# Seeing Program Output Improves Novice Learning Gains

Juho Leinonen
The University of Auckland
Auckland, New Zealand
juho.leinonen@auckland.ac.nz

Arto Hellas
Aalto University
Espoo, Finland
arto.hellas@aalto.fi

John Edwards
Utah State University
Logan, Utah, United States
john.edwards@usu.edu

## ABSTRACT

In this article, we report results from a randomized controlled trial where novice programmers completed code mimicking exercises – writing and modifying code shown to them – designed to help learn the basics of how variables work. Using a tailored code writing system with feedback on program correctness, we conducted a two-group design study where only one of the groups could see the program output and feedback on the correctness of the program they wrote, while the other group just saw feedback on correctness. Learning gain was measured using a code-reading multiple choice questionnaire as both a pretest and a posttest. Our data suggests that being able to see program output leads to higher learning gains for novices, when compared to just being able to see feedback on the correctness of the code. For more experienced students, we observed benefits from code mimicking in both groups, without a strong distinction between being able to see the output and not being able to see the output. Based on our experiment, we recommend that environments used by novices for learning programming should encourage – or even require – running the code before allowing submitting the program for assessment.

## CCS CONCEPTS

• **Applied computing** → *Interactive learning environments*; • **Social and professional topics** → *Computing education*.

## KEYWORDS

automated assessment, randomized controlled trial, feedback, program output, misconceptions, variables, novice programmer

**ACM Reference Format:**
Juho Leinonen, Arto Hellas, and John Edwards. 2023. Seeing Program Output Improves Novice Learning Gains. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023), July 8–12, 2023, Turku, Finland*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587102.3588796

## 1 INTRODUCTION

Recently, there has been an emergence of studies that have explored the use of simple syntax practice for helping novices in introductory programming courses [9, 10, 14, 22, 26]. In simple syntax practice tasks, students are shown code that they need mimic either by rewriting the exact same code or by rewriting the code and adjusting it in a marginal fashion such as changing variable values. When mimicking code, students may not have been told beforehand about the purpose of the specific syntax they are working with or what the code that they mimic does.

The technical implementation of these studies has often differed to some extent. One could, for example, show code that needs to be written as is, and highlight every character that is mistyped [22]. On the other hand, it could be that students are given code and they are told what the code outputs, and the students are then told to recreate a program that does something similar [9, 10]. While [22] found that the interventions did not improve learning outcomes, the opposite was observed in [9, 14, 26]. In [22], students were expected to simply write code that they saw, while the other studies have often focused on also adjusting shown code. In addition, the environments have differed to some extent between the studies – in [14, 22], the environment did not show the program output, while in [9, 26] the environment did show the output.

In the present work, we explore the possibility of using simple syntax practice for learning about the very basics of programming, looking into the effect of the used environment. Controlling whether students have access to program output and feedback from automated assessment versus just the feedback from automated assessment, we answer the following research questions:

RQ1 Given feedback from automated assessment, what is the effect of seeing program output on the learning of novice programmers?
RQ2 Given feedback from automated assessment, what is the effect of seeing program output on the mental effort of novice programmers?
RQ3 Given feedback from automated assessment, what is the effect of seeing program output on the attitudes toward programming of novice programmers?

This article is organized as follows. In the next section, we discuss background on automated assessment & feedback, syntax practice, programming misconceptions and the research gap this study tackles. In Section 3, we present the experimental setup of the study and our methods. Section 4 briefly presents results from a pilot study that informed the pivotal study, the results of which are discussed in Section 5. We discuss the results in Section 6 and conclude the article by answering our research questions in Section 7.

## 2 BACKGROUND

Many introductory programming courses rely on automated assessment of programming assignments. Automated assessment is used to provide feedback on submission correctness in a number of ways [1, 17, 20, 29]: at the bare minimum, such systems tell whether a program written by a student is correct or not based on, e.g., input-output tests, but the systems can also provide feedback on structure [8] and test coverage [11, 35]. They can also provide functionality for setting limits on the time and instructions of the

*ITiCSE 2023, July 8–12, 2023, Turku, Finland*
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0138-2/23/07.
https://doi.org/10.1145/3587102.3588796

assessed programs [12, 18, 39], and in some cases have a limited number of submissions to curb over-reliance [7, 25]. There are many benefits to using automated assessment, including scaling instruction to a larger student body, allowing teachers to focus on tasks other than manual grading, and providing more opportunities for when and where students can work [1, 17].

Automated assessment feedback can be provided from multiple angles. Some systems only tell the student whether the program was correct or incorrect, omitting details, while other systems may, e.g., provide inputs that the student should try out if something fails in the program as well as inform the user if there are syntax errors [1, 17, 33]. Some systems, when providing information on the output of the program, may provide the output in a separate window such as a tab that may need to be explicitly accessed [36] or in a downloadable file [27].

While automated assessment systems are widely used, they also come with a number of downsides. Automated assessment tools may become a crutch that students rely on instead of trying to figure out what is wrong [19], students may try to game the automated assessment systems [16], and the availability of feedback may lead to a wheelspinning behavior – going full speed ahead without actually progressing [3]. Indeed, there exists a small chorus of researchers who have started to question the infallibility of automated assessment of programming exercises (c.f. [2]).

The misuse of automated assessment systems and the reliance on perhaps poorly construed feedback from them could also lead to – or maintain – misconceptions. Even introductory concepts such as variables and assignment can cause confusion [15, 21, 30, 34, 37][1]. As an example, the statement `a = b` might be understood as copying the contents of `a` to `b` or as "creating a follower" where whatever is assigned to `b` would be reflected on `a` as well. Having the possibility to observe the output of a program and using that possibility could indeed help to understand such a case. So could feedback from an automated assessment system, however. Hence, our question – *what is the effect of running a program and seeing its output versus just being able to check the correctness of a program*?

## 3 EXPERIMENT DESIGN

### 3.1 Environment

For the purposes of the study, we developed an online environment for writing, running, and checking JavaScript code. We used the Ace[2] editor as the starting point for the environment, introduced a new function called `print` to the JavaScript language, and created the functionality needed to show the output of written JavaScript programs to the user. Basic functionality for testing the output and the structure of the program was implemented. For example, if the user is asked to print a variable, a check is made to ensure that the user prints the variable instead of just printing the variable's value.

Two operating modes were created for the environment: "Run code" mode and "Check code" mode. In the "Run code" mode, users have a "Run code" button which, when pressed, evaluates the written code and shows a console with the output (given that the program produces printed output) and any feedback from automated assessment. In the "Check code" mode, users have a "Check code" button which evaluates the written code and shows any feedback from automated assessment but omits the output of the program.

The automated assessment verified that the user is attempting to perform the given task, i.e., mimic given code instead of trying to bypass the task with simple print statements, and that the output has the expected values. Feedback from automated assessment was succinct. For example, if the program written by the student fails to output an expected number, the shown feedback highlights the issue and tells what the expected number was.

In both operating modes, when the entered program is correct, the users see a thumbs-up emoji and the sentence "Your code is correct!". At this point, a "Next" button is enabled, allowing the user to progress to the next task. The environment also provides a safeguard, where users who get stuck for more than 90 seconds are shown a "Skip" button that allows them to move forward. Pasting code to the environment is disabled. If a user pastes in code, the environment opens a dialog and shows the user a message not to do so. When the dialog is closed, the pasted content is removed.

A screenshot of the environment is shown in Figure 1. The screenshot is from the "Run code" mode. In the screenshot, the user has been given the task to mimic a given program and to create a version of the program that prints the numbers four and two. The user has pressed the "Run code" button and sees the program output and the feedback: the program does not work correctly as the first printed value was not as expected. The key difference between the two operating modes is that in the "Check code" mode, users can not run the program and thus can not not see the program output. They can still get feedback by pressing the "Check code" button.

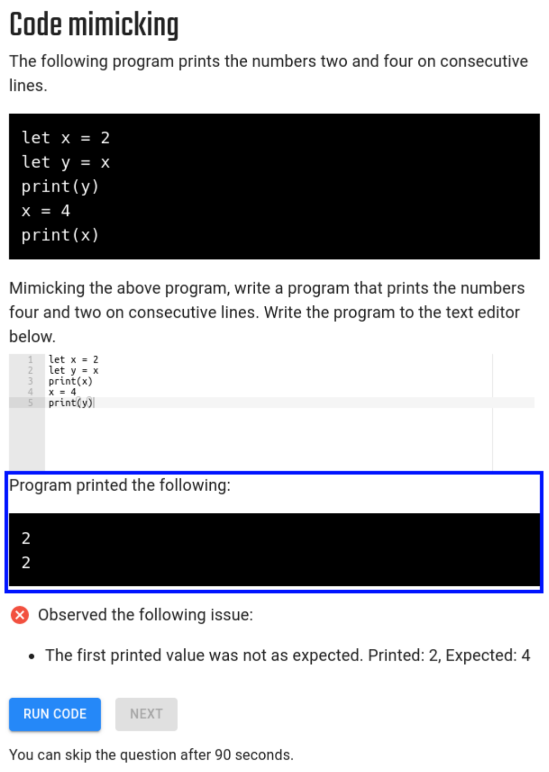
### 3.2 Experiment: Randomized Controlled Trial

The environment was used for an experiment where students were learning about variables, assignment, and printing. The experiment was conducted as a randomized controlled trial where participating students were randomly assigned into two groups, each in one of the two modes of the environment. The experiment consisted of a question gauging previous programming experience, three pretest quiz questions, five code mimicking tasks that students solved within the environment in the mode they were assigned to, a mental effort question [28], three posttest quiz questions, and five questions that gauged the participants' attitudes about the experiment[3]. Other than the environment mode, the contents of the experiment were the same for both groups. The pretest and posttest quizzes contained code-tracing questions that asked the students to determine the output of given programs; the programs contained statements that have been previously found to elicit variable-related misconceptions and were inspired by the literature on misconceptions (briefly mentioned in Section 2). No feedback was given on the pretest and posttest quizzes.

Each code-mimicking task showed an example program to the student which the student then had to write in the code editor. The tasks always included some sort of adjustment where the student had to change what was being printed. As the objective of our study was to determine the impact of being able to run the code and to see

[1]Perhaps also in part due to the learner's prior experience with mathematics [21, 24, 31].
[2]https://ace.c9.io/

[3]Due to space constraints, the full structure of the experiment is outlined in an online appendix at https://osf.io/x8aym/?view_only=9ed93e0320da44e0ba512f10b357f72f



Code mimicking

The following program prints the numbers two and four on consecutive lines.

```
let x = 2
let y = x
print(y)
x = 4
print(x)
```

Mimicking the above program, write a program that prints the numbers four and two on consecutive lines. Write the program to the text editor below.

```
let x = 2
let y = x
print(x)
x = 4
print(y)
```

Program printed the following:

```
2
2
```

Observed the following issue:

- The first printed value was not as expected. Printed: 2, Expected: 4

You can skip the question after 90 seconds.

**Figure 1: A screenshot of the programming environment. The screenshot shows the "Run code" mode, which in addition to the automated assessment feedback shows the program output. The output section in the blue square is omitted for the "Check code" mode.**

the output, and as our intended study population were complete novices, the code mimicking tasks focused on the use of variables.

The question gauging previous programming experience had three options ranging from "No previous programming experience" to "Plenty of previous programming experience" and a separate option for indicating that the participant was not certain. The pretest and posttest multiple-choice questions were graded either correct or incorrect. The mental effort question had 9 items ranging from "Very, very low mental effort" to "Very, very high mental effort" [28]. The questions that gauged participants' attitudes had the statements (1) "I enjoyed solving the preceding code mimicking tasks." (*Enjoyed*); (2) "I think that the experiment was educational." (*Educational*); (3) "I think that the experiment was difficult." (*Difficult*); (4) "I feel that I improved my understanding of how computer programming works." (*How programming works*); and (5) "I feel that I was able to get an idea of how I was doing." (*Idea of how was doing*) and were answered using a 5-item Likert-scale survey from "Strongly agree" to "Strongly disagree".

### 3.3 Analysis

For the present study, we focus on novice programmers as our main interest is in understanding the effects of the two environment modes on those who have not yet learned programming and do not have an intuition of how programs work from prior experience. We define novice programmers as those who both report not having any prior programming experience and who do not score full points in the pretest. Most of our analyses use data from such students, although we briefly discuss the effect of the two modes on more experienced students for completeness in Section 5.5.

For RQ1, to analyze differences in learning between those who were assigned into the "Run code" mode (the *run code* group), and those who were assigned to the "Check code" mode (the *check code* group), we compare the learning gain between students in the two groups. Both the pretest and the posttest had three questions, where we gave a point for each correct answer. Based on performance in the pretest and posttest, we calculate learning gain as *(posttest score - pretest score)*. We then compare the distributions of learning gain between the run and check groups using an appropriate statistical test, determined by whether the data is normally distributed or not.

For RQ2, we examine students' mental effort during the experiment using the Paas mental effort scale [28], while for RQ3, we examine students' attitudes towards the intervention and programming in general. For both RQ2 and RQ3, differences between the groups are studied using Mann-Whitney U tests, as the data is ordinal. When discussing effect sizes for Mann-Whitney U tests, we use Cliff's Delta (CD), which is a non-parametric effect size measure.

When conducting statistical tests, we report $p$-values of the tests as one component among others that together contribute to understanding of the results [41]. We do not make threshold-based claims of statistical significance [5] and do not perform corrections for multiple testing, which may lead to too stringent interpretation of study outcomes [4, 32].

## 4 PILOT STUDY

To assess the feasibility of the experiment, we conducted a pilot study at a mid-sized university in the United States. Participants were drawn from the university's CS1 course. The course instructor (who was not an investigator in this study) sent a message to all students inviting them to participate in the study and visit the learning platform with the environment. Participation was incentivized using a raffle of Amazon vouchers. All student interactions were done in compliance with an IRB-approved protocol.

In total, 29 students participated in the experiment, showing us that the platform and the experiment worked as intended. Out of the 29 students, 8 were novices according to our definition. Of the 8 novices, 4 were in the *run code* condition and 4 were in the *check code* condition. We observed an average learning gain of 1.0 for the *run code* condition and 0.5 for the *check code* condition. This suggested that being able to run the code instead of only being able to check the correctness of the code could potentially be beneficial for learning. The results encouraged us to continue with running a new study with a larger population.

## 5 PIVOTAL STUDY

The pivotal study was conducted in an introductory programming course at a university in Finland. The experiment was linked to the beginning of the course, where participants were asked to help with research. Participation was voluntary, not incentivized, and all interactions were done in compliance with local ethical research

| | Pretest | Posttest | Learning Gain Mean | SD | Median |
|---|---|---|---|---|---|
| *Run code* group | 0.78 | 2.11 | 1.32 | 1.16 | 1.0 |
| *Check code* group | 0.82 | 1.66 | 0.85 | 1.15 | 1.0 |

**Table 1: Pretest and posttest averages as well as learning gain means, standard deviations and medians for the *check code* and *run code* groups in the pivotal study.**

protocols. In total, 274 volunteering learners completed the experiment and of these, 141 learners matched our definition of novice. Of these, 65 were in the *check code* group while 76 were in the *run code* group. All results and discussion in this paper, with the exception of Section 5.5, use only the data from novice participants.

## 5.1 Effects on Learning for Novices

To answer RQ1, we calculated learning gain for each student as outlined in Section 3.3. Table 1 shows pretest and posttest averages as well as the mean, standard deviation and median of the learning gain for both groups. No considerable difference in pretest score is observable between the groups, which is confirmed with a (two-sided) Mann-Whitney U test ($U = 2554.0, p = 0.71, CD = -0.03$).

Table 1 shows that the learning gain mean (1.32 vs 0.85) is higher for the *run code* group, while the medians are the same. Shapiro-Wilk test of normality indicated that the learning gain was not normally distributed ($stat = 0.92, p < 0.001$), and thus we used the (two-sided) Mann-Whitney U test to compare the groups. The test suggests a difference in the learning gains of the two groups ($U = 1964.5, p = 0.03$) with a small effect size ($CD = 0.20$).

## 5.2 Mental Effort

To answer RQ2, we analyzed whether there were differences in how mentally taxing the two different conditions were for the students. Using the Paas scale for mental effort [28], we observed no considerable differences between the mental effort of the two groups (two-sided Mann-Whitney U test, $U = 1162.0, p = 0.36, CD = 0.10$; Mann-Whitney U test was chosen as the data was not normally distributed, Shapiro-Wilk $stat = 0.95, p < 0.001$).

## 5.3 Effects on Attitudes

We proceeded to assess the effect of the experiment on students' attitudes, answering RQ3. To study whether participating in the experiment had different effects on students' attitudes towards programming, students answered five Likert-scale questions related to their attitudes. Figure 2 shows the distribution of students' answers to the five attitude questions separately for the groups. Based on visual analysis of the distributions, there are no considerable differences between the groups.

To confirm this, we ran (two-sided) Mann-Whitney U tests between the groups separately for each question. The results are summarized in Table 2, which also shows the means for each question group[4]. The results suggest that the group might affect perceived

[4]We chose to include means as the medians were the same in both groups, but acknowledge that the data is ordinal which should be kept in mind when interpreting the results.

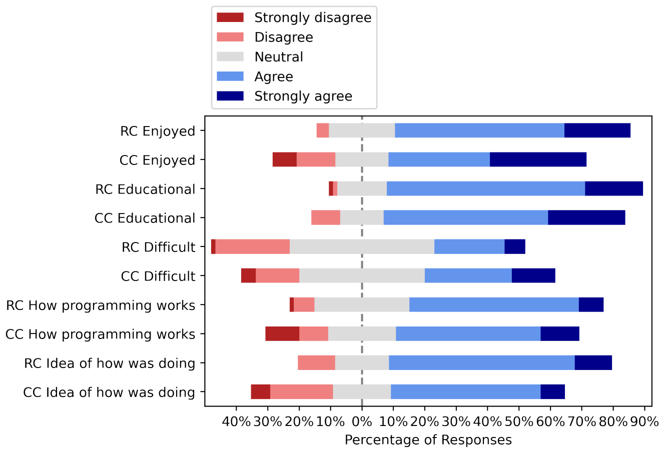


**Figure 2: Distribution of attitude question answers for both the *run code* (RC) and the *check code* group.**

| Aspect | CC | RC | U-val | $p$ | CD |
|---|---|---|---|---|---|
| Enjoyed | 3.66 | 3.92 | 2617.5 | 0.52 | 0.06 |
| Educational | 3.92 | 3.96 | 2463.0 | 0.98 | 0.00 |
| Difficult | 3.32 | 3.09 | 2113.0 | 0.12 | -0.15 |
| How programming works | 3.40 | 3.61 | 2601.5 | 0.56 | 0.05 |
| Idea of how was doing | 3.31 | 3.71 | 2958.5 | 0.03 | 0.20 |

**Table 2: Means (1 = Strongly disagree, 5 = Strongly agree) for the attitude related questions for the *check code* (CC) and *run code* (RC) groups as well as Mann-Whitney U test statistics for comparison of the groups for each individual question.**

difficulty of the task (albeit the effect size $CD = -0.15$ is small) and influence whether one has an idea of how one is doing (albeit the effect size $CD = 0.20$ is again small).

## 5.4 Time-on-Task

As we observed differences in learning gain but no differences in mental effort, we also looked for differences in time-on-task between the groups. Time-on-task was measured as the time that the learners took to complete the experiment. On average, learners in the *check code* group spent 15.6 minutes on the experiment (SD=5.6 minutes), while learners in the *run code* group spent 14.9 minutes on the experiment (SD=5.1 minutes). As time-on-task was not normally distributed (Shapiro-Wilk $stat = 0.96, p < 0.001$), we used (two-sided) Mann-Whitney U test for examining the differences between the groups. No considerable differences were observed ($U = 2681.0, p = 0.38, CD = -0.09$).

## 5.5 Effects on Learning for Those With Some Experience

To gain a more complete picture of the potential benefits of seeing the program output, we studied differences in the groups of participants who reported having at least some prior programming experience. To keep the learning gain analysis meaningful, we focused on the 67 students with prior experience who did not receive full points from the pretest. From these, 30 were in the *run code* group, while 37 were in the *check code* group.

The average learning gain for the *run code* group was 1.13 (pretest score average 1.33), while the average learning gain for the *check code* group was 0.89 (pre-test score average 1.27). Using a two-sided Mann-Whitney U test, we do not observe differences between the groups ($U = 502.5, p = 0.49, CD = -0.09$). The same holds for mental effort ($U = 538.5, p = 0.84, CD = 0.03$), enjoying the experiment ($U = 538.5, p = 0.84, CD = -0.03$), considering the experiment educational ($U = 579.0, p = 0.74, CD = 0.04$), feeling that the experiment helped understand how programming works ($U = 513.5, p = 0.58, CD = -0.07$), forming an idea of own performance ($U = 606.0, p = 0.47, CD = 0.09$), and for time-on-task ($U = 635.0, p = 0.32, CD = -0.14$). Similar to the experiment conducted with novices, the group might have an effect on the perceived difficulty of the task, where the *check code* group might see the experiment as more difficult ($U = 411.5, p = 0.05, CD = -0.259$).

Despite not seeing major differences between the groups, we do observe learning gains from the experiment in both groups, which provides further evidence of the benefits of code mimicking tasks.

## 6 DISCUSSION

### 6.1 Running the Program and Seeing the Output Aids Novices

We found that the novices in the *run code* group had, on average, higher learning gains, in terms of improvement from pretest to posttest, compared to the students in the *check code* group. This observation was present in both the pilot and in the pivotal study. This suggests that being able to run the program and see the output of the program is beneficial for learning of novices. What is interesting is that the feedback from the automated assessment, which was available to both the *run code* and the *check code* group, could be considered as more detailed than the program output by itself as it includes information on both the actual and the expected output. One potential explanation for this is that the act of "checking" code is perceived differently by students than "running" code. For example, maybe running the code encourages students to practice simulating program execution, or perhaps running the code more explicitly maps the outputs to the student-written source code, thus helping the student better develop an understanding of how the program works. Moreover, considering that the experiment focused on variables, our results suggests that even a brief code mimicking intervention (the whole experiment took on average approximately 15 minutes per novice) helps learning about them (and could potentially alleviate variable-related misconceptions).

We assessed whether there were differences between the groups in terms of invested mental effort and time-on-task. Neither of these showed differences between the groups, suggesting that the mental effort needed for the task was similar for both groups, as was the time-on-task. The latter observation in particular is highly relevant, as one common pitfall in educational experiments that lead to better learning outcomes is that the group with better outcomes simply spends more time on the task — prior research has suggested that time-on-task is one of the key contributors to learning [23, 40].

Our results highlight that novices benefit from being able to run the program and seeing the program output. One interpretation of this result is that the program output helps novices rationalize about the execution of the program, by allowing them to match the outputs with the code. One could argue that an output window could be used as a notional machine for novices, when interpreting notional machines as pedagogical devices as discussed in [13]: the output window provides focus and indicates control and flow and other constructs important to understanding how a program works.

Another reason that output may improve outcomes is interest. Instead of simply trying to find a way to pass the automated tests, interacting with output may encourage interest in why the program is behaving the way it is.

### 6.2 Experiment and Attitudes

Five questions were included in the survey that measured attitudes: a question on enjoyment of the activity, one on perceived difficulty, one related to educational value, one to general understanding of how programming works, and one asking if the student felt like they had an idea of how they were doing. None of these questions showed a considerable difference between the groups, although there might be some differences between the groups in difficulty and in having an idea of how one was doing.

Considering difficulty, the experiment seems to have possibly felt slightly more difficult for the *check code* group than for the *run code* group (3.32 vs 3.09 mean for the question "I think that the experiment was difficult"). This supports the idea that being able to run and see the output of programs can potentially make learning easier. We acknowledge however that we did not observe differences in mental effort.

Furthermore, the *run code* group responded somewhat more positively to the question on whether they had an idea of how they were doing. This observation could indicate that being able to run the program and see its output helps keep track of progress. Indeed, intuitively it makes sense that feedback telling what to fix might not give the same feeling of progression and understanding of what is going on compared to seeing the output of the program where incremental advancement towards the correct solution can potentially be observed by repeatedly running the program and observing its output between modifications to the source code.

### 6.3 Code Mimicking and Some Prior Experience

We studied the effect of the code mimicking task on those who were not full novices, i.e. who had at least some prior programming experience but did not receive a full score from the pretest. We omitted students who received full points in the pretest from the analysis. Our results indicate that the code mimicking practice is beneficial for both conditions, as both had positive learning gains.

Interestingly, when compared to the novices, more advanced programmers do not seem to benefit from being able to run the code and to see the program output. This was observed both for the learning gain and for perceived understanding of how they were doing. One possible reason for this is that more experienced students may already have a model of how programs work and thus are able to simulate program execution without the external help of the tool.

### 6.4 Benefits of Writing Practice

This study had students *mimic* code that they were shown and adjust it based on given guidelines. The results provide further evidence on the benefits of writing practice when learning programming, which has been studied in the past [9, 10, 14, 26]. While [22] found that the syntax practice did not help, [9, 14, 26] observed the opposite. Although there are multiple differences in their research methodologies and contexts, there are two interesting and seemingly inconsequential aspects: (1) the environments used in [9, 26] showed program output and the other environments did not, and (2) the environment used in [22] provided immediate feedback (already during typing) while the other environments did not.

In the present study, we looked into the effect of being able to see the output or not, and suggest that the effect of immediate versus on-demand feedback could be explored in the future. Although we observed that novices in both conditions showed learning gains, our results indicate an added benefit in being able to see the output. These results can also shed some light on earlier studies that have found that using small programming assignments can be beneficial [6, 9, 38]. What if, in addition to other observed benefits such as students starting their work earlier [6], these small assignments increase the tendency to run the programs and consequently observe the output? If this is the case, our results strengthen the case for using small programming assignments for novices.

### 6.5 Limitations

Our study has limitations, which we acknowledge here. Firstly, students knew that they were participating in an experiment, that data from their learning was collected, and that their behavior was observed. This might influence their behavior, which means that we do not know whether the results generalize to an actual classroom.

Secondly, participation was voluntary, and students in the pivotal study were not incentivized in any way. There is a possibility of selection bias, which again highlights the need for further studies to assess generalizability of the results.

Thirdly, the present study focused on novice programmers and elementary programming concepts (variables, assignment, printing), which means that we do not know whether the studied code mimicking approach would generalize to more difficult concepts. Our results are in line with some prior studies where students have participated in syntax practice [9, 14, 26], which have reported benefits of such practice, and in which students have also been given more complex topics to write. It is possible that the code mimicking approach would work with more challenging concepts as well, although this should be explored in future studies.

Finally, the experiment was relatively short. While we observed larger positive learning gains in the *run code* group compared to the *check code* group, we do not know whether the effects of the intervention persist. At the same time, for limited interventions targeting specific learning objectives related to e.g. variables, as was the case in our work, it is possible that even a short intervention with positive results could lead to longer term benefits, as the use of variables continues after they have been introduced. We also note that there are examples of other experiments for teaching variables that are also relatively short (see e.g. [42]), highlighting that these topics can be taught in a relatively short time. In our future work, we are interested in replicating our study over a longer period of time, where we look also into the persistence of the effects.

## 7 CONCLUSION

In this study, we investigated a potential shortcoming of automated assessment systems that provide feedback on program correctness but do not require that students run their code. We explored to what extent seeing the output of a program influences learning – positive influences on learning would make sense when one subscribes to the idea that the output of a program is feedback.

To summarize, our research questions and their answers are as follows. For RQ1, *Given feedback from automated assessment, what is the effect of seeing program output on the learning of novice programmers?*, we found that novices who saw program output had better learning gains than novices who just saw feedback from automated assessment. We hypothesize that students gain a better understanding of flow and control, state, and causality when they can observe the internal behavior of the program through output. For those with some prior programming experience, the difference in learning gain between the groups was not noticeable.

For RQ2, *Given feedback from automated assessment, what is the effect of seeing program output on the mental effort of novice programmers?*, we found no considerable differences in terms of mental effort or time-on-task between the groups. The same holds also for those with some prior experience.

For RQ3, *Given feedback from automated assessment, what is the effect of seeing program output on the attitudes toward programming of novice programmers?*, we found no evidence of program output affecting enjoyment, the perceived educational value of the experiment, or on forming an understanding of how programming works. We did observe minor differences between the groups in terms of the perceived difficulty of the task and on forming an idea of how they were doing. In both cases, being able to run the program and see its output had a slight positive effect, i.e. decreased perceived difficulty and increased understanding of how the student was doing. For those with some prior experience, only perceived difficulty had any differences between the *check code* and *run code* groups.

To summarize, it appears that programming exercises – especially ones focused on code mimicking – could be less beneficial for novices when program output is not shown. This is an actionable finding. It highlights the need to emphasize – or even require – running programs and observing their outputs before sending programs for assessment. Future work could include replication of our results with more challenging concepts and in new contexts; exploration of the effect of on-demand versus immediate feedback; exploration of the cognitive, theoretical underpinnings of the phenomenon; and further investigation of the effects of automated testing on computing education.

## ACKNOWLEDGMENTS

We are grateful for the grant from the Ulla Tuominen Foundation to the first author. We would also like to thank Chad Mano for including the pilot study in his course.

## REFERENCES

[1] Kirsti M Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Computer science education* 15, 2 (2005), 83–102.

[2] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. 2021. STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance. In *Proc. of the 52nd ACM Technical Symp. on Computer Science Education*. 1062–1068.

[3] Joseph E Beck and Yue Gong. 2013. Wheel-spinning: Students who fail to master a skill. In *Int. conf. on artificial intelligence in education*. Springer, 431–440.

[4] Marc Buyse, SA Hurvitz, F Andre, Z Jiang, HA Burris, M Toi, W Eiermann, M-A Lindsay, and D Slamon. 2016. Statistical controversies in clinical research: statistical significance – too much of a good thing... *Annals of Oncology* 27, 5 (2016), 760–762.

[5] Ronald P Carver. 1993. The case against statistical significance testing, revisited. *The J. of Experimental Education* 61, 4 (1993), 287–292.

[6] Paul Denny, Andrew Luxton-Reilly, Michelle Craig, and Andrew Petersen. 2018. Improving complex task performance using a sequence of simple practice tasks. In *Proc. of the 23rd Annual ACM Conf. on Innovation and Technology in Computer Science Education*. 4–9.

[7] Paul Denny, Jacqueline Whalley, and Juho Leinonen. 2021. Promoting Early Engagement with Programming Assignments Using Scheduled Automated Feedback. In *Australasian Computing Education Conference*. 88–95.

[8] Anton Dil and Joseph Osunde. 2018. Evaluation of a tool for Java structural specification checking. In *Proc. of the 10th Int. Conf. on Education Technology and Computers*. 99–104.

[9] John Edwards, Joseph Ditton, Dragan Trninic, Hillary Swanson, Shelsey Sullivan, and Chad Mano. 2020. Syntax exercises in CS1. In *Proc. of the 2020 ACM Conf. on Int. Computing Education Research*. 216–226.

[10] John M Edwards, Erika K Fulton, Jonathan D Holmes, Joseph L Valentin, David V Beard, and Kevin R Parker. 2018. Separation of syntax and problem solving in Introductory Computer Programming. In *2018 IEEE Frontiers in Education Conf. (FIE)*. IEEE, 1–5.

[11] Stephen H Edwards and Manuel A Perez-Quinones. 2008. Web-CAT: automatically grading programming assignments. In *Proc. of the 13th annual Conf. on Innovation and technology in computer science education*. 328–328.

[12] Emma Enström, Gunnar Kreitz, Fredrik Niemelä, Pehr Söderman, and Viggo Kann. 2011. Five years with kattis—using an automated assessment system in teaching. In *2011 Frontiers in Education Conf. (FIE)*. IEEE, T3J–1.

[13] Sally Fincher, Johan Jeuring, Craig S Miller, Peter Donaldson, Benedict Du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice Pearce, and Andrew Petersen. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proc. of the Working Group Reports on Innovation and Technology in Computer Science Education*. 21–50.

[14] Adam M Gaweda, Collin F Lynch, Nathan Seamon, Gabriel Silva de Oliveira, and Alay Deliwa. 2020. Typing exercises as interactive worked examples for deliberate practice in cs courses. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. 105–113.

[15] John D Gould. 1975. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies* 7, 2 (1975), 151–182.

[16] Derrick Higgins and Michael Heilman. 2014. Managing what we can measure: Quantifying the susceptibility of automated scoring systems to gaming behavior. *Educational Measurement: Issues and Practice* 33, 3 (2014), 36–46.

[17] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proc. of the 10th Koli calling international Conf. on computing education research*. 86–93.

[18] David Jackson and Michelle Usher. 1997. Grading student programs using ASSYST. In *Proc. of the twenty-eighth SIGCSE technical Symp. on Computer science education*. 335–339.

[19] Ville Karavirta, Ari Korhonen, and Lauri Malmi. 2006. On the use of resubmissions in automatic assessment systems. *Comp. science education* 16, 3 (2006), 229–240.

[20] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)* 19, 1 (2018), 1–43.

[21] Tobias Kohn. 2017. Variable evaluation: An exploration of novice programmers' understanding and common misconceptions. In *Proc. of the 2017 ACM SIGCSE Technical Symp. on Computer Science Education*. 345–350.

[22] Antti Leinonen, Henrik Nygren, Nea Pirttinen, Arto Hellas, and Juho Leinonen. 2019. Exploring the applicability of simple syntax writing practice for learning programming. In *Proc. of the 50th ACM Technical Symp. on Computer Science Education*. 84–90.

[23] Juho Leinonen, Francisco Enrique Vicente Castro, and Arto Hellas. 2022. Time-on-Task Metrics for Predicting Performance. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. 871–877.

[24] Juho Leinonen, Paul Denny, and Jacqueline Whalley. 2021. Exploring the Effects of Contextualized Problem Descriptions on Problem Solving. In *Australasian Computing Education Conference*. 30–39.

[25] Juho Leinonen, Paul Denny, and Jacqueline Whalley. 2022. A Comparison of Immediate and Scheduled Feedback in Introductory Programming Projects. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. 885–891.

[26] Anna Ly, John Edwards, Michael Liut, and Andrew Petersen. 2021. Revisiting Syntax Exercises in CS1. In *Proceedings of the 22st Annual Conference on Information Technology Education*. 9–14.

[27] Sin Chun Ng, Steven O Choy, Reggie Kwan, and SF Chan. 2005. A web-based environment to improve teaching and learning of computer programming in distance education. In *Int. Conf. on Web-based Learning*. Springer, 279–290.

[28] Fred GWC Paas. 1992. Training strategies for attaining transfer of problem-solving skill in statistics: a cognitive-load approach. *J. of educational psychology* 84, 4 (1992), 429.

[29] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)* 22, 3 (2022), 1–40.

[30] Ralph T Putnam, Derek Sleeman, Juliet A Baxter, and Laiani K Kuspa. 1986. A summary of misconceptions of high school Basic programmers. *J. of Educational Computing Research* 2, 4 (1986), 459–472.

[31] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–24.

[32] Priya Ranganathan, CS Pramesh, and Marc Buyse. 2016. Common pitfalls in statistical analysis: the perils of multiple testing. *Perspectives in clinical research* 7, 2 (2016), 106.

[33] Graham HB Roberts and Janet LM Verbyla. 2003. An online programming assessment tool. In *Proc. of the fifth Australasian Conf. on Computing education-Volume 20*. Citeseer, 69–75.

[34] Teemu Sirkiä and Juha Sorva. 2012. Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises. In *Proc. of the 12th Koli Calling Int. Conf. on Computing Education Research*. 19–28.

[35] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K Hollingsworth, and Nelson Padua-Perez. 2006. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *ACM Sigcse Bulletin* 38, 3 (2006), 13–17.

[36] Thomas Staubitz, Hauke Klement, Ralf Teusner, Jan Renz, and Christoph Meinel. 2016. CodeOcean-A versatile platform for practical programming excercises in online environments. In *2016 IEEE Global Engineering Education Conf.* 314–323.

[37] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming misconceptions for school students. In *Proc. of the 2018 ACM Conf. on Int. Computing Education Research*. 151–159.

[38] Arto Vihavainen, Matti Paksula, and Matti Luukkainen. 2011. Extreme apprenticeship method in teaching programming for beginners. In *Proc. of the 42nd ACM technical Symp. on Computer science education*. 93–98.

[39] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. 2013. Scaffolding students' learning using test my code. In *Proc. of the 18th ACM Conf. on Innovation and technology in computer science education*. 117–122.

[40] Herbert J Walberg. 1988. Synthesis of research on time and learning. *Educational leadership* 45, 6 (1988), 76–85.

[41] Ronald L Wasserstein and Nicole A Lazar. 2016. The ASA statement on p-values: context, process, and purpose. *The American Statistician* 70, 2 (2016), 129–133.

[42] Albina Zavgorodniaia, Arto Hellas, Otto Seppälä, and Juha Sorva. 2020. Should explanations of program code use audio, text, or both? A replication study. In *Koli Calling'20: Proc. of the 20th Koli Calling Int. Conf. on Computing Education Research*. 1–10.