Investigating the Impact of Using a Live Programming Environment in a CS1 Course

FirstName1 LastName1
firstlast1@affiliation.com
Affiliation

FirstName2 LastName2
firstlast2@affiliation.com
Affiliation

FirstName3 LastName3
firstlast3@affiliation.com
Affiliation

FirstName4 LastName4
firstlast4@affiliation.com
Affiliation

FirstName5 LastName5
firstlast5@affiliation.com
Affiliation

ABSTRACT

Novice programmers often struggle with code understanding and debugging. Live Programming environments visualize the runtime values of a program each time it is modified to provide immediate feedback, which help with tracing the program execution. This paper presents the use of a Live Programming tool in a CS1 course to better understand the impact of Live Programming on novices' learning metrics and their perceptions of the tool. We conducted a within-subjects study at a large public university in a CS1 course in Python (N=237) where students completed tasks in a lab setting, in some cases with a Live Programming environment, and in some cases without. Through post-lab surveys and open-ended feedback, we measured how well students understood the material and how students perceived the programming environment. To understand the impact of Live Programming, we compared the collected data for students who used Live Programming with the data for students who did not. We found that while learning outcomes were the same regardless of whether Live Programming was used or not, students who used the Live Programming tool completed some code tracing tasks faster. Furthermore, students liked the Live Programming environment more, and rated it as more helpful for their learning.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; • **Human-centered computing** → *Graphical user interfaces.*

KEYWORDS

Live Programming, programming environment, experimental study

ACM Reference Format:
FirstName1 LastName1, FirstName2 LastName2, FirstName3 LastName3, FirstName4 LastName4, and FirstName5 LastName5. 2022. Investigating the Impact of Using a Live Programming Environment in a CS1 Course. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education (SIGCSE '22), March 2–5, 2022, Providence, RI*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SIGCSE '22, March 2–5, 2022, Providence, RI

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Novice programmers struggle with various aspects of programming, among which code tracing and debugging are some of the most cited challenges [18, 21, 25, 38]. These challenges have been linked to novices' inaccurate mental models of code [7, 12, 20, 31]. As such, tools and techniques that address such challenges among novices can be invaluable in improving Computer Science (CS) education.

Live Programming [35, 36] is a technique where the development environment provides immediate feedback about program state, visualizing runtime values as the code is modified. Such an environment is low-threshold [24] and assists with code understanding and debugging without the need for advanced environments [15, 37]. Its visualization of code execution further helps programmers trace the code and build more accurate mental models [36]. As such, it is a promising technique that could help novices learn to program.

Meanwhile, Live Programming poses an unavoidable risk of information overload [17] as it constantly displays such information at each and every change of the program, which may instead make programming more overwhelming. Furthermore, interpreting the Live Programming visualization could be "one more thing" to learn for novice programmers and thus offset any benefits it provides.

There has been a fruitful line of research on exploring Live Programming for CS education. Various techniques, programming environments and visualizations have also been developed and tested aiming to assist novices with programming [10, 13, 32]. However, to our knowledge, no large-scale experimental studies in a natural educational setting have been conducted to study the effects of Live Programming environments on novice programmers.

To fill this hole in the research literature, we present the evaluation of the Live Programming tool Projection Boxes [17] in a large CS1 course at a U.S. university. We conducted this experiment across four 50-minute lab programming sessions to assess the effects of Live Programming on students compared to an environment without Live Programming. We used students' post-lab test scores and lab task completion times as learning metrics, and used post-lab surveys to acquire quantitative and qualitative data on students' perceptions of the difficulty and helpfulness of the tool and their preferences. We aim to answer the following questions:

> **RQ1**: How does using Projection Boxes in a lab affect learning metrics, such as test scores and completion times?
> **RQ2**: What are students' perceptions of Projection Boxes?

Our main contribution to computing education research is a large-scale experimental study on investigating the impact of using a Live Programming environment in a CS1 course.

2 RELATED WORK

There is a long line of work on Live Programming, dating back to the seminal work of Hancock [10] that introduces many of the important concepts in Live Programming. There have been Live Programming environments for general-purpose languages [3, 15, 17, 26], as well as specific domains [4, 5, 39]. Various empirical studies, most small lab studies, have shown that Live Programming is positively perceived by programmers [5, 17, 39]. There have also been several studies that look specifically at using Live Programming with novices for educational purposes [9–11, 13, 33]. Our work distinguishes itself by providing a large-scale study (N=237) in a naturally occurring educational environment with novices.

The most closely related work is a study of the UUhistle tool [33] and its impact on students' learning for predicting program output. Sorva et al. [30, 32] showed that UUhistle improved student performance on only one out of four tasks, and students considered the tool easy to use and helpful for learning output prediction. The study had 172 students, recruited from a CS1 class, and the study consisted of one session, which was run *outside* of the classroom setting. Their work also states that the tasks provided to the students were program reading tasks, and were sometimes beyond the student's expected CS1 ability to solve. In contrast: (1) our study was conducted in a *natural educational setting* with tasks related to an existing CS1 curriculum – and thus not intended to go beyond expectations for a CS1 curriculum; (2) our labs included code writing tasks; and (3) our study consisted of multiple sessions.

Besides UUhistle, there have been several other Live Programming tools designed for novices [32], such as Jype [11], Online Python Tutor [9], and Omnicode [13]. An exploratory study on Omnicode [13] with 10 post-secondary students found the tool conducive for debugging and forming mental models of Python code. In a technical report, Alvarado et al. [2] analyzed usage log data of Codelens, a version of Online Python Tutor [9] embedded in a digital textbook [22], from a CS1 course (N=61) and demonstrated that students with more interactions with Online Python Tutor scored higher on midterm exams (though we found no peer-reviewed papers on the work). Both studies show the positive educational impact of Live Programming tools and indicate the need for a large-scale experimental study to better understand the impact over a CS1 course. We believe our work is a step in this direction.

There have also been explorations of Live Programming in education, but not for college-level CS1 courses. For example, Wilcox et al. [37] found through a senior-level operating systems course that students considered Live Programming helpful for debugging. From a small-scale exploratory study, Cabrera et al. [4] noted Live Programming improves children's interactions with physical computing devices and makes learning more engaging.

In addition to *Live Programming*, which refers to the idea of providing immediate feedback to the programmer, there is a similarly-named but different concept of *Live Coding*. Live Coding refers to the presentation during class of the explicit step-by-step thought process of writing code [29]. Live Coding has been used in CS1 education [27, 28] to help novices understand the notional machine [7] and build correct mental models of computer programs [12], thus addressing difficulties with code tracing and debugging [21]. Past studies have examined the impact of Live Coding on student learning. For example, Soosai Raj et al. [28] found that Live Coding reduces the extraneous cognitive load imposed on students in learning. Nevertheless, Live Programming, a visualization paradigm for programming rather than an instructional technique, differs from Live Coding, and the impact of Live Programming on student learning remains understudied. Our work aims to fill this gap by further exploring how Live Programming might address common struggles faced by novices.

Figure 1: In-browser online editor with Projection Boxes.

3 METHODOLOGY

3.1 Course Context and Participants

We conducted our study at a large research-intensive public university in the United States in a large intro to programming (CS1) course in Python, which has a mandatory weekly 50-minute lab. Our study was run in the Fall 2020 offering of this course, which had an initial enrollment of 610. Due to COVID-19 restrictions, the course was fully remote, and the weekly lab was run online over Zoom. Students were split into groups of 4 to 7, and each group was given a Zoom meeting with a tutor at a particular time. There were a total of 12 times given throughout the lab day to accommodate students across different time zones. Students would work on the lab tasks during this live Zoom session, and could ask questions of the tutor or interact with other students. We further allowed students who could not attend any of the assigned lab times to do the lab tasks on their own time.

We excluded students who could not attend all of their assigned live Zoom sessions, skipped any lab, missed any post-lab survey, dropped the class at any point, were less than 18 years of age at the start of the study, or opted out of the study as per our approved IRB protocol. We ended up with 237 participants (99 women, 137 men, 1 unknown), including 51 from underrepresented ethnicities[1], with 86 having no prior programming experience.

3.2 Experimental Design

We ran the study across 4 different labs: a lab on `for` loops (Lab F), a lab on `while` loops (Lab W), a lab on testing code (Lab T), and a lab on Python dictionaries (Lab D). We had two programming environments: an online in-browser editor with Projection Boxes (which we refer to as `PB`) and the exact same online editor with No Projection Boxes (which we refer to as `No-PB`). Fig. 1 shows the view of `PB`, where runtime values can be examined inside the boxes. Students using `No-PB` examine their code by clicking a "Run"

[1]The demographic information was obtained as anonymized aggregates from the university, and more fine-grained data was not available.

button at the upper right corner of the interface to see an output box showing standard output and/or error, which disappears as they make changes to the program.

For each lab we also generated two partitioning of students. We first split all the students into two groups to get Group I (N=111) and Group II (N=126) in Labs F, W and T. To get these two groups, we randomly assigned each lab time to either Group I or Group II. We generated the groups in this way so that all the students doing the lab at the same time would use the same environment. As we switched to a different experimental design in Lab D (more details below), we reassigned all the students into two new groups, independently of the first splitting, to generate Group III (N=119) and Group IV (N=118). Since all students had used Projection Boxes by this point, we randomly assigned each Zoom session (instead of whole lab times) to be either in Group III or Group IV.

In Lab F and Lab W, we used a *nonrandomized control group* 2×2 *crossover design* [8], where Group I used `PB` and Group II used `No-PB` in Lab F and the opposite in Lab W. This is a *nonrandomized* trial because the randomization was only done at the level of lab times, not for each individual student. This design for Lab F and Lab W allowed us to measure post-study test scores in all possible combinations of using vs. not using Projection Boxes.

Noting that this design could result in differences in post-test scores caused by pre-existing between-group differences rather than the intervention [6], in Lab T we conducted an *A/A test* [14], where all participants used the `No-PB` environment. We measured their post-lab test scores to detect any between-group differences.

Finally, recognizing that post-tests only measure knowledge after the lab, as opposed to learning, in Lab D, we adopted a *nonrandomized control group pretest-posttest design* [6] where Group III used `PB` and Group IV used `No-PB`. We used a pre-test and a post-test to measure the amount of learning participants did in the lab.

3.3 Experimental Procedure

Environment Setup (all labs). At the beginning of each lab, the tutors distributed to students a link to a "starter" document containing instructions for accessing the in-browser online programming environment and troubleshooting steps. The document provides a URL to the environment. When students clicked on this URL, they would be asked to login using their institutional emails, and in the back-end we would route each student to the right environment based on the Lab and the Group. If, due to technical or other difficulties, students still could not access the environment 20 minutes into the lab, they were asked to use Python IDLE [1] and were hence excluded from the study.

Tutorials (Labs F and W only). The tutorials were videos with lengths between 5 and 7 minutes, linked from the online programming environment. Participants had to watch the tutorials before working on the lab tasks, but could rewatch them throughout the lab. For Labs F and W we developed two tutorial videos, one for `PB` and one for `No-PB`, on using the environment to work with programming concepts covered in that lab.

Pre-test (Lab D only). In Lab D, a pre-test (linked from the online environment) was conducted to determine the participants' understanding of dictionaries (`dict`) prior to the lab. There were 5 multiple-choice questions (1 point each) on code understanding covering 5 concepts: `list`-to-`dict` conversion, `dict`-to-`list` conversion, `dict` with non-primitive keys, key existence checking, and entry insertion/update. The chosen concepts were covered in lectures prior to Lab D but not in any hands-on labs. Participants could not work on the lab tasks until after they completed the pre-test.

Programming tasks (all labs). Following the tutorials or the pre-test, participants were asked to perform several programming tasks on the topic of each lab. In all labs but Lab T, participants were given 30 minutes; they were given 45 minutes in Lab T as the only non-task component of the lab was the post-test. The task descriptions were provided as an online document linked from the programming environment. For each lab, we created two separate versions of this document for students using `PB` and `No-PB`, which were identical, except for the attached tutorial videos and small variations in the task instructions to account for the environment (e.g., "examine the value of variable `x` using *Projection Boxes*" vs. "... using *a print statement*"). The tasks were developed jointly with the course instructors and teaching assistants (TAs) to align with the learning objectives of the course. They largely mirrored the lab tasks that were used in prior instances of the course.

Post-test (all labs). All participants answered post-lab questionnaires after they completed all tasks for the lab or reached the time limit. The first part of the questionnaire was a post-test consisted of multiple-choice code understanding questions (1 point each) on the topic of the lab. The questions were also developed jointly with the course instructors and TAs. We did not time the participants on the completion of the post-test. There were 6, 4 and 4 questions in the post-tests for Labs F, W, and T, respectively. There were 5 questions for Lab D covering the same concepts tested in the pre-test, but with the questions reordered and each question modified from the corresponding question in the pre-test.

Experience survey (Labs F, W and D only). The second part of the post-lab questionnaire was an experience survey, where students were asked to rate the helpfulness of the programming environment for understanding lab materials and difficulty of using the environment. In Labs W and D, we also asked students to compare `PB` to `No-PB`. We asked which one they preferred, and which one they found more helpful. Each rating used a 5-point Likert scale. Finally, we collected open-ended feedback on the tool.

Data Collection. For all 237 participants, we collected demographic information, lab task durations, and all the test and survey data. A complete collection of the tasks, test questions and surveys can be found at this link: http://bit.do/live_programming_cs1.

4 RESULTS

4.1 Learning Metrics

For all of Labs F, W, T and D, we used pre- and post-test scores and task durations as our metrics to measure student learning.

Pre-Test/Post-Test. We conducted Welch's Two Sample t-test (all labs but Lab T had unequal sample variances) for each test in each lab. In all cases we used $p < .05$ as a threshold for statistical significance. We checked the following: (1) the data is continuous; (2) both groups are random samples chosen from respective populations and are independent of each other. Although none of the labs had

Table 1: Post-test scores for Labs F (on `For` Loops), W (on `While` Loops) and T (on Testing Code). N = group size, M = mean, SD = standard deviation.

Lab	Group	Env.	N	M	SD
F	I	PB	111	5.48 (out of 6)	0.77
	II	No-PB	126	5.51 (out of 6)	0.79
W	I	No-PB	111	3.14 (out of 4)	0.94
	II	PB	126	3.13 (out of 4)	0.99
T	I	No-PB	111	2.07 (out of 4)	1.34
	II	No-PB	126	2.03 (out of 4)	1.34

Table 2: Pre-test, post-test, and gain scores (the difference between post-test and pre-test) for Lab D (on Dictionaries).

			Pre-Test		Post-Test		Gain	
Group	Env.	N	M	SD	M	SD	M	SD
III	PB	119	3.5	1.3	4.0	1.1	0.5	1.0
IV	No-PB	118	3.6	1.3	4.0	1.1	0.3	1.2

normally distributed pre-test/post-test scores, our samples are large enough that assumptions of normality were not required [19].

First, we compared the post-test scores of students using PB vs. No-PB in Labs F and Lab W (Tab. 1). We found *no significant difference* in post-test scores between the two groups, both in Lab F ($t = -.30, df = 232.25, p = .77, d = .04$, 95% CI = [-.23, .17]) and in Lab W ($t = -.07, df = 233.6, p = .94, d = .01$, 95% CI = [-.26, .24])[2].

We then compared the post-test scores in Lab T where both Group I and Group II used No-PB to understand if there are any pre-existing between-group differences (Tab. 1). We found *no significant difference* in post-test scores between the two groups ($t = .23, df = 231.19, p = 0.82, d = .03$, 95% CI = [-.30, .38]) even when they used the same programming environment in the lab.

Finally, for Lab D, we compared pre-test/post-test differences for students using PB vs. No-PB to measure learning gain. We reassigned students to Groups III (using PB) and IV (using No-PB) to equalize groups on existing characteristics prior to Lab D (detailed in Sec. 3.2). We found *no significant difference* in pre-test scores ($t = -.74, df = 234.81, p = .46, d = .10$, 95% CI = [-.45, .20]), post-test scores ($t = .06, df = 234.93, p = .95, d = .01$, 95% CI = [-.28, .30]), and gain scores ($t = .91, df = 228.05, p = .36, d = .12$, 95% CI = [-.15, .42]) between the two groups (Tab. 2). That is, we saw no significant difference in either knowledge of dictionaries prior to the lab or knowledge gain about dictionaries between Groups III and IV.

Combining the results from the above, we conclude:

> ***Takeaway 1**: Between students using PB vs. No-PB, we detected no pre-existing between-group differences, no between-group differences in the ability to understand the lab, and no difference in the knowledge gain on Python dictionaries.*

Task Durations. We used Welch's Two Sample t-test to compare the task durations of students using PB vs. No-PB in Labs F, W and D. A Q-Q plot showed that the distributions are not normal, but we considered the test to be applicable for our large sample size [19].

Out of 19 tasks and subtasks (6 for Lab F, 8 for Lab W, and 5 for Lab D), we found 5 tasks to have statistically significant difference ($p < .05$), and in all these cases students using PB took less time than with No-PB: F-4 ($t = -4.52, p < .001, d = -.70$, 95% CI = [-1.02, -.37]), W-1 ($t = -4.82, p < .001, d = -.65$, 95% CI = [-.91, -.39]), W-2.1.1 ($t = -2.11, p = .04, d = -.29$, 95% CI = [-.55, -.02]), W-3.1 ($t = -2.04, p = .04, d = -.28$, 95% CI = [-.54, -.02]), and W-4.1 ($t = -3.15, p < .01, d = -.50$, 95% CI = [-.79, -.20]). Tab. 3 shows the results. Since not all students started all tasks, N differs for each task and can be less than the group sizes. Note that both tasks F-4 and W-1 ask students to trace multiple variables inside a loop, and because doing so using PB is intuitively faster than using No-PB, we observe large time difference and high statistical significance between PB and No-PB in these two tasks.

[2] t - t-statistic; p - p-value; d - Cohen's d effect size; CI - Confidence interval.

Table 3: Statistically significant task durations. ** indicates $p < .001$. N = number of students who started the task.

Lab	Task	Group	Env.	N	M (min.)	SD (min.)
F	**4****	**I**	**PB**	**71**	**2.33**	**1.33**
		II	**No-PB**	**85**	**3.65**	**2.27**
W	**1****	**I**	**No-PB**	**110**	**5.91**	**3.27**
		II	**PB**	**126**	**4.19**	**1.96**
	2.1.1	I	No-PB	106	1.91	1.53
		II	PB	123	1.52	1.17
	3.1	I	No-PB	104	3.17	2.70
		II	PB	124	2.55	1.73
	4.1	I	No-PB	80	1.23	1.05
		II	PB	104	0.81	0.66

Based on our analysis of task duration times, we conclude:

> ***Takeaway 2**: Students using Projection Boxes completed some code tracing tasks more quickly.*

4.2 Students' Perceptions

After each of Labs F, W and D, students rated the helpfulness and difficulty of using the environment (the **Environment Rating**). After Labs W and D, all participants (having used both PB and No-PB) were asked which one they found more helpful and which one they preferred (the **Preference Rating**). For each rating in each lab, we performed a two-sided Mann-Whitney U test to identify statistical significance. We converted each Likert scale rating to a value between 1 and 5 and checked the following assumptions: (1) the data is on an ordinal scale; (2) there is one independent variable consisting of two categorical variables (group); (3) the groups are independent; (4) the data have similar shapes.

Environment Rating. Tab. 4 summarizes the results from the two-sided Mann-Whitney U test on the helpfulness rating (Lab F: $U = 8745, p < .001, r = .24$; Lab W: $U = 8374.5, p = .004, r = .19$; Lab D: $U = 8554.5, p = .002, r = .20$)[3]. We can see that the mean helpfulness rating for PB is higher than No-PB in all labs, and the difference is *statistically significant* ($p < 0.05$ for all labs). For difficulty of use of the environment (results also in Tab. 4), we see that in the first two labs, Labs F and W, the Mann-Whitney U test reveals *no statistical significance* (Lab F: $U = 6430.5, p = .25, r = .075$; Lab W: $U = 6994.5, p = .998, r < .001$; all with $p < .05$). However, we see that later in the academic term, in Lab D, there is a *statistically significant difference* for the difficulty rating for Lab D ($U = 8046.5, p = .04, r = .14$), where students found the environment with Projection Boxes easier to use.

The findings above lead us to the following takeaway:

[3] U - Mann-Whitney U statistic; r - Rank-biserial correlation effect size.

Table 4: Helpfulness (1 = not at all helpful, 5 = very helpful) and difficulty (1 = very difficult to use, 5 = very easy to use) of using the environment provided in each lab. All medians are 4. * = $p < .05$ and ** = $p < .001$.

Item	Lab	Group	Env.	N	M	SD
Helpfulness	**F** **	**I**	**PB**	**126**	**4.17**	**.70**
		II	**No-PB**	**111**	**3.79**	**.83**
	W *	**I**	**No-PB**	**111**	**4.01**	**.75**
		II	**PB**	**126**	**4.27**	**.71**
	D *	**III**	**PB**	**119**	**4.18**	**.79**
		IV	**No-PB**	**118**	**3.83**	**.87**
Difficulty	F	I	PB	126	4.06	.80
		II	No-PB	111	4.18	.75
	W	I	No-PB	111	4.16	.82
		II	PB	126	4.19	.73
	D *	**III**	**PB**	**119**	**4.22**	**.82**
		IV	**No-PB**	**118**	**4.02**	**.81**

Table 5: Comparison of helpfulness for learning programming concepts (1 = Using No-PB is much more helpful, 5 = Using PB is much more helpful) and preference (1 = strongly prefer No-PB, 5 = strongly prefer PB) for the environments the students had used so far. All have $p < .001$; medians = 4.

Item	Lab	Group	Env.	N	M	SD
Comparison of Helpfulness	W	I	No-PB	111	3.60	.97
		II	PB	126	4.16	.77
	D	III	PB	119	4.07	.75
		IV	No-PB	118	3.56	.86
Preference	W	I	No-PB	111	3.70	.99
		II	PB	126	4.12	.83
	D	III	PB	119	4.03	.87
		IV	No-PB	118	3.49	.98

> ***Takeaway 3**: At the end of each lab, students consistently rated PB as more helpful than No-PB. Although at the beginning of the academic term, students rated PB just as easy-to-use as No-PB, later in the term, students rated it easier to use.*

Preference Rating. Figures 2 and 3 show the distributions of the ratings. The visual intuition is that the "blue-ish" bars (light-blue and dark-blue) are the ones leaning toward PB and the "red-ish" bars (red and pink) are the ones leaning toward No-PB; the grey bars are neutral. We can see that students generally prefer PB and find PB more helpful.

Tab. 5 shows the Mann-Whitney U test results for the comparison of helpfulness for learning (Lab W: $U = 9280.5, p < .001, r = .30$; Lab D: $U = 9208, p < .001, r = .29$) and environment preference (Lab W: $U = 8654.5, p < .001, r = .221$; Lab D: $U = 9174.5, p < .001, r = .28$). First, we see that all means are over 3 (neutral) and often close to 4, while all medians are 4. This means that in general students leaned toward PB (as the visuals from Figures 2 and 3 already showed). Second, we can also see that students leaned *more* toward PB when PB was the last environment they used. More specifically, notice how in Tab. 5, when the "Env." column is PB, the means are higher than the corresponding row where "Env." is No-PB. The Mann-Whitney U test shows that these differences are *statistically significant.*

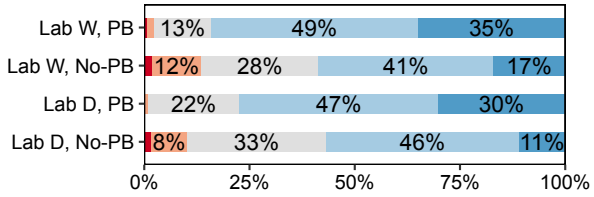

Figure 2: Experience survey question: "PB vs. No-PB, which helped you learn the lab material better?" ■ No-PB - much more helpful; ■ No-PB - More helpful; ■ About the same; ■ PB - More helpful; ■ PB - Much more helpful.

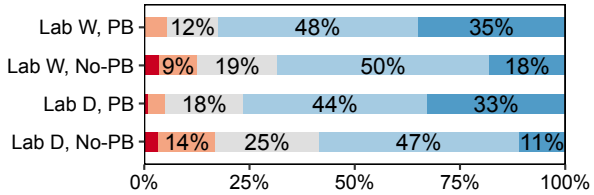

Figure 3: Experience survey question: "PB vs. No-PB, which do you prefer?" ■ Strongly prefer No-PB; ■ Prefer No-PB; ■ No preference; ■ Prefer PB; ■ Strongly prefer PB.

Putting all of the data together, we conclude the following:

> ***Takeaway 4**: Regardless of the environment used in each lab, students rated PB as more helpful and preferable for understanding lab materials, and those who used PB right before submitting the surveys rated these two items higher.*

Open-Ended Feedback Themes. Two authors analyzed open-ended feedback collected at the end of Labs F, W and D by open coding for thematic analysis, following the procedure similar to the one in [23]. To establish reliability of the analysis, we calculated percentage agreement [34] using 25% of the coded data. We consider two coders *agreed* on a data if they had at least one code in common or neither assigned codes to it. The two authors agreed upon 126 out of 138 sample data, yielding a high percentage agreement [16] of 91.3%. We identified the following themes along with quotes (note that thematic analysis does not indicate frequency of the themes, just their existence):

PB helpful for programming. As Projection Boxes visualize runtime values at every statement, students considered them helpful for understanding the behavior of *"each line of code"* and for noticing and locating errors faster *"without having to run the program."* Students found the assistance of tracing and bug locating altogether served as a more efficient alternative to `print` statements: *"I use the print statements to [see the behavior of my functions], and with the Projection Boxes, I get work done much faster."* A few also felt Projection Boxes helped them with writing code to produce desired outputs following the lab instructions.

Mixed views upon how PB assist with learning. As for how Projection Boxes helped with learning, we observed mixed views. Some students believed Projection Boxes assisted with learning new concepts and better understanding existing knowledge, and were especially *"helpful for beginners"* and *"good for visual learners."* However, some students argued that Projection Boxes were *"not more helpful"*

or *"no different"* than tools without PB in assistance with learning and understanding a concept, especially with concepts they were *"already familiar with."* A few students compared both critically: *"Without Projection Boxes, I don't have the 'training wheels' provided to me, so it takes a bit longer to run through [the code], but it also pushes me to try understand the code without help."*

***Frustration with* PB**. A theme that emerged is students' concerns with Projection Boxes *"cluttering up the screen"* and *"distracting when [they] try to work on the code."* Some found the clutteredness *"obtrusive,"* while others felt it made navigating Projection Boxes confusing: *" I first need to understand which part of the code the variables in the Projection Boxes come from."*

***Suggestions for* PB**. Our analysis also uncovered another theme, which is students' suggestions on the outlook and the usage of PB. Some students suggested customizability of the view: *"There should be 3 options: 1. Keep all boxes on. 2. Keep all boxes off. 3. When all boxes are off the option to cursor over is available."* One student noted it might be more helpful to use Projection Boxes for larger, more complex programs because it was rather easy to *"run and see the outputs"* for smaller code snippets as the ones in the labs. In addition, we noticed a few complaining about *"not [having] enough time"* to finish the lab and hoping to use Projection Boxes for an extended duration, even *"for [their] homework assignment."*

5 DISCUSSION

Observations from Prior Work. Our results reaffirm observations from prior work, but in the setting of a large introductory class with beginner programmers. Indeed, our results agree with: Sorva et al. [30, 32], who observed that students considered the Live Programming tool UUhistle helpful for understanding program behavior and predicting program output; Wilcox et al. [37], who found that participants felt Live Programming was helpful for debugging tasks; Kramer et al. [15], who noted Live Programming speeds up debugging because the programmer could constantly check their code's correctness after each incremental change; Lerner [17], who found that information overload hinders the use of Live Programming, and that participants had different preferences on the amount of information they found helpful.

Contradiction between Test Scores and Student Perceptions. In post-lab experience surveys, participants preferred Projection Boxes and rated Projection Boxes as more helpful for learning. And yet, we observed no impact of PB on performance in post-lab tests.

There are several ways of making sense of this apparent contradiction. One possibility is that learning *did* improve with Projection Boxes, but our post-tests with only code tracing questions *did not* detect this improvement. Another possibility is that while Projection Boxes could help students with code tracing, they could not improve students' own *ability* to trace the code without help and hence there were no improvements in code tracing task performance. In this situation the results are still interesting: our study demonstrates that we can provide students with an environment they prefer, while still achieving the same learning metrics (and in some cases, also having students go through the content faster). It would be interesting to conduct further studies that can differentiate between these possibilities.

This apparent contradiction should also be placed in the context of related studies. Both Campusano et al. [5] and Wilcox et al. [37] found that quantitative measurements of Live Programming didn't show an improvement in performance for debugging or code understanding. These two studies were small in scale, conducted with experienced programmers using in-lab controlled experiments, and assessed participants *while* using Live Programming. In contrast, our study involved a large sample of novices in a natural educational setting and assessed how students performed *after* they had used Live Programming. Also, Campusano et al. conducted longer interventions (4 hours) than Wilcox et al. or us (both 30 minutes), although we repeated the intervention four times in total. Since some of our participants suggested using Projection Boxes for longer durations with larger programs, future studies could investigate if such longer interventions might reveal any impact on performance in post-lab tests.

Limitations and Threats to Validity. One of the main threats to validity comes from the design of post-lab tests. First, we did not include code writing questions in the post-lab tests and thus have nothing to report on the acquisition of code writing skills. Second, participants answered up to six questions on small code snippets in the post-lab tests; additional questions might have revealed undetected impacts of using vs. not using Live Programming on learning. Third, we conducted the tests immediately after each lab; a more ecologically-valid study would be to evaluate the long-term effects by examining student grades on assignments and exams.

Another threat to validity relates to the length of the intervention. Our weekly 50-minute labs (with instructional video and post-test) limited the length of the intervention to 30 minutes once a week. Had students used Live Programming longer, possibly even in programming assignments, we would have been able to understand the effects on student learning with extended use of the tool.

Finally, the study was conducted remotely due to COVID-19 restrictions, and so there remain questions about how these results would translate to an in-person lab.

6 CONCLUSION

We conducted a within-subjects study at a large public university in a CS1 course in Python where 237 students completed tasks in a lab setting, in some case with a Live Programming visualization, and in some cases without. Through post-lab tests and surveys, we measured how well students understood the material and how students perceived the programming environment. While we found no significant difference in students' performance on any post-lab test regardless of whether Live Programming was used or not, students using Live Programming completed some code tracing tasks faster. We also found that students preferred the Live Programming environment and rated it as more helpful for their learning. We recognized that students' perceptions of the tool did not match their performance on post-lab tests, and we believe more work has to be done to better measure learning outcomes and evaluate how Live Programming affects student learning in an educational setting.

ACKNOWLEDGMENTS

The acknowledgement section of this work has been removed for the dual-anonymous review process.

REFERENCES

[1] 2021. IDLE – Python 3.9.6 Documentation. https://docs.python.org/3/library/idle.html

[2] Christine Alvarado, Briana B Morrison, Barbara Ericson, Mark Guzdial, and Brad Miller. 2012. *Performance and Use Evaluation of an Electronic Book for Introductory Python Programming.* Technical Report Technical Report GT-IC-12-02. Georgia Institute of Technology. https://smartech.gatech.edu/handle/1853/45044

[3] Benjamin Biegel, Benedikt Lesch, and Stephan Diehl. 2015. Live Object Exploration: Observing and Manipulating Behavior and State of Java Objects. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 581–585. https://doi.org/10.1109/ICSM.2015.7332518

[4] Lautaro Cabrera, John H. Maloney, and David Weintrop. 2019. Programs in the Palm of Your Hand: How Live Programming Shapes Children's Interactions with Physical Computing Devices. In *Proceedings of the 18th ACM International Conference on Interaction Design and Children*. ACM, Boise, ID, USA, 227–236. https://doi.org/10.1145/3311927.3323138

[5] Miguel Campusano, Alexandre Bergel, and Johan Fabry. 2016. Does live programming help program comprehension?–A user study with Live Robot Programming. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, Amsterdam, Netherlands, 8 pages. http://bergel.eu/MyPapers/Camp16-ComprehensionWithLRP.pdf

[6] Dimiter M. Dimitrov and Phillip D. Rumrill. 2003. Pretest-Posttest Designs and Measurement of Change. *Work (Reading, Mass.)* 20, 2 (2003), 159–165. https://pubmed.ncbi.nlm.nih.gov/12671209/

[7] Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (Feb. 1986), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

[8] Bruce B. Frey. 2018. *The SAGE Encyclopedia of Educational Research, Measurement, and Evaluation.* SAGE Publications, Inc., 2455 Teller Road, Thousand Oaks, California 91320. https://doi.org/10.4135/9781506326139

[9] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[10] Christopher Michael Hancock. 2003. *Real-time programming and the big ideas of computational literacy.* Thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/61549

[11] Juha Helminen and Lauri Malmi. 2010. Jype - A Program Visualization and Programming Exercise Tool for Python. In *Proceedings of the 5th International Symposium on Software Visualization - SOFTVIS '10*. ACM Press, Salt Lake City, Utah, USA, 153. https://doi.org/10.1145/1879211.1879234

[12] Matthew Hertz and Maria Jump. 2013. Trace-Based Teaching in Early Programming Courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 561–566. https://doi.org/10.1145/2445196.2445364

[13] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. Association for Computing Machinery, New York, NY, USA, 737–745. https://doi.org/10.1145/3126594.3126632

[14] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. 2009. Controlled Experiments on the Web: Survey and Practical Guide. *Data Mining and Knowledge Discovery* 18, 1 (Feb. 2009), 140–181. https://doi.org/10.1007/s10618-008-0114-1

[15] Jan-Peter Kramer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How Live Coding Affects Developers' Coding Behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 5–8. https://doi.org/10.1109/VLHCC.2014.6883013

[16] Klaus Krippendorff. 2011. Agreement and Information in the Reliability of Coding. *Communication Methods and Measures* 5, 2 (April 2011), 93–112. https://doi.org/10.1080/19312458.2011.568376

[17] Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3313831.3376494

[18] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *ACM SIGCSE Bulletin* 36, 4 (June 2004), 119–150. https://doi.org/10.1145/1041624.1041673

[19] Thomas Lumley, Paula Diehr, Scott Emerson, and Lu Chen. 2002. The Importance of the Normality Assumption in Large Public Health Data Sets. *Annual Review of Public Health* 23, 1 (2002), 151–169. https://doi.org/10.1146/annurev.publhealth.23.100901.140546

[20] Linxiao Ma, John D. Ferguson, Marc Roper, Isla Ross, and Murray Wood. 2008. Using Cognitive Conflict and Visualisation to Improve Mental Models Held by Novice Programmers. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 342–346. https://doi.org/10.1145/1352135.1352253

[21] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. 2019. A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education* 62, 2 (May 2019), 77–90. https://doi.org/10.1109/TE.2018.2864133

[22] Bradley N. Miller and David L. Ranum. 2012. Beyond PDF and ePub: Toward an Interactive Textbook. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*. Association for Computing Machinery, New York, NY, USA, 150–155. https://doi.org/10.1145/2325296.2325335

[23] Alexandra Milliken, Wengran Wang, Veronica Cateté, Sarah Martin, Neeloy Gomes, Yihuan Dong, Rachel Harred, Amy Isvik, Tiffany Barnes, Thomas Price, and Chris Martens. 2021. PlanIT! A New Integrated Tool to Help Novices Design for Open-Ended Projects. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. ACM, Virtual Event, USA, 232–238. https://doi.org/10.1145/3408877.3432552

[24] Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction* 7, 1 (March 2000), 3–28. https://doi.org/10.1145/344949.344959

[25] Greg L. Nelson, Benjamin Xie, and Amy J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, Tacoma, Washington, USA, 2–11. https://doi.org/10.1145/3105726.3106178

[26] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–32. https://doi.org/10.1145/3290327

[27] John Paxton. 2002. Live Programming as a Lecture Technique. *Journal of Computing Sciences in Colleges* 18, 2 (Dec. 2002), 51–56.

[28] Adalbert Gerald Soosai Raj, Pan Gu, Eda Zhang, Arokia Xavier Annie R, Jim Williams, Richard Halverson, and Jignesh M. Patel. 2020. Live-Coding vs Static Code Examples: Which Is Better with Respect to Student Learning and Cognitive Load?. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. ACM, Melbourne VIC Australia, 152–159. https://doi.org/10.1145/3373165.3373182

[29] Ana Selvaraj, Eda Zhang, Leo Porter, and Adalbert Gerald Soosai Raj. 2021. Live Coding: A Review of the Literature. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ACM, Virtual Event, Germany, 164–170. https://doi.org/10.1145/3430665.3456382

[30] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education.* Ph.D. Dissertation. Aalto University. https://aaltodoc.aalto.fi/handle/123456789/3534

[31] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education* 13, 2 (June 2013), 1–31. https://doi.org/10.1145/2483710.2483713

[32] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education* 13, 4 (Nov. 2013), 1–64. https://doi.org/10.1145/2490822

[33] Juha Sorva and Teemu Sirkiä. 2010. UUhistle: A Software Tool for Visual Program Simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*. Association for Computing Machinery, New York, NY, USA, 49–54. https://doi.org/10.1145/1930464.1930471

[34] Moin Syed and Sarah C. Nelson. 2015. Guidelines for Establishing Reliability When Coding Narrative Data. *Emerging Adulthood* 3, 6 (Dec. 2015), 375–387. https://doi.org/10.1177/2167696815587648

[35] Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing* 1, 2 (June 1990), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6

[36] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[37] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. 1997. Does continuous visual feedback aid debugging in direct-manipulation programming systems?. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. ACM, Atlanta Georgia USA, 258–265. https://doi.org/10.1145/258549.258721

[38] Benjamin Xie, Greg L. Nelson, and Amy J. Ko. 2018. An Explicit Strategy to Scaffold Novice Program Tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, Baltimore Maryland USA, 344–349. https://doi.org/10.1145/3159450.3159527

[39] Xiong Zhang and Philip J. Guo. 2017. DS.Js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, Québec City QC Canada, 691–702. https://doi.org/10.1145/3126594.3126663