How Do Learners With Varying Skills Perceive Misconception Indicators and Feedback?

Abigail Evans
abi.evans@york.ac.uk
University of York
York, United Kingdom

Daniel Lock
daniel.lock@york.ac.uk
University of York
York, United Kingdom

Abstract

Misconceptions about programming concepts can block learners' progress. Automated feedback for misconceptions is appealing because providing feedback while learners work may help them get unstuck faster. However, a key challenge for automating misconception feedback is that the underlying cause of code patterns suggestive of a misconception can vary greatly by learner, meaning that different learners will have different feedback needs. Existing approaches to automated feedback for task-independent misconceptions favour succinct messages that do not overload users with excessive explanation. Although this approach may work well for learners with relatively shallow misconceptions, it also leaves out learners with deeper conceptual issues who arguably have greater need for additional support. We conducted a qualitative study to investigate how learners perceive misconception indicators and how they make sense of feedback. We find that individual learners can view the same issue very differently and face markedly different challenges in making use of feedback. These findings can be used to inform the design of automated feedback that accounts for learners' varying knowledge and skills.

CCS Concepts

• **Applied computing → Education**; • **Human-centered computing → User studies**.

Keywords

novice programmers, misconceptions, automated feedback

ACM Reference Format:
Abigail Evans and Daniel Lock. 2025. How Do Learners With Varying Skills Perceive Misconception Indicators and Feedback?. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2025), June 27-July 2, 2025, Nijmegen, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3724363.3729073

1 Introduction

Feedback is critical for learning in any context [16]. Error messages are an important source of feedback during programming, and progress has been made to make them more useful to novices [4, 10, 11]. However, compiler errors do not fully encapsulate the range of errors that novices make. Style issues may stem from deeper conceptual problems [9] and permissive languages like Python and JavaScript bring their own challenges for novices [17]. Extending IDE support for errors to include potential misconceptions may help learners progress faster. A challenge for feedback in standard IDEs is that the experience and skill level of a user is unknown.

In standard IDEs, error messages are written for experienced programmers. Guidelines for the design of learner-friendly error messages state that messages should be concise and free of jargon to improve readability and reduce cognitive load [4]. Because it is not possible to know the exact cause of an error from code alone [9, 19], some argue that messaging should avoid trying to scaffold users' understanding so as not to overwhelm them [10].

Although we agree it is important not to overload learners with excessive information, concise messaging may not help learners with more significant misconceptions. Error messages fall under Hattie and Timperley's category of task feedback, also known as corrective feedback [16]. Hattie and Timperley state that corrective feedback is important but caution that it "is more powerful when it is about faulty interpretations, not lack of information. If students lack necessary knowledge, further instruction is more powerful than feedback information" [16, p. 91]. This implies that a *differentiated* approach to automated feedback may be useful, in which learners with different needs can access different forms of feedback, such as succinct guidance for learners with faulty interpretations and further instruction in the case of deeper issues.

In this work, we focus on how learners interpret misconception indicators—code patterns that may suggest an underlying conceptual issue—and the challenges they encounter in making use of misconception-focused feedback. Our goal is to map the design space that differentiated misconception feedback will need to address. We are guided by the following research questions:

RQ1 How do learners perceive misconception indicators?
RQ2 How do learners interpret indicator feedback?

To explore these questions, we asked learners to respond to code samples containing misconception indicators with and without the assistance of feedback. We were interested in understanding variation among learners rather than commonalities. We found that learners perceive the same indicators in different ways with sometimes contradictory implications for feedback design. Learners with strong conceptual knowledge were able to make use of brief messages while those with more significant challenges benefitted from extra guidance.

The contributions of this work are qualitative insights into the variations in learner understanding of misconception indicators

ITiCSE 2025, June 27-July 2, 2025, Nijmegen, Netherlands
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1567-9/2025/06
https://doi.org/10.1145/3724363.3729073

and feedback needs. Our findings will be of interest to designers of automated feedback and educators of mixed-experience cohorts.

2 Related Work

De Ruvo et al. [9] describe statement-level code patterns that compile but are typically avoided by more experienced programmers as *semantic indicators*, which may point to shaky conceptual knowledge. Our term *misconception indicator* includes these semantic indicators but we use a broader term because we aim to provide feedback on potential misconceptions that may also lead to errors.

Prior work has documented and described misconceptions through extensive inventories [6, 13, 24]. Chiodini et al. [7] maintain a website with detailed misconception descriptions and suggestions for how teachers can tackle these misconceptions with students. SIDE-lib [12] is a library rather than an inventory but it provides detection of a set of Python misconception indicators.

Inventories describe how a particular misconception can manifest in code. However, we cannot know for sure that an indicator represents a misunderstanding rather than some other issue [9]. Misconception indicators generally feature undesirable code patterns so feedback is likely useful in any case, but learners with different causes may need different forms of feedback [16]. Therefore, we shift the focus from analysing the indicators themselves to analysing learners' views of those indicators to inform a nuanced approach to feedback design. In contrast to prior work such as Albrecht and Grabowski [1] and Bayman and Mayer [3], in which mistakes are categorized by experts into fixed, exclusive categories such as syntactic or conceptual, we look for themes in how *learners* describe indicators in order to shape differentiated feedback.

Much of the prior work on automated feedback for misconceptions requires knowledge of the learner's task or an expert solution (e.g. [15, 23]). These approaches can tackle strategic and logic issues as well as programming language misconceptions but they are constrained to specific tasks and are therefore out of scope for this summary. One exception is Pedal [14], a Python library for auto graders, including built-in feedback for misconception indicators in the form of concise messages.

Prior work on the design of effective error messages is relevant to the design of misconception indicator feedback [4, 10, 11]. Becker et al. [4] reviewed the literature on compiler error messages and the challenges they pose for learners, including terse and confusing language, poor localization, and lack of guidance on how to resolve the error. They developed 10 design guidelines such as *increase readability* by using lay language and *reduce cognitive load* by providing just enough information to resolve the issue. Error messages designed according to these guidelines have been shown to be more effective than standard messages [10]. This prior work has evaluated redesigned error messages by aggregating students' views and actions. We take a different approach and look for differences in how individuals respond to the same feedback in order to better understand their needs, preferences, and challenges.

Large Language Models (LLMs) such as ChatGPT and Codex can generate contextualized feedback and provide explanations for errors. The quality of LLM feedback has improved rapidly but they are still prone to giving incorrect explanations and providing direct solutions rather than guiding learners to develop their own understanding. As such, researchers and educators remain wary of recommending them for use without careful scaffolding [2, 8]. Additionally, learners may not know how to prompt an LLM effectively [20] and can struggle to interpret its output [20, 22]. Some interventions have avoided these drawbacks by scaffolding learners' interactions with the models, for example by pre-prompting the LLM to communicate in a particular manner [8, 18], or by providing it with additional contextual information such as failed unit tests [21]. We do not use LLMs in our feedback prototype because our research goals demand more control over what the participants interact with. However, we expect our findings may be useful to those designing scaffolded AI tools for educational use.

3 Methods

We conducted semi-structured interviews with 15 participants from our university in the United Kingdom. All activities were reviewed and approved by our departmental ethics committee.

12 participants were Computer Science BSc students taking an introductory programming module at the time of the interviews. The module teaches programming fundamentals using Python and assumes no prior experience. However, nine had studied computer science in high school, two were self-taught, and only one had no prior experience. Seven of the students identified as male, four as female, and one did not specify. All of these participants reported using standard IDEs, primarily VSCode and Pycharm.

Three participants were scientists learning Python to support data analysis and other aspects of their work. These participants were recruited to reflect the diversity of what it means to be a learner programmer—not all learners are engaged in formal education. All of the scientists had experience with other languages used in scientific computing (R, Stata, and Praat) but described themselves as beginners with respect to Python and programming. Two scientists identified as female and one as male. The IDs of scientist participants have the suffix *sci*, all others are CS students. Two of the scientists reported using standard IDEs and one did not specify.

During the interview, participants were asked to respond to preselected code samples. We did not ask participants to write code themselves because the occurrence of misconception indicators is hard to predict. Samples were selected from a public dataset [23] of Python code written by students in an introductory programming course. This dataset was chosen because the code snapshots it contains are brief (roughly 5 - 10 lines of code).

To select samples, we first chose a set of misconception indicators detected by SIDE-lib [12] that were present in the dataset, covered a range of concepts, and had varying impact on program output. The indicators are described in Table 1. The selected code samples contained at least one indicator, responded to a range of task prompts, and represented a mix of correct and incorrect program output.

A web interface was created for the study[1]. The code sample was displayed in a functional code editor. The description of the programming task was shown above the editor. SIDE-lib[12] was used to detect misconception indicators. For simplicity, we only implemented feedback for indicators in Table 1. Feedback was generated at every keystroke, updating if edits introduced or removed

[1]https://supportive-ide.hosted.york.ac.uk/details/feedback-evaluation1.html

Table 1: The misconception indicators. Names of misconceptions documented in [9, 13] have been changed to follow the convention used in [7, 12]. The Theme column lists categorising themes (Table 2) in participants' comments. Understanding # shows the count of participants who appeared to understand an indicator pre- vs. post-feedback. N/A means no understanding.

Indicator	Themes	Understanding #		
		Pre	Post	N/A
CompareMultipleWithOr: Tests equality of multiple values in the form `var == val1 or val2` [12]	1, 2, 3	2	8	0
ConditionalIsSequence: A sequence of if statements is the same as `if-elif-else` [7]	1, 4a	4	3	0
DeferredReturn: Believes code after a return statement will still execute [7]	1, 3	7	3	0
FunctionCallsNoParentheses: Parentheses omitted from a function call [12]	2, 3a	2	7	1
LocalVariablesAreGlobal: Attempts to access a local variable in global scope [13]	1	10	2	1
MapToBooleanWithIf: `if-else` must be used to map a boolean expression to a boolean value [7, 9]	1, 4a, 4c	0	8	1
ParamMustBeAssignedInFunction: Passed in value of a parameter is immediately overwritten [13]	1, 4b	5	1	0
TypeMustBeSpecified: A value with guaranteed type (e.g. a literal) is cast to the same type [9, 12]	1, 3, 3a, 4a	1	9	0
UnusedReturn: The value returned by a function is not saved or used [13]	1, 3a	8	4	1

indicators. The interviewer could toggle feedback on and off and change the sample in the participant's view via a separate webpage.

The Becker et al. [4] error message readability guidelines informed the feedback design. The entry point to the feedback is a highlight in the code, following the *provide context* guideline. Hovering over the highlight opens a popover. The popover provides the minimum amount of information required to address the issue (*reduce cognitive load*). If more information is needed, clicking the popover opens extended guidance below the editor. The extended guidance begins with a short explanation, drawing on the user's own code (*provide context*) and *showing examples*. A longer instructional section walks through the underlying concept in more depth to *provide scaffolding* for users that need it. Text is broken up with runnable examples. We do not *show solutions* to the user's problem directly but do suggest generic solutions via the interactive examples. All text was written to *use a positive tone* and *increase readability* by using lay language as much as possible.

Interviews were conducted on Zoom and lasted between 30 minutes and 1 hour depending on the participant's pace. The interview began with questions about experience and motivation to learn programming. Next, we introduced the prototype. Samples were presented one at a time in random order. The participant was asked to review the code and identify anything they thought should be changed. Each sample was initially presented without feedback. If the participant did not find the indicator independently, feedback was enabled and the participant was given time to interact with it to find out how they used it and if they were able to fix the indicator as a result. If the participant was still unable to fix the indicator, the interviewer followed up to try to understand their thought process. Participants were told to take as long as they wanted on each code sample and were asked if they wished to continue after each sample past 30 minutes. Nine code samples were available but we expected that most would complete fewer.

3.1 Data Analysis

The interview data was analysed using Reflexive Thematic Analysis (RTA) [5], a widely used qualitative method to identify themes in data. We chose RTA over other methods because of its flexibility in allowing deductive and inductive analysis. RTA is distinct from other forms of thematic analysis in its acknowledgment of the researcher's perspective as an integral part of the analysis and a resource rather than a source of bias. RTA does not use inter-rater reliability or develop a code book, but instead requires the researcher's identity and perspective to be surfaced and reflected upon in the context of the data throughout the analysis, which is systematic and iterative. Reliability is gained through the cyclical analysis process.

Neither of the authors have been involved in the delivery of the module that most of our participants were recruited from but both have substantial experience of teaching introductory programming. The first author created the prototype and conducted the interviews. Although RTA recommends that analysis is typically carried out by one person [5], both authors were involved in the analysis. Given the prototype's role as a probe, we felt it important to include the perspective of someone removed from its design (second author).

We followed the six steps of RTA outlined in [5]. The first author familiarised herself with the data by reading the transcripts and rewatching videos of participants' interactions with the prototype. The first author carried out two iterations of inductive and deductive coding using NVivo and Miro. These codes were then discussed with the second author, who conducted a third iteration of coding after familiarising himself with the data. The first author then undertook iterative theme generation and development, followed by theme refinement through the writing process, drawing together insights from both researchers.

4 Results

The computer science students completed six samples on average (std. dev. = 1.28). The scientists completed fewer—one completed two samples and the others completed three. Table 1 (Understanding #) shows the number of participants who saw each indicator by when it was apparent they understood it, including four cases where unplanned indicators arose.

In the sections below, we discuss our themes by research question. RTA is a "big Q" qualitative method and requires that themes are *not* quantified [5]. To qualify as a theme in this analysis, relevant codes were observed in multiple interviews.

4.1 RQ1: Perception of Indicators

The themes and sub-themes developed in response to RQ1 are shown in Table 2, themes 1-4.

4.1.1 Indicator categorisation. Participants were not prompted to categorise indicators in terms of their nature or cause but their explanations often suggested implicit categorisation, represented by themes 1-3 in Table 2, which are drawn from the error categories in [1], plus a fourth inductive theme, *practice*. Not all participants categorised indicators, but among those that did, we observed divergent perspectives for all but one indicator, (*LocalVariablesAreGlobal*, which was consistently described in *conceptual* terms).

The largest divergence among individual participants was observed for *DeferredReturn*. P05 reported a genuine misconception after reading the feedback: “I didn’t actually know that it would exit a function off the return.” In contrast, P08 describes this indicator as *sloppiness*: “if you were to encounter a mistake like this, and it would probably just be a mistake, like a typo, or something.”

Sloppiness is closely related to the *conceptual* and *syntactic* themes; the primary difference being understanding. Conceptual and syntactic errors happen when there is something you don’t know or don’t understand whereas sloppiness occurs when you forget or overlook something you do know. As demonstrated by the *DeferredReturn* example, something that is clearly a typo to someone who understands the underlying concept may be more significant to someone who is not at the same level of understanding.

Sometimes participants went further than acknowledging an oversight and spoke about how a concept could be known so well it makes related errors almost *invisible*:

> “I didn’t spot that because that’s one of those tiny mistakes that I think are very difficult to actually spot, simply because you either do them by habit or once you don’t do it you don’t realize, because it’s such a habit. You just wouldn’t think that you’d miss something like that.” P05, *FunctionCallsNoParentheses*

The *conceptual* theme arose from participants’ descriptions of certain concepts as particularly challenging, as well as cases of actual misconceptions, such as P05’s belief that code following a return would still execute. The *syntactic* theme was most often observed in comments about switching to Python from other languages. An interesting finding was that two of the scientists described the syntax of programming as harder than the semantics and logic. For example, P10sci said, "Conceptually, a lot of it makes sense, you know, for loops, if statements, that kind of thing...The thing I’m yet to overcome is just getting used to the syntax."

The deductive categories drawn from Albrecht and Grabowski [1] describe participants’ views of the cause of an issue. The *practice* theme is better described as the relevance of an indicator—the reason given for correcting it. Although relevance is semantically different from cause, participants’ comments suggested they viewed it as a distinct category—things you *should* do rather than things you *must* do. This theme arose from analysis of indicators which don’t produce error messages and rarely cause incorrect output, and are therefore perhaps less urgent than others.

ConditionalIsSequence was often described as a *practice* issue. The sample code contained a sequence of if statements which were not mutually exclusive. Although some participants described it as a conceptual issue, the sub-theme *optimization* arose multiple times. For example, P06 simply stated "efficiency" when asked why she would change some `ifs` to `elifs`. P04 noted "you can optimize the code with else if statements".

Another sub-theme of *practice* is code *style*. This sub-theme arose in discussion of *MapToBooleanWithIf*. The sample code is correct but contains an `if-else` that simply returns `True/False`, meaning it can be replaced with a single statement returning the boolean expression. None of the participants spotted this indicator on their own but after reading the extended guidance, some brought up style. For example, P11 rejected the idea of shortening the conditional: “I like the structure, like seeing the indentation.” In contrast, P09sci had a genuine misconception, declaring “I’m learning a lot today!” after reading the guidance.

4.2 RQ2: Use of Feedback

Themes 5 and 6 in Table 2 were developed in response to RQ2.

4.2.1 Information needs. The *information needs* theme captures participants’ comments about what they liked about the feedback or what they would have preferred to have seen.

Some participants wanted to see an actionable *solution first*, with any additional explanation later—they wanted the feedback to help them fix the code quickly and move on:

> “I like the different colours where you have what was missing...I think it makes like a more immediate, almost action point...You know immediately what you have to do to fix it.” P12

In contrast, other participants thought feedback should *hide the solution* at least at first. For example, P07 suggested, “I think get at the problem first rather than solutions and then they can maybe try it themselves again.”

The varying needs of *beginner vs. experienced* users was often brought up by participants with solid conceptual understanding of an issue. In particular, participants noted that the extended feedback might benefit those with conceptual issues but would be unnecessary for those with good understanding:

> “The examples aren’t really necessary, because I know why the brackets are necessary but I think for someone who doesn’t know why those brackets would need to be, these would be really good.” P05, *FunctionCallsNoParentheses*

On the other hand, some participants felt that the extra guidance could be useful for experienced users, particularly in the case of *sloppiness*. The key difference between the two groups was that experienced participants could handle (and preferred) much shorter and more succinct explanations.

4.2.2 Feedback effect. For participants who missed an indicator due to *sloppiness*, the feedback served as a simple *reminder* or confirmation of what they already knew. For these participants, the brief explanation in the popover was often enough although some did proceed to the extended feedback before showing that they understood the issue.

Several participants commented that extended feedback provided *learning* opportunities for less experienced coders and others

Table 2: Themes and sub-themes. The first three themes are adapted from Albrecht and Grabowski [1]. Themes 1-4 apply to both RQs but primarily RQ1. Themes 5 and 6 apply to RQ2.

Theme	Definition
1. Conceptual	An indicator is attributed to a misunderstanding of a programming concept, independent of syntax.
2. Syntactic	An indicator is attributed to incorrect knowledge of syntax. The underlying concept is understood.
3. Sloppiness	An unintended oversight or typo. The underlying concept is understood and correct syntax is known.
a) Invisibility	A code detail that is such a habit it is easily overlooked when forgotten, such as remembering to call a function or type parentheses. A reason given for sloppiness.
4. Practice	Matters of coding practice rather than correctness or error-free code.
a) Optimization	An indicator affects code efficiency or is described as a more general optimization.
b) Style	An indicator is a matter of code style, which may be a formal requirement or personal preference.
5. Information needs	What participants wanted from the feedback, or thought it should provide.
a) Solution first	Understanding how to fix an issue is prioritised over understanding why.
b) Hide solution	Explain the problem and hide the solution so the user can attempt to fix it on their own.
c) Beginner vs. experienced	Users at different stages may benefit from different amounts and styles of information, with users who understand a concept needing far less.
6. Feedback effect	How the feedback helped participants recognise and understand an indicator.
a) Reminder	The feedback is a reminder of something the participant already knows.
b) Learning	The feedback helps to fill in small gaps in knowledge or provide motivation.
c) Partial understanding	Participant understood aspects of the indicator but not enough to address it.

directly stated that they had learned from it. Some noted that it addressed the challenge of trying to find external resources when there was no error message:

> “I wouldn’t know what to search up for, like what I’ve done wrong...because, well, in the log, it wouldn’t really, you know, say, there was an error here or anything before... This would save me, probably like an hour, if not more. ” P13, *CompareMultipleWithOr*

When extended feedback was needed, examples were often more useful for learning than accompanying text. The interactive examples in the extended feedback was typically structured as a before and after in order to demonstrate the effects of an issue, which proved popular with participants who had genuine conceptual issues. P09sci, who did not understand the *TypeMustBeSpecified* indicator until after working through the interactive examples, highlighted the value of seeing them alongside text explanation:

> “It’s nice to have this kind of explanation and then a really clear comparison, as well, between what happens in the first example, when you don’t [type cast], and then a visual comparison as well between the code and the output of both examples.” P09sci

There were four cases where the extended feedback helped participants with genuine misconceptions reach only *partial understanding*. One of these participants was the true beginner among the CS students, P14, who was unable to apply the generic example solutions in the *UnusedReturn* feedback to the indicator in the code sample. The other cases of *partial understanding* occurred with two of the scientists. Both of these participants had extremely fragile or missing knowledge of concepts used in the code samples, which may be due to the very different way in which they learn and use programming in comparison to the computer science students. For example, P10sci reported learning concepts out of the typical order, skipping certain fundamentals in order to solve an immediate problem. He also noted that, "there are certain things that you do so infrequently that it’s very easy to forget."

5 Discussion

The themes outlined above show considerable variation among participants in terms of how they viewed individual misconception indicators and what they wanted or needed from the feedback. In this section, we discuss the takeaways from these findings for the design of automated feedback for misconception indicators.

5.1 Misconception indicator feedback is useful

All participants were enthusiastic about the provision of IDE-based feedback for misconception indicators, especially when the affected code would not cause an error message. In most cases, participants who did not spot an indicator did not hold an associated misconception but they were still appreciative of the reminder, typically citing time saved or cleaner code.

We also note that two indicators (*ConditionalIsSequence* and *MapToBooleanWithIf*) proved controversial with some participants who viewed them as *practice* issues and thought they should be communicated differently. Other participants saw these indicators as *conceptual* with one case of a genuine misconception which, in our view, is good reason to provide feedback. Future work should consider how the UX of a feedback system can prioritise and communicate differences between indicators.

5.2 Beware of dismissing mistakes as sloppiness

The category of *sloppiness* was introduced by Albrecht and Grabowski [1] who describe it as a distinct category, separate from *conceptual*, *syntactic*, and other mistakes. One of the main takeaways from this work is that *sloppiness* may be a progression from other categories—to be sloppy, a learner must have reached a certain level of mastery of the concept or syntax. We saw a stark example of this in discussion of the *DeferredReturn* indicator, where one participant held a genuine misconception while others viewed it as a sloppy mistake. As shown in Table 1, there were several other indicators that some participants described as *sloppiness* while others saw *conceptual* or *syntactic* issues.

Divergence in terms of how participants categorised indicators or experienced genuine misconceptions adds further support for the idea of differentiated feedback that enables those who have made simple mistakes to quickly correct them and move on while at the same time helps those with more significant conceptual issues to fill in the gaps in their knowledge. This finding relates to Becker et al.'s error message guidelines [4], specifically *reduce cognitive load*, which our findings suggest looks different for participants with and without conceptual issues.

5.3 Learners with solid conceptual knowledge want short and sweet feedback

Feedback served as a *reminder* for those who did not have conceptual issues relating to an indicator. These participants often expressed a preference for succinct messaging that enabled them to quickly understand the issue and find a fix. For these learners, *reduce cognitive load* means keep messaging brief and actionable, as we have seen in prior research [4, 11]. While many participants in this group appreciated having the option to dig deeper into an issue via the extended feedback and acknowledged that it was easy to ignore if they didn't need it, they also described it as being for coders less experienced than themselves (*beginner vs. experienced).* In our ongoing design work on differentiated feedback for misconception indicators, we will continue to refine the initial message that users see (e.g. the popover in our prototype) to target users who have simply overlooked something.

5.4 Learners with conceptual issues may benefit from more explanation

The true measure of success for feedback on misconception indicators is whether or not it supports *learning* for those with genuine misconceptions. Where conceptual issues were evident, the brief message in the popover was not enough. The instructional component of the extended feedback was helpful for many, with some participants explicitly stating they had learned something new. The participants found the interactive examples particularly useful, which aligns with the Becker et al. guidelines *show examples* and *provide scaffolding for users* [4]. Our findings lend initial support to the idea that learners with conceptual issues may need more explanation, which was one of the initial motivations for this work.

Some participants were not able to reach sufficient levels of understanding to act on an indicator based on the feedback. It is notable that these participants were also arguably the weakest coders: one participant who was only a few months in to learning to code, and two of the scientists who had gaps in their knowledge relating to the sample code. Our initial thinking about these cases is that the learners simply did not have sufficient base knowledge to make sense of the guidance as it was presented. While we may be able to address some of their challenges through further design iterations of the prototype, it is likely that more research focused specifically on the needs of learners with knowledge gaps is needed to develop a deeper understanding of how automated systems can best support them. The scientists' struggles also serve as a reminder that learner programmers are not only found in computer science courses.

5.5 Implications for automated feedback design

For pre-programmed tools such as our prototype, the primary implications and challenges are for the UX design. We suggest providing the shortest, most direct, and actionable feedback first, then allowing those who need it to dig deeper into the underlying concepts. A significant UX challenge is how to present the deeper layers of guidance in a way that invites engagement from those who need it without overwhelming those who don't. Our goal is to embed differentiated feedback in a standard IDE, therefore one approach may be to allow users to customise feedback presentation through IDE settings. For example, users could have the option to turn off extended guidance.

An interesting avenue for future work would be to explore if LLM-driven feedback could provide differentiation, for example, by learning a user's experience level through dialogue. Users could also be encouraged to treat their interactions with an LLM as a discussion rather than a means to a quick solution.

5.6 Limitations

The main limitation of this work is that participants were responding to code samples rather than writing their own code, which means that the indicators were presented in a somewhat artificial manner and participants were sometimes faced with code that did not resemble how they would solve the given task. A longitudinal deployment is planned for the next iteration of the prototype in order to capture what happens when indicators arise naturally and to measure the effectiveness of misconception indicator feedback for learning, which could not be captured in this study.

Although the design of the prototype used in this study was informed by established error message guidelines, it was apparent that refinements were needed in some places. Evaluation of the prototype was not the main purpose of this work but participants provided us with feedback on what would have made it more usable and accessible from a learning point of view, which will be incorporated into future iterations of the prototype.

6 Conclusion

We have presented a qualitative study of how learners of varying skill levels perceive task-independent misconception indicators and automated feedback for these indicators. Our findings provide evidence that it is useful to give feedback for these indicators but that learners can have divergent views of what an indicator represents and why it matters depending on their skill level and the context. We also find that adding an instructive component to feedback may be useful to those who have conceptual misunderstanding of an indicator but further work is needed to explore how automated feedback can best support these learners at the same time as those with stronger conceptual knowledge.

References

[1] Ella Albrecht and Jens Grabowski. 2020. Sometimes It's Just Sloppiness - Studying Students' Programming Errors and Misconceptions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, Portland OR USA, 340–345. https://doi.org/10.1145/3328778.3366862

[2] Imen Azaiz, Natalie Kiesler, and Sven Strickroth. 2024. Feedback-Generation for Programming Exercises With GPT-4. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*. ACM, Milan Italy, 31–37. https://doi.org/10.1145/3649217.3653594

[3] Piraye Bayman and Richard E Mayer. 1988. Using Conceptual Models to Teach BASIC Computer Programming. *Journal of Educational Psychology* 80, 3 (1988), 291–298.

[4] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ACM, Aberdeen, Scotland, UK, 177–210. https://doi.org/10.1145/3344429.3372508

[5] Virginia Braun and Victoria Clarke. 2022. *Thematic Analysis: A Practical Guide*. Sage, London.

[6] Ricardo Caceffo, Breno de França, Guilherme Gama, Raysa Benatti, Tales Aparecida, Tania Caldas, and Rodolfo Azevedo. 2017. *An Antipattern Documentation About Misconceptions Related To An Introductory Programming Course In C*. Technical Report. Universidade Estadual De Campinas.

[7] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. 2021. A Curated Inventory of Programming Language Misconceptions. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ACM, Virtual Event Germany, 380–386. https://doi.org/10.1145/3430665.3456343

[8] Veronica Cucuiat and Jane Waite. 2024. Feedback Literacy: Holistic Analysis of Secondary Educators' Views of LLM Explanations of Program Error Messages. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*. ACM, Milan Italy, 192–198. https://doi.org/10.1145/3649217.3653595

[9] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B. Rowe, and Nasser Giacaman. 2018. Understanding Semantic Style by Analysing Student Code. In *Proceedings of the 20th Australasian Computing Education Conference*. ACM, Brisbane Queensland Australia, 73–82. https://doi.org/10.1145/3160489.3160500

[10] Paul Denny, James Prather, and Brett A. Becker. 2020. Error Message Readability and Novice Debugging Performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, Trondheim Norway, 480–486. https://doi.org/10.1145/3341525.3387384

[11] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. 2021. On Designing Programming Error Messages for Novices: Readability and its Constituent Factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–15. https://doi.org/10.1145/3411764.3445696

[12] Abigail Evans, Jieren Liu, Zihan Wang, and Mingming Zheng. 2023. SIDE-lib: A Library for Detecting Symptoms of Python Programming Misconceptions. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023)*. ACM Press, Turku, Finland. https://doi.org/10.1145/3587102.3588838

[13] Guilherme Gama, Ricardo Caceffo, Renan Souza, Raysa Benatti, Tales Aparecida, Islene Garcia, and Rodolfo Azevedo. 2018. *An Antipattern Documentation About Misconceptions Related To An Introductory Programming Course In Python*. Technical Report. Universidade Estadual De Campinas. 106 pages.

[14] Luke Gusukuma, Austin Cory Bart, and Dennis Kafura. 2020. Pedal: An Infrastructure for Automated Feedback Systems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, Portland OR USA, 1061–1067. https://doi.org/10.1145/3328778.3366913

[15] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-Driven Feedback: Results from an Experimental Study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, Espoo Finland, 160–168. https://doi.org/10.1145/3230977.3231002

[16] John Hattie and Helen Timperley. 2007. The Power of Feedback. *Review of Educational Research* 77, 1 (March 2007), 81–112. https://doi.org/10.3102/003465430298487

[17] David Liu and Andrew Petersen. 2019. Static Analyses in Python Programming Courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, Minneapolis MN USA, 666–671. https://doi.org/10.1145/3287324.3287503

[18] Rongxin Liu, Carter Zenke, Charlie Liu, Andrew Holmes, Patrick Thornton, and David J. Malan. 2024. Teaching CS50 with AI: Leveraging Generative Artificial Intelligence in Computer Science Education. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. ACM, Portland OR USA, 750–756. https://doi.org/10.1145/3626252.3630938

[19] Davin McCall and Michael Kölling. 2019. A New Look at Novice Programmer Errors. *ACM Transactions on Computing Education* 19, 4 (Dec. 2019), 1–30. https://doi.org/10.1145/3335814

[20] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)read Each Other. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–26. https://doi.org/10.1145/3613904.3642706

[21] Tung Phung, Victor-Alexandru Pădurean, Anjali Singh, Christopher Brooks, José Cambronero, Sumit Gulwani, Adish Singla, and Gustavo Soares. 2024. 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 14th Learning Analytics and Knowledge Conference*. ACM, Kyoto Japan, 12–23. https://doi.org/10.1145/3636555.3636846

[22] James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2024. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Transactions on Computer-Human Interaction* 31, 1 (Feb. 2024), 1–31. https://doi.org/10.1145/3617367

[23] Kelly Rivers, Erik Harpstead, and Ken Koedinger. 2016. Learning Curve Analysis for Programming: Which Concepts do Students Struggle With?. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, Melbourne VIC Australia, 143–151. https://doi.org/10.1145/2960310.2960333

[24] Renan Souza, Ricardo Caceffo, Pablo Frank-Bolton, and Rodolfo Azevedo. 2018. *An Antipattern Documentation About Possible Misconceptions Related To Introductory Programming Courses (CS1) In Java*. Technical Report. Universidade Estadual De Campinas, Brazil.