Symbolic annotations were difficult for some students to interpret. In order to succinctly summarize multiple hints visually, we used text strikethroughs (red for deletions, purple for replacements), and icons (a plus sign for insertions). However, students noted challenges interpreting the different colors and symbols. "*I actually didn't know the difference between the red and purple*" [P4] strikethroughs, one student noted. Another thought that the strikethrough should not be used to communicate replacements: "*I don't think you should cross out the ones that have suggestions*" [P3]. Similarly, the plus icon was not universally clear: "*I don't really understand the meaning of these cross marks*" [P2]. There was also concern that not all students would know to hover over annotations to read the tooltips: "*I think not alot of people are gonna hover on their own code*" [P4]. However, other students expressed no difficulty intuiting this, saying they hover "*all the time, whenever I get them*" [P1], navigating the annotations easily. The interface did provide a link to a help page explaining each annotation, but we never observed a student using this link. The inconsistent interpretation of the annotations suggests that feedback designers should use icons and text markup carefully. It also reinforces the need for pedagogy to support the use of hints, as one student suggested: "*in the first class when the professor is explaining [the programming environment] they could give some information*" [P6].

How much detail to include?

Students discussed our choice to avoid suggesting exact edits to make, showing only where to make a change and the type of code element to add:

Students needed more detail to use hints. Students appreciated how specific the hints' location information was, saying "*It's good because it tells you exactly what's wrong*" [P5]. However, for most students, the hints were too vague to be immediately actionable, "*if it could be just more elaborated I would have understood... it's also a little ambiguous*" [P2]. This was echoed in the surveys, saying hints were "*too vague*" [S] and "*not as specific compared to in-person hints*" [S]. The fact that students found the hints to be vague is likely due to our decision not to suggest exact edits. Students also wanted more explanation of *why* a hint made a suggestion, especially when hints contradicted their existing plan, "*it says... to add a binary operation. I'm like, 'Why?,' because creating a new list doesn't really require a binary operation*" [P1].

Withholding exact edits did not promote reasoning. Our rationale for avoiding suggestions of exact edits was to encourage students to reason through the hint, rather than following it blindly. In practice, students reported that the hints did not provide *enough* detail to promote learning: "*It doesn't really help engage the thought process so you get better. It's more of just saying, like, oh, this is wrong. This is wrong. This is wrong*" [P5]. This may be in part because novices lacked the prior knowledge to interpret hints. One student noted, "*it's very vague. I can't tell what I would need to do unless I already have sort of an idea prefixed in my mind*" [P4]. One way we tried to promote reasoning was by keeping hints fine-grained (i.e. suggesting one token at a time instead of a whole line), but as the instructor pointed out, students can still get that information, "*it's just that they're doing it in two steps instead of one.*" Students did report spending additional time trying to interpret hints, "*just sort of going through my whole thought process again because then it would be kind of confusing*" [P4], but this time did not seem to get them to an answer any more efficiently than students who did not have hints.

DISCUSSION

Our results suggest that certain design choices we made may have reduced the effectiveness of hints, and others involve trade-offs that must be balanced. Here we highlight key takeaways to inform the design of future systems.

Transformation hints should suggest precise next steps. One of our goals in CodeChecker was to promote reasoning and prevent help abuse by withholding the exact edit a student should make, showing them only where to edit and the general type of edit (e.g. "call a function" not "call `len()`"). Other tutoring systems similarly employ "pointing hints" and minimize "bottom-out" hints that give away the answer. Our results suggest that for *transformation hints* this may not be advisable, since they do not include human-authored explanations. We found that when the hints did not communicate *exactly* what edit to make, students found them vague or confusing and lacked sufficient detail to evaluate whether the hint was appropriate for their situation.

The system should prioritize hints for students. Drawing on prior work showing that system-selected hints could contradict students' own goals [2], we explored how to allow students to select a relevant hint, by displaying all available hints as annotations to their code. However, we found that this choice could be overwhelming for students, especially if many hints were available. It may be particularly difficult for *novices* to evaluate which hint is most useful, since they lack domain knowledge to begin with. Hint systems should therefore present a single, top-priority hint for the student to consider, to reduce uncertainty. However, students also appreciated the ability to browse alternative hints, if the original was not useful to them, suggesting that student choice should be supported, just not the *default*.

REFERENCES

[1] Thomas W. Price, Rui Zhi, and Tiffany Barnes. 2017a. Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming. In *Proceedings of the International Conference on Educational Data Mining*.

[2] Thomas W. Price, Rui Zhi, and Tiffany Barnes. 2017b. Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior. In *Proceedings of the International Conference on Artificial Intelligence in Education*.

[3] Kelly Rivers and Kenneth R. Koedinger. 2017. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64. http://link.springer.com/10.1007/s40593-015-0070-z