highlighted edits fix the bug?* This question also had three options, each featuring highlighted modifications in the program, with one correct answer and two distractors, resulting in chance accuracy of 0.33.

All questions remained consistent across different hint conditions. Q1 and Q2 were consistent across different program representation conditions, as they focused on the input/output behavior of an ideal and buggy program, respectively, which is consistent across both program representation conditions for a particular algorithmic task. Q3 and Q4 involved identifying and fixing bugs by selecting the option with the correct portion highlighted. These questions varied across different program representations; in the text-based condition, highlights were on the text stimuli, and in the Python condition, highlights were on Python stimuli. However, we ensured that the underlying content remained the same for both questions; the locations and fixes in the Python version corresponded to text descriptions in the text-based version. We provide the quantitative questions and answer choices for the algorithmic task example shown in Figure 1 in our GitHub repository.

E. Participant Recruitment:

Our experimental study was designed and implemented using Qualtrics. We recruited participants via Cloud Research, a crowdsourcing platform that builds on top of Amazon Mechanical Turk, but applies a series of stringent filters and quality control measures to significantly enhance the reliability of the participant pool. Our study received formal approval from the Ethics Review Board of our institute, ensuring that all ethical considerations were met. Before beginning the tasks, each participant provided informed consent. The study was designed for completion within 30 minutes, with participants compensated at 12 USD per hour.

F. Participant Demographics

Our study was conducted over two weeks and included a total of 753 valid respondents. 417 participants were male, 322 were female, 10 identified as non-binary, and the remainder

preferred not to disclose their gender. The median age of participants was 39, with ages ranging from 20 to 78. The average programming experience in the past five years was 1.13 years, with a minimum of 0 years and a maximum of 5 years. The average self-rated Python programming skill level was 2.36, on a scale from 1 to 10.

G. Data Analysis:

As described in Section III-A, we asked each participant to complete two independent algorithmic tasks, each involving a different problem but the same type of program representation and hint. Additionally, we applied a 2-second threshold per question to flag and exclude responses indicative of guessing, based on a conservative estimate from an internal pilot survey. Of the 759 responses initially collected, 6 participants were excluded. The remaining responses had an average response time of 124 seconds and a median of 89.6 seconds.

For our data analysis, participants were first grouped according to their understanding of the problem description, as evidenced by their accuracy on Q1: participants who answered Q1 correctly were labeled as the "clear" understanding group, while those who answered incorrectly were labeled as the "confused" understanding group. To ensure our dataset consisted of independent observations and minimized within-person variance, we averaged the scores of the quantitative questions (Q2, Q3, Q4) across both task responses for participants consistently classified as either "clear" or "confused" in both tasks. If a participant fell into different groups across tasks, we randomly selected one of the two responses and discard the other response from all the analyses. Similarly, for average time results, as discussed in Section III.B, we only focus on participants from the "clear" group and only include responses where Q2, Q3, and Q4 were all correctly answered. For each participant, if Q2, Q3, and Q4 were correctly answered in both tasks, we averaged the response times across the two tasks. If these questions were correctly answered in only one task, we used the time from that task. Importantly, all other conditions – such as hint type and program representation – were consistent across the two tasks based on our study design. This process ensured that our dataset includes only one data point per participant, aligning with the assumptions required for the statistical tests we will discuss shortly. Participants were also grouped based on the program representations they were assigned (Python vs. text). When analyzing the effect of hints, we aggregated over the three different hint types to compare the presence of hints to the absence of hints in RQ2. In RQ3, we analyzed the different hints separately in comparison to the no-hint condition. For each RQ, we present bar plots of the mean utility metrics, with error bars representing the standard error of the mean.

H. Significance Testing

We use the Wilcoxon rank-sum test [41] to determine significant differences between different conditions and indicate significant differences by asterisks in all result figures ($*$ for p-value < 0.05, $**$ for p-value < 0.01). The Wilcoxon rank-sum test relies on two key assumptions: the samples must come from populations with the same shape, and the observations must be independent. To ensure these assumptions are met, we test whether the samples share the same shape using the Kolmogorov-Smirnov test [42] before drawing any conclusions about statistical significance. Additionally, as we discussed earlier, we use only one data point per participant, which maintains the independence of observations. We use the Wilcoxon signed-rank test [41] to determine whether a particular condition is significantly different from the corresponding chance accuracy. We chose these statistical tests because they do not make assumptions about the underlying data distribution (e.g. that data is distributed according to a Normal distribution). Additionally, we use Cohen's d [43] to measure effect sizes for significant differences between conditions. We report Cohen's d and the difference of means in Section IV, along with any mention of significant differences between the two conditions.

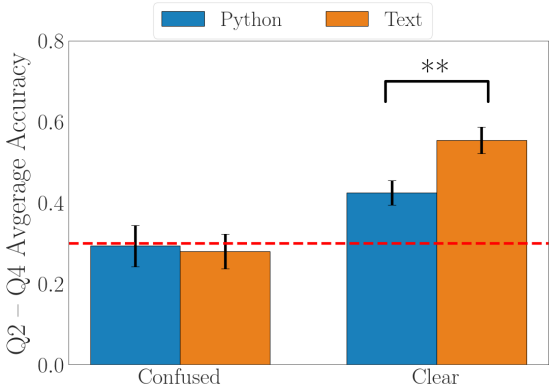

Fig. 3: Accuracy of participants when presented with text-based vs. Python-based program representations and no hints. The bar plot represents the mean Q2–Q4 average accuracy, with the vertical lines indicating the standard error of the mean. The red dotted line represents chance accuracy, and significant differences between program representations are indicated with an asterisk ($*$). Surprisingly, clear participants perform significantly better when presented with text-based representations than Python-based representations.

IV. Results

RQ1. *How does program representation affect a user's ability to find and fix bugs, and how does this depend on the user's understanding of the task?*

To investigate this question, we first compare the accuracy of participants who view the program in natural text with those who view it in Python. For this analysis, we only consider responses from participants in the "No Hint" condition to avoid any potential interaction effects between the hint type and program representation. We further divide the analysis according to two groups of participants: those with a confused

understanding and those with a clear understanding. The results are reported in Figure 3.

Unsurprisingly, participants with a confused understanding of the problem description performed at chance levels (i.e., 0.305) for both text-based and Python representation conditions. In contrast, participants with a clear understanding of the problem description performed significantly above chance in both the text-based (p-value: 3.01×10^{-9}, effect-size: 0.959, difference of means: 0.248) and Python representation conditions (p-value: 3.75×10^{-5}, effect-size: 0.505, difference of means: 0.119). Additionally, these participants performed significantly better when viewing the program in text-based representations compared to Python-based representations (p-value: 0.004, effect-size: 0.520, difference of means: 0.129).

We further examine how program representation affects the participants' response time. To investigate this, we analyze the time taken by participants to correctly answer Q2 through Q4 in Figure 6-(a). For this analysis, we only considered the time taken by participants who answered all three questions (Q2, Q3, and Q4) correctly. Moreover, we focused on the clear group participants, as only a few participants with a confused understanding remained after filtering out the incorrect responses. In Figure 6-(a), we compared the average time taken by participants using text-based representations with Python representations and find a text-based representations to take a significantly longer time (p-value: 0.047, effect-size: 0.439, difference-of-means: 0.547). However, it is important to note that the average response time for questions is influenced by the time required to read program representations, as Q3 and Q4 multiple-choice options contain the original or modified program representations. To isolate this factor, we compared the average time for only correctly answering Q2, which has identical questions and multiple-choice options for both Python and text-based representations. We found no significant differences in response times for Q2 (p-value > 0.05).

RQ1 Takeaways: For participants with clear understanding, text-based representations led to significantly better accuracy than Python representations. Confused participants performed at chance levels for both representations. While text-based representations required more time overall, this was likely due to longer reading times rather than reduced efficiency in problem-solving.

RQ2: *What is the utility of a hint on the user's ability to find and fix bugs across different program representations, and how does this depend on the user's understanding?*

We investigate this question by comparing the accuracy of participants in the "No Hint" condition with the average accuracy across the other three hint conditions, denoted via "+ Hint" in Figure 4. Given the significant effects of representation condition and participant understanding group observed in Figure 3, we analyzed the utility of hints separately for text-based and Python representation conditions and for both clear and confused understanding groups of participants.

In Figure 4, we present the accuracies when presented with

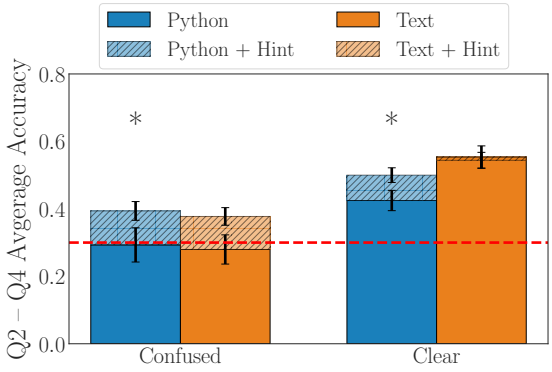

Fig. 4: Accuracy of participants when presented with hints and no hints, across different program representations. The bars represents the mean Q2–Q4 average accuracy, with the vertical lines indicating the standard error of the mean. The red dotted line represents chance accuracy, and significant differences between the no hint and with hint conditions are indicated with an asterisk ($*$). Hints significantly improved accuracy for confused and clear participants for Python program representations. Hints also bridged accuracy gaps between representations (for clear participants) and understanding levels (for Python representation).

any hint compared to the no hint condition, for each participant group and representation-condition pair. Significantly better accuracy with hints over no hint is indicated with an asterisk ($*$). For the Python representation, hints significantly helped both confused (p-values 0.043 effect-size: 0.36, difference of means: 0.10) and clear participants (p-value: 0.049, effect-size: 0.27, difference of means: 0.07). In contrast, for text-based representation, while both clear and confused participants don't seem to be significantly helped by the provided hints, the confused participant groups show a small to medium effect size of 0.37 (p-value > 0.05, difference of means: 0.09).

Confused participants, who initially performed at chance, improved significantly with hints in the Python condition. Additionally, the accuracy difference between the Python and text-based conditions for clear participants without hints disappeared when hints were provided for the Python condition (p-value > 0.05, effect-size: 0.19, difference of means: 0.05). This suggests that hints help close the accuracy gap between different programming representations. Furthermore, for the Python condition, hints boosted the accuracy of confused participants to match that of clear participants without hints (p-value > 0.05, effect-size: 0.116, difference of means: 0.031), showing that hints can reduce the gap between participants with varying levels of understanding.

In Figure 6-(a), we analyze how hints affect the participants' response time. We note that hints improve efficiency (reduce response time) for text-based program representations (p-value: 0.004, effect-size: 0.401, difference of means: 0.405), while for Python program representations, there is no significant difference. This result is complementary to the findings in Figure 4, where hints improved the accuracy of

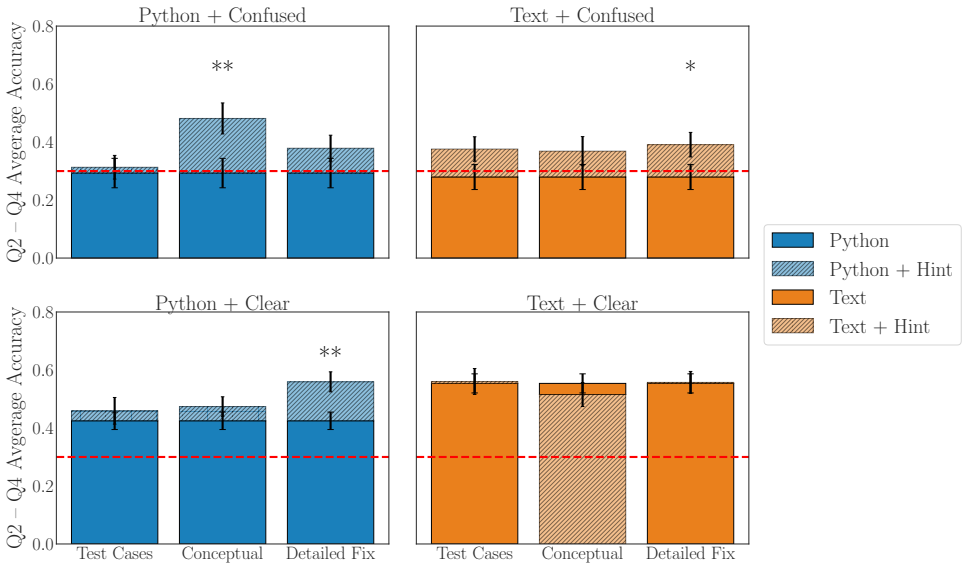

Fig. 5: Accuracy of participants when presented with different hints or no hint, across different program representations and participants' level of understanding separately. The bars represents the mean Q2–Q4 average accuracy, with the vertical lines indicating the standard error of the mean. The red dotted line represents chance accuracy, and significant differences between the no hint and different hint type conditions are indicated with an asterisk (∗). Detailed fixes are generally most helpful, while conceptual hints are particularly useful for participants with confused understanding in the Python representation condition.

clear group participants only for Python representations, with no significant effect on text-based representations.

RQ2 Takeaways: Hints significantly improved accuracy for Python representations across both clear and confused understanding groups. For text-based representations, hints reduced response time for those with clear understanding. Additionally, hints bridged accuracy gaps between different representations and understanding levels.

RQ3: *What types of hint are more suitable for different program representations, and how does this depend on the user's understanding?*

As previously mentioned, we aggregated results over three different hint types for investigating **RQ2**. In this section, we explore whether different hint types are more beneficial for different program representations and whether this varies based on the participants' understanding. To investigate this, we extend our previous findings by analyzing the accuracy for each hint type separately and comparing whether they help improve accuracy compared to the no hint condition.

In Figure 5, we present results for each hint type. Among the three types of hints, detailed fixes are generally the most helpful. They significantly improve accuracy for clear participants with Python representations (p-value: 0.006, effect-size: 0.527, difference of means: 0.134), and confused participants with text-based representations (p-value: 0.047, effect-size: 0.484, difference of means: 0.111). Additionally, conceptual hints are especially beneficial for confused participants working with Python representations (p-value: 0.009, effect-size: 0.626, difference of means: 0.188). We did not observe any significant effect of test case-based hints on accuracy compared to no hints. Thus, the effectiveness of hints varies depending on the program representation and the user's understanding level.

In Figure 6-(b) & -(c), we plot the time taken by clear group participants to correctly answer Q2, Q3, and Q4. Consistent with our results while investigating RQ2, we note that response time reductions are only observed for the text-based program representations, where test cases reduce the response time significantly (p-value: 0.002, effect-size: 0.512, difference of means: 0.541). We do not see significant effects for other hint types, and observe small effect-size for conceptual hint and (effect-size: 0.104), and small-to-medium effect size for detailed fix (effect-size: 0.408). Notably, test cases did not improve the accuracy for any of the program-representation and participant-understanding conditions (refer to Figure 5).

RQ3 Takeaways: Different hint types vary in utility depending on program representation and participant understanding. Detailed fixes generally improved accuracy most, while test cases reduced response time for participants with clear understanding viewing text-based representations.

V. Threats to Validity

A. Internal Validity

Internal validity refers to the ability to accurately establish cause-and-effect relationships between independent and dependent variables. In our study, internal validity is threatened by the use of crowdsourcing for data collection, which can introduce participant-dependent confounders. To mitigate this threat, we implemented a comprehensive randomization procedure that ensures that participants are randomly assigned

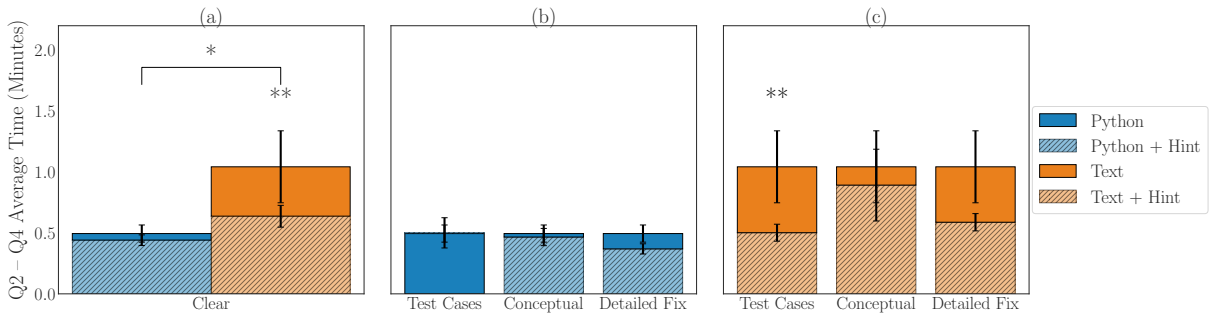

Fig. 6: Time taken by participants to correctly answer Q2, Q3, and Q4 when presented with different hint types or no hint, across different program representations for clear understanding group participants. The bar plot represents the mean of the average time required to answer Q2–Q4 correctly, with the vertical lines indicating the standard error of the mean. In (a), "+ Hint" represents the average of the three different hint conditions, while in (b) and (c), we plot the different hint types separately. Significant differences between the no hint and hint conditions are represented by ($*$). Even though the hints may not be helpful in improving accuracy for the Text + Clear condition, they help reduce participants' response time. Similarly, improvement in accuracy doesn't necessarily imply reduction in response time (Python + Clear condition). For the Text + Clear condition, test cases significantly reduce response time.

to one of our eight distinct conditions. With more than 750 participants, we expect the influence of individual differences to be evenly spread across all conditions, effectively balancing out. Furthermore, we carefully considered the various independent variables during data analysis. For instance, when determining the effect of program representation (refer to **RQ1** in Section III-G), we analyzed only the "No Hint" condition to avoid confounding effects from the presence of hints. Likewise, we analyzed different hint conditions separately for each program representation. These methodological considerations strengthen the internal validity of our findings.

B. Construct validity

A key consideration in our study was ensuring construct validity, that is, reliably measuring the utility of different program representations and hints in bug understanding. As common in the literature, we measure utility through accuracy and response time metrics. We expect that a deeper understanding of a program and bug will lead to improved accuracy and reduced response time. To assess accuracy, we crafted three questions for each algorithmic task, probing participants' comprehension of the bug's impact (Q2), location (Q3), and potential fix (Q4). Please refer to Section III-D for a detailed discussion regarding the different questions. The mean accuracy across these questions served as our accuracy metric. For response time, we measured the average time taken to answer these questions by participants who answered accurately. The construct validity for average time can be threatened by program representation. Specifically, text representations generally require longer reading times compared to Python code, and two of our questions (Q3 and Q4) included program representations in their answer choices. This could lead to artificially inflated response times for participants in the text condition, potentially masking true differences in response time. Therefore, while reporting the response time comparisons between Python and text representation conditions, we also report the response time based solely on Q2, which is exactly the same across both representations.

C. Statistical conclusion validity

To ensure statistical conclusion validity – the degree to which conclusions we reach from analyses are accurate and appropriate – we implemented several key measures in our study design and analysis. First, we secured a large sample size of 753 participants, significantly enhancing the power of our statistical tests and allowing us to detect even subtle effects with sufficient confidence. In our analysis, we employed appropriate statistical tests that do not make assumptions about the underlying data distribution. Specifically, we examined the statistical difference between conditions using the Wilcoxon rank-sum test and the statistical difference between a condition and the theoretical chance accuracy using the Wilcoxon signed-rank test. The Wilcoxon rank-sum test relies on two key assumptions: the samples must come from populations with the same shape, and the observations must be independent. To ensure these assumptions are met, we verify that the samples share the same shape using the Kolmogorov-Smirnov test [42] before drawing any conclusions about statistical significance. Additionally, since we use only one data point per participant, the independence of observations is maintained. We indicate significant differences in all result figures using asterisks ($*$ for p-value < 0.05, $**$ for p-value < 0.01). Furthermore, we implemented rigorous randomization procedures to balance our experimental groups. This approach to group assignment minimized the potential for confounding variables and strengthened our ability to attribute observed effects to our manipulated conditions.

D. Ecological validity

Ecological validity measures how well an experimental setup reflects real-world conditions. Our study environment diverged from typical programming scenarios in some respects,

potentially impairing ecological validity, as participants lacked access to an Integrated Development Environment (IDE) or a debugger–common tools used in programming. This design choice, while potentially reducing ecological validity, served to increase internal validity by standardizing the environment across all participants. Moreover, it allowed us to isolate the accuracy and response time improvement effects to the program representation and/or the provided hints. Additionally, we selected tasks and code snippets particularly relevant to novice programmers and educational settings. This approach, though not perfectly replicating professional development scenarios, aligned with the experiences of our target population in learning contexts. Thus, we strived to maintain ecological validity while adhering to rigorous experimental controls.

E. External validity

External validity concerns the ability of our results to generalize to other settings, participants, and measures. We posit that our findings are likely generalizable to similar contexts, i.e., small to medium-sized code snippets and algorithmic tasks of comparable complexity. However, we acknowledge that the effectiveness of these representations and hints may not necessarily extend to large-scale software projects. Additionally, our participant pool, primarily recruited from crowdsourcing platforms, consisted largely of individuals with limited programming experience. This may limit the generalizability of our results to other populations such as computer science students or highly experienced professional programmers. Nevertheless, it is important to consider the evolving landscape of software engineering, where diverse user groups – often untrained in programming – are increasingly playing central roles in application development [44]. Our insights align well with this emerging trend. Finally, it is important to note that our study represents one of the first comprehensive investigations into the utility of program representations, hints, and their interactions in the context of bug understanding. As such, our findings offer valuable guidance for developing tools that adapt to programming representations and user expertise. They also underscore the need for future research to explore the generalizability of these findings across a broader spectrum.

VI. Discussion

In our study, we found that text representations are more beneficial in terms of accuracy than Python representations for individuals with a clear understanding of the algorithmic task. One hypothesis for this possibly counter-intuitive finding is that Python code is structured and contains implicit beacons (e.g., variable and function names) that aid program comprehension without the need of going through every statement – a process called *top-down comprehension* [45], [46]. The text representation forces even seasoned programmers to go through all details from beginning to end, called *bottom-up comprehension* [47], [48], which makes it more likely to spot otherwise hidden bugs.

H1: Python representations trigger top-down comprehension, whereas text representations trigger bottom-up comprehension. The difference between the two will be more pronounced the more experienced programmers are.

An alternate hypothesis is that, even though our sample size was large (over 700 participants), the majority of the participants had little self-reported experience with coding or Python and this may also have contributed to their improved accuracy with text over Python representations.

H2: Text representations aid inexperienced programmers since they strip the algorithmic description from syntactical and technical information arising from the use of a formal programming language.

Future work that repeats our experiment in a population of experienced participants with Python can disambiguate between these two hypotheses, as they make very different predictions for experienced and inexperienced participants: the first hypothesis predicts that the more experience with code the participant has, the stronger their prior will be, and the easier it would be to overlook the bug and therefore the gap between text and Python representations will widen. The second hypothesis predicts that the more experience with code the participant has, the more they will benefit from the Python representation in terms of accuracy of finding bugs. We conducted a preliminary investigation by analyzing participants divided into three groups based on self-reported programming proficiency: low, medium, and high experience. Our findings revealed that participants across all experience levels performed better with text-based representations than with Python. This effect was significant for those with low (p-value: 0.021, effect-size: 0.452, difference of means: 0.08) programming experience. We share the detailed results in our GitHub repository due to space constraints. These initial results suggest evidence towards H2, i.e., text representations aid inexpierenced programmers. However, to draw more definitive conclusions, further experiments under controlled conditions using objective and verifiable measures of programming expertise beyond self-reporting are necessary.

We further found that the addition of hints was particularly useful for the Python representation. In these cases, hints improved user accuracy regardless of whether the participant had a clear or confused understanding. Furthermore, participants with a confused understanding benefited so much from the hint that their accuracy was statistically indistinguishable from that of the participants with originally clear understanding. This suggests that the lack of algorithmic understanding can be made up using hints. Additionally, hints also bridge the gap between the Python and text representations for participants with clear understanding, such that participants who see the program in Python and receive a hint perform at the level of those who see the program in text. This result suggests that hints can be a powerful tool in aiding program debugging in code, when faithful text descriptions are difficult to generate.

Interestingly, we find that hints have contrasting effects on accuracy and completion time for Python vs. natural text representations: hints improve accuracy but not speed for Python representations, and improve speed but not accuracy

for text representations. Again, this may be because the Python representation already provides a sufficiently high-level description of the program that can be accessed quickly so the speed is difficult to improve upon. As said in H1, this representation can also make it harder to detect the bugs, which is where hints can help by drawing attention to specific issues in the program. In contrast, text representations require going through the program word-by-word to extract the overall structure and may thus require longer time to integrate the higher-level understanding of the program. While this low-level presentation can be helpful in detecting bugs, manipulating it can be slow, which can be improved by the addition of higher-level hints.

H3: Hints alter strategies for program comprehension and bug finding.

Lastly, we found that among the 3 different types of hints–test cases, conceptual, and detailed fix–detailed fixes are generally the most helpful in improving accuracy across all representation modalities and user types. Additionally, conceptual hints are particularly beneficial for users with a confused understanding when working with Python representations. While we don't find a significant improvement in response accuracy due to test cases, we observe that they lead to the biggest improvement in efficiency (i.e. reduced time per accurate response), specifically for text-based representations. Similarly to the benefit of the natural language conceptual hint in the Python representation, here we also observe a benefit when mixing the presentation modalities of the program and the hint. We hypothesize that the mixing of presentation modalities may contribute to a more holistic understanding of the program.

H4: Mixing natural text and code representations improves holistic understanding of the program.

Overall, these results suggest a debugging workflow that is personalized to the level of algorithmic understanding of the user: if the user understood the posed algorithmic task, then they benefit most from seeing the program described in natural language. If the user did not understand the algorithmic task from the start, then they most benefit from seeing the program in Python and receiving a conceptual hint. While we focus here on short-term improvements in program understanding, we hope that our results can serve as a starting point for future work that investigates the utility of hints in long-term program understanding over multiple educational sessions.

VII. Conclusion and Perspectives

Recent advancements in generative AI have revolutionized programming accessibility, allowing users to solve tasks through various representations, including natural language and pseudo-code. However, these new tools still require users to possess algorithmic thinking and debugging skills. It is currently unclear which type of hint is most helpful and how this depends on the program representation and the user's understanding level. Our study aimed to address this gap by investigating the effectiveness of different hint types across various program representations and user understanding levels.

We conducted a large-scale, crowd-sourced study involving 753 participants to examine the utility of different program representations and hint types in finding and fixing bugs for participants with varying levels of understanding. Specifically, our research focused on three key questions: how program representation affects a user's ability to find and fix bugs, the utility of hints across different program representations, and which hint types are most suitable for various program representations, all in relation to the user's level of understanding of the algorithmic task.

Our findings revealed several important insights:

1) Text-based program representations improve accuracy for users with clear understanding of the algorithmic task.
2) Hints significantly improved accuracy for Python representations across both clear and confused understanding groups. For text-based representations, hints reduce response time for those with clear understanding. Additionally, hints bridged accuracy gaps between different representations and understanding levels.
3) Different hint types vary in utility depending on program representation and participant understanding. Detailed fixes generally improved accuracy most, while test cases reduce response time for participants with clear understanding viewing text-based representations.

These results have significant implications for the design of next-generation programming tools. They suggest the potential for personalized support systems that adapt to users' experiences and understanding levels. By tailoring program representations and hint types to individual users, we can enhance their ability to find and fix bugs, ultimately improving their programming skills and reducing response time. Additionally, for software engineering researchers, our work represents the first study to systematically investigate the intersection of different programming representations and types of hints. Our methodology and study setup provide a blueprint for exploring similar and follow-up research questions, some of which we articulate in Section VI in the form of explicit hypotheses.

VIII. Data Availability

The raw data from the human study and scripts to generate all the plots present in the paper are available at https://github.com/bridge-ai-neuro/HintsCodeText.

Acknowledgements

The authors would like to thank Norman Peitek, Anna-Maria Maurer, Tung Phung, Ahana Ghosh, Gabriele Merlin, Emin Çelik, and Aritra Mitra for their feedback on our Cloud Research survey workflow. Financial support was provided by the German Research Foundation through Collaborative Research Center TRR 248 (389792660), by the European Union as part of ERC Advanced Grant "Brains On Code" (101052182), and by the European Union as part of ERC Starting Grant "TOPS" (101039090).

References

[1] GitHub, "GitHub Copilot," https://github.com/features/copilot, 2021.

[2] OpenAI, "ChatGPT," https://openai.com/blog/chatgpt, 2023.

[3] N. Callinan and M. Perry, "Critical success factors for citizen development," *Open Journal of Applied Sciences*, vol. 14, no. 4, pp. 1121–1149, 2024.

[4] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: Finding, Fixing and Flailing, a Multi-institutional Study of Novice Debuggers," *Computer Science Education*, vol. 18, 2008.

[5] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: A Review of the Literature from an Educational Perspective," *Computer Science Education*, 2008.

[6] C. Li, E. Chan, P. Denny, A. Luxton-Reilly, and E. D. Tempero, "Towards a Framework for Teaching Debugging," in *Proceedings of the Australasian Computing Education Conference (ACE)*, 2019.

[7] J. Whalley, A. Settle, and A. Luxton-Reilly, "Novice Reflections on Debugging," in *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*, 2021.

[8] J. Prather *et al.*, "The Robots Are Here: Navigating the Generative AI Revolution in Computing Education," in *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR)*, 2023.

[9] P. Denny, S. Gulwani, N. T. Heffernan, T. Käser, S. Moore, A. N. Rafferty, and A. Singla, "Generative AI for Education (GAIED): Advances, Opportunities, and Challenges," *CoRR*, vol. abs/2402.01580, 2024.

[10] J. Leinonen, A. Hellas, S. Sarsa, B. N. Reeves, P. Denny, J. Prather, and B. A. Becker, "Using Large Language Models to Enhance Programming Error Messages," in *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*, 2023.

[11] S. Wang, J. C. Mitchell, and C. Piech, "A Large Scale RCT on Effective Error Messages in CS1," in *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 2024, pp. 1395–1401.

[12] T. Phung, V. Padurean, J. Cambronero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla, and G. Soares, "Generative AI for Programming Education: Benchmarking ChatGPT, GPT-4, and Human Tutors," in *Proceedings of the Conference on International Computing Education Research (ICER) - Volume 2*. ACM, 2023, pp. 41–42.

[13] T. Phung, V. Padurean, A. Singh, C. Brooks, J. Cambronero, S. Gulwani, A. Singla, and G. Soares, "Automating Human Tutor-Style Programming Feedback: Leveraging GPT-4 Tutor Model for Hint Generation and GPT-3.5 Student Model for Hint Validation," in *Proceedings of the Learning Analytics and Knowledge Conference (LAK)*. ACM, 2024, pp. 12–23.

[14] N. A. Kumar and A. S. Lan, "Using Large Language Models for Student-Code Guided Test Case Generation in Computer Science Education," *AI4ED Workshop at AAAI*, 2024.

[15] H. Heicka and A. S. Lan, "Generating Feedback-Ladders for Logical Errors in Programming using Large Language Models," in *Proceedings of International Conference on Educational Data Mining (EDM)*, 2024.

[16] Y.-F. Liu, J. Kim, C. Wilson, and M. Bedny, "Computer code comprehension shares neural resources with formal logical inference in the fronto-parietal network," *Elife*, vol. 9, p. e59340, 2020.

[17] Y.-F. Liu, C. Wilson, and M. Bedny, "Contribution of the language network to the comprehension of python programming code," *Brain and Language*, vol. 251, p. 105392, 2024.

[18] A. A. Ivanova, S. Srikant, Y. Sueoka, H. H. Kean, R. Dhamala, U.-M. O'reilly, M. U. Bers, and E. Fedorenko, "Comprehension of computer code relies primarily on domain-general executive brain regions," *elife*, vol. 9, p. e58906, 2020.

[19] B. Floyd, T. Santander, and W. Weimer, "Decoding the representation of code in the brain: An fmri study of code review and expertise," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 175–186.

[20] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "Eye movements in code reading: Relaxing the linear order," in *2015 ieee 23rd international conference on program comprehension*. IEEE, 2015, pp. 255–265.

[21] Z. Karas, A. Jahn, W. Weimer, and Y. Huang, "Connecting the dots: rethinking the relationship between code and prose writing with functional connectivity," in *Proceedings of the ACM Joint Meeting on European Software Eng. Conference and Symposium on the Foundations of Software Eng.*, 2021, pp. 767–779.

[22] P. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, and P. Daleiden, "An empirical study on the impact of C++ lambdas and programmer experience," in *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2016, pp. 760–771.

[23] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 19:1–19:40, 2013.

[24] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck, "Object-oriented program comprehension: Effect of expertise, task and phase," *Empirical Software Engineering*, vol. 7, no. 2, pp. 115–156, 2002.

[25] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man–Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[26] F. Détienne, *Software Design—Cognitive Aspects*. Springer, 2002.

[27] O. Dieste, A. Aranda, F. U. Uyaguari, B. Turhan, A. Tosun, D. Fucci, M. Oivo, and N. Juristo, "Empirical evaluation of the effects of experience on code quality and programmer productivity: An exploratory study," *Empirical Software Engineering*, vol. 22, no. 5, pp. 2457–2542, 2017.

[28] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated Feedback Generation for Introductory Programming Assignments," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2013, pp. 15–26.

[29] S. Gulwani, I. Radicek, and F. Zuleger, "Automated Clustering and Program Repair for Introductory Programming Assignments," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2018, pp. 465–480.

[30] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, 2017.

[31] A. Head, E. L. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann, "Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis," in *Proceedings of the Conference on Learning @ Scale (L@S)*. ACM, 2017, pp. 89–98.

[32] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What Would Other Programmers Do: Suggesting Solutions to Error Messages," in *Proceedings of the International Conference on Human Factors in Computing Systems (CHI)*, 2010.

[33] A. Al-batlaa, M. Abdullah-Al-Wadud, and M. A. Hossain, "A Review on Recommending Solutions for Bugs Using Crowdsourcing," in *Saudi Computer Society National Computer Conference (NCC)*, 2018.

[34] T. Phung, J. Cambronero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla, and G. Soares, "Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models," in *Proceedings of the Conference on Educational Data Mining (EDM)*, 2023.

[35] N. Peitek, A. Bergum, M. Rekrut, J. Mucke, M. Nadig, C. Parnin, J. Siegmund, and S. Apel, "Correlates of programmer efficacy and their link to experience: A combined eeg and eye-tracking study," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 120–131.

[36] J. Middleton, J.-P. Ore, and K. T. Stolee, "Barriers for students during code change comprehension," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[37] geeksforgeeks.org, "GeeksforGeeks: A Computer Science Portal for Geeks," https://www.geeksforgeeks.org/, 2009.

[38] K. Fisler, "The recurring rainfall problem," in *Proc. of the Conference on International Computing Education Research (ICER)*, 2014.

[39] P. Denny, D. H. S. IV, M. Fowler, J. Prather, B. A. Becker, and J. Leinonen, "Explaining Code with a Purpose: An Integrated Approach for Developing Code Comprehension and Prompting Skills," *CoRR*, vol. abs/2403.06050, 2024.

[40] V. Padurean, P. Denny, and A. Singla, "BugSpotter: Automated Generation of Code Debugging Exercises," *CoRR*, vol. abs/2411.14303, 2024.

[41] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 196–202.

[42] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[43] J. Cohen, *Statistical power analysis for the behavioral sciences*. routledge, 2013.

[44] Quixy Editorial Team, "Game-changing top 60 no-code low-code & citizen development statistics," Feb 2024. [Online]. Available: https://quixy.com/blog/no-code-low-code-citizen-development-statistics-facts/
[45] R. Brooks, "Towards a theory of the comprehension of computer programs," *Journal of Man–Machine Studies*, vol. 18, no. 6, pp. 543–554, 1983.
[46] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on Software Engineering*, vol. 10, no. 5, pp. 595–609, 1984.
[47] B. Shneiderman and R. Mayer, "Syntactic/semantic interactions in programmer behavior: A model and experimental results," *Journal of Parallel Programming*, vol. 8, no. 3, pp. 219–238, 1979.
[48] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology*, vol. 19, no. 3, pp. 295–341, 1987.