Exploring Design Choices in Data-driven Hints for Python Programming Homework

Thomas W. Price
NC State University
Raleigh, NC, USA
twprice@ncsu.edu

Samiha Marwan
NC State University
Raleigh, NC, USA
samarwan@ncsu.edu

Joseph Jay Williams
University of Toronto
Toronto, CA
williams@cs.toronto.edu

ABSTRACT
Students often struggle during programming homework and may need help getting started or localizing errors. One promising and scalable solution is to provide automated programming hints, generated from prior student data, which suggest how a student can edit their code to get closer to a solution, but little work has explored how to design these hints for large-scale, real-world classroom settings, or evaluated such designs. In this paper, we present CodeChecker, a system which generates hints automatically using student data, and incorporates them into an existing CS1 online homework environment, used by over 1000 students per semester. We present insights from survey and interview data, about student and instructor perceptions of the system. Our results highlight affordances and limitations of automated hints, and suggest how specific design choices may have impacted their effectiveness.

Author Keywords
automated programming hints; computing education.

CCS Concepts
•**Social and professional topics** → **CS1;** •**Human-centered computing** → *Human computer interaction (HCI);*

INTRODUCTION
Students often struggle during programming homework in introductory Computer Science (CS) courses, and may need help getting started, localizing errors, or remembering when and how to use specific code elements. Additionally, in large or online courses, instructors are not always available to provide this help, suggesting the need for *automated* support that is always available and can scale to any class size.

In this paper we explore how to provide students with programming hints that can help them progress and learn during practice assignments. While instructors commonly hand-author hints or explanations that can be shown along with a homework problem, we explore how to present *automated* programming hints. These hints can be data-driven, generated in real time using prior students' data, and tailored to a student's current code [3]. Such hints can help students when they get stuck by suggesting specific edits that students can make to bring their code closer to a correct solution.

In this paper, we explore how to design data-driven programming hints to support an existing online homework environment in a large, introductory CS course. We do this by developing and deploying a prototype data-driven hint system called CodeChecker. Our goal is to understand the benefits and trade-offs of specific choices that we made in creating CodeChecker, in order to inform the design of future systems. To do so, we highlight *design choices* that creators of data-driven programming hints will have to make, and we show how these choices have varied across prior work. For example: Where should hints be displayed – embedded in a student's code or in a separate window? How much detail and how many hints should be included? We then discuss specific choices we made in creating CodeChecker, and justify these choices, drawing on design considerations derived from prior educational literature.

We deployed CodeChecker with 457 students during 2 weeks of Python programming homework. We then collected data from surveys, and student and instructor interviews, which allowed us to investigate perceptions of the specific design choices we made. Our results highlight how to better design data-driven hints for large-scale classroom deployments. They also surface trade-offs in the design of these systems, such as the tension between making help salient and respecting students' desire for independence. This paper's goal is to provide insights that designers can use in turning hint generation algorithms into real-world, student-facing systems. The primary contribution of this work is highlighting design choices for data-driven hint systems, as well as qualitative results from from an empirical classroom study that suggest affordances, limitations and trade-offs in the design of data-driven hint systems.

CodeChecker SYSTEM DESIGN
We created a prototype system called CodeChecker that automatically generates and displays *transformation hints*, suggesting edits that a student can make to their current code to get closer to a correct solution. When a student submits their code for evaluation, the hints are displayed in a panel, which shows a copy of the student's code, annotated with suggestions. These hint *annotations* indicate available hints that a student can investigate further. A red strikethrough indicates

L@S'21, June 22–25, 2021, Potsdam, Germany

ACM ISBN 10.1145/3430895.3460159.
DOI: https://doi.org/10.1145/3430895.3460159

an available deletion hint, a purple strikethrough indicates code that could be replaced, a plus icon indicates where code can be added. Students can hover over any of the annotations to see the full hint as a tooltip, including a natural language explanation and additional details about what code is missing. CodeChecker also summarizes the hints, giving a list of all missing code elements at the end. Unlike feedback from test-cases, which only tell a student if the *output* of the program is correct, these hints show how to move towards correct functionality. A student can resubmit their code at any time to receive updated hints, which adapt to any changes the student has made. CodeChecker uses the SourceCheck algorithm [1] to automatically generate programming hints, but the algorithm does not specify how to *present* these suggestions to the student.

Dimensions of the Design of Transformation Hints

In this section, we consider challenges and opportunities that arise in real-world deployments of data-driven hint systems, and discuss how they shaped the design of the CodeChecker system. We propose 3 relevant dimensions along which data-driven hints can vary. We present the choices we made in designing CodeChecker, and provide our rationale for each. Our goal here is to *explicitly surface* our choices, so that our evaluation can better inform the design of future systems by investigating how students perceived these choices.

How is the hint chosen?

Many hint generation algorithms can identify multiple possible edits a student can make to improve their code at any given time. A system must therefore have a mechanism for selecting which hint to show. Some systems show one "best" hint, as identified by the algorithm; however, technical evaluations suggest that data-driven algorithms often fail to identify which hints are most important and relevant to the student [2]. **Our Choice**: *Let the student browse hints and select one*. Rather than choosing a hint *for the student*, CodeChecker displays annotations in the student's code, marking each available hint, and allows the student to browse these hints and select one to view further, by hovering over it. This approach has been used in other systems to ensure that the most *relevant* hint to the student is always available. Since students may already have preconceptions about where their errors are, this allows them to find the most relevant hint. If a system chooses only one hint to show, there is a risk that this hint will not align with the student's goals, leading students to give up using hints [2]. However, for students who do *not* have a strong idea of where they want to change their code, this large number of hint choices may be overwhelming, and take additional time to parse.

Where to display hints?

Traditionally, programming hint systems have displayed hints in a dedicated panel or dialog box in the user interface (e.g. in ITAP [3]). However, many *programming environments* show compiler errors and suggestions by annotating a user's code (e.g. with a red underline) and showing tooltips. **Our Choice**: *Annotate a student's code with hints, and also summarize hints in a separate area*. Transformation hints suggest an edit to specific token(s) in a student's code. To make these suggestions more *salient*, we represent them by annotating the relevant token in a copy of the student's code. This suggests exactly where an error is, helping to make hints *actionable*. If a student hovers over any of these annotations, a tooltip is shown with a textual explanation of the suggested change (e.g. "You may want to delete this code"). However, as students may not always use tooltips, we also summarize the hints at the bottom of the hint display.

How much detail to include?

Prior work has distinguished between *pointing hints* that simply identify where an error or next-step is located, and *bottom-out* hints that directly tell the student what to do. A transformation hint could be presented as either, or something in between, and this choice entails trade-offs. A pointing hint may not be actionable enough to help a struggling student, but bottom-out hints can be easily abused by students to expediently complete problems without learning. **Our Choice**: *Show exactly where to change the code, and what elements are missing, but not what code to add*. Rather than giving a student an exact edit to make (e.g. "On line 3, add: `for i in range(10)`"), CodeChecker only suggests which code elements are missing (e.g. "You may be missing a `for` loop) and where to add them (via annotations). This gives the student an *actionable* next step, but it does not give away exact code structure, with the goal of *encouraging reasoning*. However, this may also cause difficulties for struggling novices, who may have trouble translating these higher-level suggestions into code.

STUDY DESIGN

To understand the impact of CodeChecker's hints on students' programming experience, we deployed CodeChecker in a classroom setting, and collected survey data on students' experience using the hints, as well as in-depth interviews with six participants and the instructor.

Population

Our study took place in an in-person introductory Computer Science course at a large public university in North America. The class consisted of CS-majors and non-majors with little to no prior programming experience. Our study focused on Python programming homework assignments during weeks 7 and 8 of the course, where students were practicing the challenging topic of lists and dictionaries. The class included 457 students who attempted any week 7 and week 8 problems and who consented to their data being analyzed. These students were randomly assigned to receive hints either in week 7 only ($n = 237$) or in week 8 only ($n = 220$), and to have no hints in the other week. We do not investigate the experimental comparison here, but include survey data from students in both groups, reporting on their experiences with the hints.

Procedure

Homework Intervention & Survey: During each homework assignment, students completed 4 code writing tasks in an online practice environment called PCRS. In each problem, students completed a function stub based on a brief description and examples of correct input/output. Each time students submitted their code, it was checked with 4-7 test cases, and the results were reported to the student. Students could submit

as many attempts to a given problem as they wanted, revising their solution until it passed all test cases. If a problem included hints, CodeChecker would display hints above the test case feedback. Each week included 2 *practice* problems that offered CodeChecker hints, which were the subject of the survey collected (below).

To explore students' perceptions about our design of hints, we administered a survey directly following the homeworks with hints. The survey included 4 open-ended survey questions, asking what was most useful and not useful about hints, what kinds of explanations were helpful, and how hints compared to help from instructors. We analyzed 349 survey responses from students who consented and also reported remembering seeing the hints (students may not have seen hints if they got practice problems correct on their first try).

Follow-up Interviews: We also recruited six students (1 male, 5 female) for follow-up interviews, after the class completed. To encourage students' participation, we offered a $10 gift card as compensation for student time. One researcher conducted one-on-one, semi-structured interviews, which lasted for 50-60 minutes. To ensure hints were salient in the student's mind during the interview, we asked the student to complete two practice programming problems in PCRS, with access to CodeChecker's hints. These problems were from the course, and were more challenging than the week 7 and 8 problems, to increase the need for hints. After each practice exercise, the interviewer asked the student follow-up questions. The students were asked to describe their process of interacting with hints, their perceptions of the hints, including features that were more and less useful, and situations when they would and would not want hints. We also asked students to reflect on their experience with each of CodeChecker's design choices.

Analysis

Thematic Analysis: To analyze interview and open-ended survey data, we adapted techniques from thematic analysis. We started with the interview data, which was more detailed, and included 12 interviews from 6 students (conducted after each of the two programming tasks). After segmenting the data, two researchers worked independently to open-code 4 of these interviews (from 2 students), using inductive analysis. They then discussed the resulting codes, merged them into a smaller group of 35 codes, and defined an initial codebook. The researchers then iteratively re-coded the same 4 interviews, measuring agreement, resolving any conflicts, and clarifying definitions in the codebook after coding each interview. The researchers had complete agreement (across all codes) for 74%, 91%, 91%, and 93% of the segments on the first 4 interviews respectively, with most codes reaching perfect agreement. Having established agreement, a single researcher proceeded to code the remaining 8 interviews, suggesting 2 additional codes to add. The same researcher also randomly sampled 100 non-empty survey responses, each including answers to 4 open-ended questions, and coded these responses using the same codebook. Afterwards, both researchers reviewed all the data, discussed the codes and grouped them into themes. We focus here on themes most relevant to our design.

RESULTS

Here we present results from our analysis of students' interview and survey responses, with the goal of understanding how students' experiences inform our specific design choices.

How is the Hint Chosen?

Students discussed our choice to show them annotations for all available hints and to let them select one:

Too many hint choices can be overwhelming. Students expressed that seeing multiple hints to select from could be "*a little bit disorganized*" [P4][1], and that "*with so many options, you may get lost.... it becomes so overwhelming that I don't actually know what I am doing wrong*" [P2]. This is especially true "*if you have a longer piece of code*" [P5], where the number of hints can get extreme; as one student noted in the survey, the hint annotations "*scratched out my entire code*" [S]. However, when the number of hints was more manageable, students could make use of them: "*I would read them all... most of the time, I would have maybe like two or three hints, and it wouldn't be too extreme*" [P4]. One student noted the advantages of multiple hint options: "*I prefer multiple for sure... it's helpful to see, oh wow, a lot of this is not correct*" [P5]. By contrast, "*if there's only one suggestion I feel like you'll be so caught up on that comment that it takes away from the rest of the problem*" [P5].

Students wanted structure for hint choices. Three interviewees independently suggested organizing multiple hints into a progression: "*So give one hint at a time... because sometimes it's only one part you're confused about and not the whole thing*" [P3]. Students also suggested how to organize this progression, e.g. "*line by line or issue by issue*" [P4], or with increasing detail: "*Less explanation to start with... If still stuck, more explanation should be available*" [S].

Where to display hints?

Students discussed our choice to visually annotate their code with hints, and to show a summary at the end:

Embedding hint annotations in students' code helped direct their attention. Every student we interviewed mentioned that hint annotations helped them localize needed changes, as hints "*point out errors in my code that like I might not have caught myself*" [P1]. This was a priority for some students: "*the hint points where the error is... I think that's the number one thing.*" [P3]. The embedded, visual nature of the hint annotations helped localize these errors: "*what's helpful is ... it's highlighted in a way that it gets pretty apparent... it's a lot more visual*" [P4]. One student noted that this location information may be valuable, even if the suggested action (described in the tooltip) is not: "*here the annotations were not that accurate, but at least it shows you where and what could be wrong.*" [P3]. Some students in the survey also noted that the annotations gave them a chance to think about how to fix the error before seeing the full hint suggestion: "*when it cross[es] out my wrong part... I know that part is wrong, and I have to figure out that by myself.*" [S]

[1]A number with 'P' (e.g. P4) after quotations indicate interview participants. 'S' indicates a survey response.

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