Examining the Trade-Offs Between Simplified and Realistic Coding Environments in an Introductory Python Programming Class

Huy A. Nguyen(✉), Christopher Bogart, Jaromír Šavelka, Adam Zhang, and Majd Sakr

Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, USA
{hn1,cbogart,jsavelka,yufanz,msakr}@andrew.cmu.edu

Abstract. Instructors in computer science classes often need to decide between having students use real programming tools to provide practical experience and presenting them with simpler educational interfaces to reduce their cognitive load. Our work investigates the trade-offs between these approaches, by comparing student learning from two offerings of an introductory Python class across several community colleges in the U.S. In the first offering ($N = 219$), students used a real IDE (Visual Studio Code) throughout the entire course. In the second offering ($N = 166$), students used a simplified in-browser code editor, with no setup, for the first three modules and transitioned to Visual Studio Code in the subsequent modules. Our results showed that the second offering led to better learning than the first offering in the first three modules with the in-browser code editor. Moreover, students in both offerings performed similarly in a subsequent module in which they performed local development with Visual Studio Code, suggesting that the ability to use a real IDE was not harmed by the initial use of the in-browser code editor. In addition, we found that students in both offerings improved in their levels of self-efficacy with the course's learning objectives at the end of the class. Finally, we identified that the revisions made in the second offering benefited full-time students more than part-time students. We conclude with a discussion of the trade-offs between employing realistic programming tools and simplified coding environments, as well as suggestions for making introductory computer science classes more effective and accessible.

Keywords: online learning · project-based learning · introductory programming

1 Introduction

The introductory phase of computer science education poses significant challenges for instructors and course designers. They must impart challenging technical skills and a resilient mindset, while fostering a learning environment that

R. Ferreira Mello et al. (Eds.): EC-TEL 2024, LNCS 15159, pp. 315–329, 2024.
https://doi.org/10.1007/978-3-031-72315-5_22

encourages retention and persistence among students [25]. High attrition rates in introductory programming courses are a concern particularly in community colleges [8], some of which do not have the institutional resources to help students persist, such as tutors and in-person office hours. Furthermore, the student population in these classes is highly diverse, with many part-time students who need to balance school with full-time jobs or child care [18]. Hence, instructors need strategies that engage and support students through their initial foray into the discipline [15,22].

To this end, prior research has investigated ways to ease the initial learning experience while allowing students to focus on developing computational thinking skills [27,29]. Possible strategies include using visual block-based programming [33,34] and assigning tasks with higher levels of scaffodings (e.g., code tracing [36], Parsons problems [11–14], pair programming [23]). Although the benefits of these strategies are well-established, due to their deviation from the traditional coding experience, they often require significant resources to design and implement – resources which community college instructors cannot afford. Thus, it remains desirable to develop solutions that can reduce cognitive load while being easily utilized by students and instructors.

A potential area of improvement is the programming environment being introduced to students. While industry-standard environments (e.g., IntelliJ, PyCharm) are commonly used to foster an authentic learning experience, the complexities of these tools can be discouraging to novice programmers [9]. On the other hand, in-browser IDEs, which allow for code authoring and execution in the browser without any technical setups, are becoming increasingly common [31]. If they can promote learning as well as traditional environments, while not hampering students' ability to transition to practical workflows later on, in-browser IDEs may greatly contribute to lowering the entry barriers to computer science.

Our research investigates this hypothesis by comparing students' experience in two versions of an introductory Python course: the *pre-revision* version in which students complete programming assignments using practical tools, and the *post-revision* version in which the first few assignments utilized an in-browser code editor, embedded within the assignment description text. Using data collected from an online learning platform which hosted the course content, we evaluated the effects of the course revision on students' performance in each assigned module (RQ1). Next, we compared the learning behaviors, as well as attitudes associated with frame of mind and persistence in the field – namely self-efficacy and STEM identity – between the two class versions (RQ2). Finally, we investigated how students' background factors and attitudes contributed to their learning outcomes in each version of the class (RQ3).

2 Background

2.1 Educational Programming Environments in Introductory Python Course

Introductory programming courses tend to be challenging for students, with attrition rates around 28% [4]. This is often attributed to the expectations that students digest several topics simultaneously, including logic and mathematical concepts, programming language syntax and interface, algorithms, flowcharts, and pseudo code associated with programming theory, which often leads to higher levels of stress, cognitive overload and frustration [16]. Among these topics, mastering the language syntax is especially critical and time-consuming [1], given the major differences between programming languages and natural languages. Therefore, programming instructors and researchers have been extensively exploring how to reduce cognitive load of the students in introductory classes [30].

The use of a real programming IDE could be an additional source of cognitive load because these tools have many advanced features that may overwhelm some students, especially those with no programming background [9]. While some argue that it is important to familiarize students with the real-world tools [6], others choose to use IDEs with fewer features that are less overwhelming but fail to expose the students to real life programming environments [21]. To better understand the trade-offs between these directions and identify a resolution, we propose and evaluate an approach where students are first introduced to a simple environment with limited functionalities, and later switched to a real-world programming IDE (Visual Studio Code). This experiment was performed in the context of an introductory Python course, described in details as follows.

2.2 Structure of the Examined Python Course

The Python course in our work was designed for learners with minimal or no programming experience, starting with introductory topics such as data types, loops, and data structures. When students need to interact with the command line interface, exact commands are provided for them to copy-and-paste. When students need to run a Python program locally, a placeholder program that is executable but with missing functionalities is provided to them, where they are tasked with implementing individual functions or modules as part of their assignments. Students view the assignment description online, and submit their code either by typing code in an in-browser code editor (in the latest offering) or by running a submitter Python script on their personal or lab computer (in a previous offering). Figure 1 illustrates how the students' working environments differ in each offering. In the top environment (in-browser code editor) students can click on the Run button to execute their code and the Evaluate button to submit. In the bottom environment (Visual Studio Code), students execute their code through the IDE and use the terminal to submit.

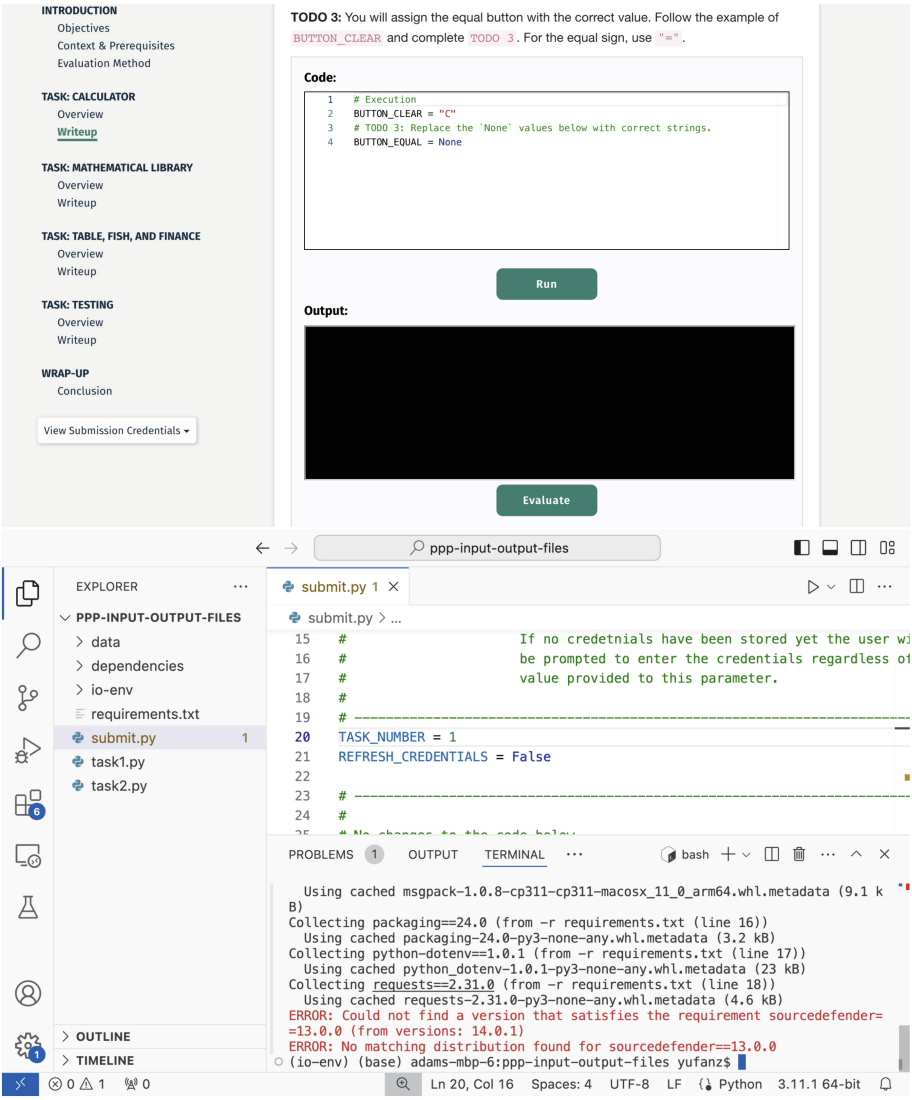

Fig. 1. The working environments for students in the latest offering (top) and previous offering (bottom)

The Python course is hosted on a proprietary platform where students' code is submitted to an auto-grading service, which immediately provides both scores and contextualized feedback (Fig. 2) to help students debug and help instructors understand their students' code more quickly. As detailed in Sect. 3.3, changes to the course have been made to simplify the coding workflow by moving the

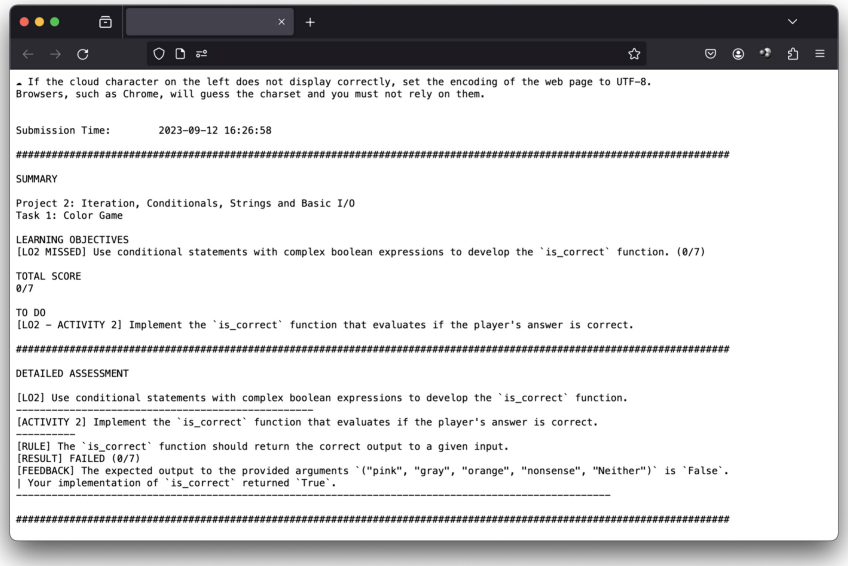

Fig. 2. The feedback shown to students for a submission that contains an incorrect solution.

development and runtime environment to the browser in the latest offering, which sets the stage for comparing the two paradigms in this paper.

Between its initial release in Fall 2022 and Spring 2024, the Python course (and all of its variants) has been taken by 955 students, in 42 offerings, at 21 institutions across the U.S.

3 Methods and Materials

3.1 Dataset

While the Python course examined in this work has been offered at several institutions, each participating instructor could decide on which modules of the course to use in their own section. For our research, we focused on the sections where all of the first three modules were mandatory, as the in-browser code editor and curriculum revision were primarily targeting these modules. Based on this criterion, we gathered student data from twenty-two sections of the class across nine community colleges in the U.S. Fourteen of these sections ($N = 219$) were offered prior to the course revision, with a mean student age of 29.04 ($SD = 11.69$); we refer to this sample as the *pre-revision group*. Eight sections ($N = 166$) were offered after the course revision, with a mean student age of 22.73 ($SD = 9.02$); we refer to this sample as the *post-revision* group.

3.2 Survey Materials

Students and instructors were presented with an informed consent form when they first logged into the course; those who did not consent to data collection could still take the course, but their data was not recorded. Students who consented then completed an optional starting survey with two primary components. The first component queried about their gender identity, age, student status (full time versus part-time) and other demographic factors. The second component, called the frame-of-mind survey, was designed to measure students' sense of self-efficacy and STEM identity. In particular, it covers the following constructs:

- The *STEM career self-efficacy* construct (4 items adapted from [19]), measures one's confidence in their ability to succeed in a STEM career, e.g., "I can persist in a STEM major during the next year."
- The *course LO self-efficacy* construct (5 items) measures one's confidence in their ability to meet the course's specific learning objectives, e.g., "I could write a small computer program that reads data from several files and produces a summary report stored in a new file."
- The *STEM Identity* construct (1 item adapted from [20]) measures the extent to which one identifies as a STEM professional.

All survey items were rated by students on a Likert scale from 1 ("not at all confident") to 5 ("extremely confident"), except for the STEM identity item with a scale from 1 to 7. An identical frame-of-mind survey was also offered at the end of the class in order to measure how students' attitudes changed over the course of their learning.

3.3 Learning Materials

After completing the surveys, students in both the pre-revision and post-revision groups went through the learning materials in the same manner. In each module, students would start with the conceptual readings and primer materials, then attempted the hands-on coding project. Each project contained one to six tasks which could be submitted separately, for a total of 95 points. The final 5 points in a module came from students' reflections on the project (2 points) and their discussion of others' reflections (3 points) – these activities would be unlocked after the project due date, if the instructor opted to assign them.

There were several changes to the post-revision sections of the Python class, detailed as follows. In terms of the coding environment, the first three graded modules (Modules 1, 2 and 3) were redesigned to utilize the in-browser code editor (Fig. 1, top), while all subsequent modules retained the local IDE environment, similar to those in the pre-revision sections (Fig. 1, bottom). In terms of the assignment contents, two tasks in Module 2 and 3 were moved to a later module in order to balance the initial difficulties. In terms of the curriculum revision, a new ungraded Module 0 was added to introduce the in-browser code editor feature. After Module 3, there was another ungraded module which taught

Pre-revision Module	Post-revision Module	Note
Module 0: Learning with *Platform X*: Developing Python programs locally	**Module 0**: Learning with *Platform X*: In-browser Submission	
Module 1: Types, Variables, and Functions	**Module 1**: Types, Variables, and Functions (in-browser IDE)	
Module 2: Iteration, Conditionals, Strings	**Module 2**: Iteration, Conditionals, Strings (in-browser IDE)	One task moved to Post-revision Module 5
Module 3: Lists, Sets, Tuples, Dictionaries	**Module 3**: Lists, Sets, Tuples, Dictionaries (in-browser IDE)	One task moved to Post-revision Module 5
-	**Module 4**: Learning with *Platform X*: Developing Python programs locally	Identical to Pre-revision Module 0
-	**Module 5**: Working with Files	Comprises two tasks moved from Modules 2 and 3
Module 4: Classes, Objects, Attributes and Methods	**Module 6**: Classes, Objects, Attributes and Methods	
Module 5: Debugging, Refactoring, Testing, Packaging	**Module 7**: Debugging, Refactoring, Testing, Packaging	
Module 6: Files and Datastores	**Module 8**: Files and Datastores	Identical between pre-revision and post-revision
Module 7: Web Scraping and Office Document Processing	**Module 9**: Web Scraping and Office Document Processing	
Module 8: Data Analysis	**Module 10**: Data Analysis	

Fig. 3. All the modules in the pre-revision and post-revision sections of the Python class.

students about setting up a local working environment with Visual Studio Code and the terminal – this module is identical to Module 0 in the pre-revision sections, which employed local development from the beginning. Finally, we added a new graded module, Module 6, which contained the two removed tasks from Modules 2 and 3. A complete overview of the curriculum revision is provided in Fig. 3. Here we note that, starting from the *Classes, Objects, Attributes and Methods* module, there were no differences in the learning content or working environment between the pre- and post-revision sections.

4 Results

RQ1: How Did the Course Revisions Impact Students' Learning Outcomes? We first compared students' final scores in Modules 1, 2, 3, as the in-browser code editor was incorporated into these three modules in the post-revision sections of the class. For Modules 2 and 3, we only considered the tasks that were common in both the pre-revision and post-revision sections. For consistency, we also normalized all student scores to be in the range of $0 - 100$. Table 1 shows the descriptive statistics for the final scores in each module.

Because the score distributions were not normal but bimodal, with one mode at 0 (from students who did not attempt or make any successful submission) and one mode around the mean, we used non-parametric Mann-Whitney test to compare the pre-revision and post-revision group. Our results showed a significant difference in final scores in Module 1 ($U = 15618.5$, $p = .01$), Module 2

Table 1. The mean and standard deviation, expressed in $M(SD)$ format, of students' final scores in each module.

Module	Pre-revision	Post-revision
Module 1: Types, Variables, and Functions	52.03 (46.69)	71.43 (42.99)
Module 2: Iteration, Conditionals, Strings	67.38 (41.72)	76.30 (37.71)
Module 3: Lists, Sets, Tuples, Dictionaries	39.85 (37.74)	57.65 (37.56)

($U = 13780.0,\ p < .001$), as well as Module 3 ($U = 13449.0,\ p < .001$). In all cases, the post-revision group had higher scores than the pre-revision group.

Next, we compared the final scores in later modules which were not affected by the course revision (i.e., all the modules in Fig. 3 starting from *Classes, Objects, Attributes and Methods*). However, because most of these later modules were made optional by the course instructors, we only had sufficient data to compare performance in the *Classes, Objects, Attributes and Methods* module. For this comparison, we identified a subsample of 17 sections where this module, in addition to the first three modules, was mandatory, with 10 sections ($N = 134$) in the pre-revision group and 7 sections ($N = 153$) in the post-revision group. Here the post-revision group ($M = 50.16, SD = 42.64$) also outperformed the pre-revision group ($M = 42.65, SD = 40.89$), but the difference was not significant, $U = 9221.5,\ p = .13$.

RQ2: How Did the Course Revisions Impact Students' Frame of Mind and Learning Behaviors? To measure how students' frame of mind constructs changed from the start to the end of the class, as well as whether this change was influenced by the course revisions, we conducted a series of mixed ANOVAs, with survey time (start of class versus end of class) as the within-subject factor, and course revision (pre-revision versus post-revision) as the between-subject factor. Our results showed a significant main effect of survey time on course LO self-efficacy ($F = 60.68,\ p < .001$), where students reported higher ratings at the end of the class ($M = 3.37, SD = 0.98$) than at the start ($M = 2.31, SD = 0.96$). For the other two constructs, STEM career self-efficacy and STEM identity, there were no significant main or interaction effects, indicating that the course revision did not have a clear impact on students' frame of mind.

Next, we analyzed how the course revision could lead to differences in learning behaviors. To this end, we compared the number of submissions that students made to each module, between the pre-revision and post-revision group. For Modules 2 and 3, only submissions to the tasks that were common in both the pre-revision and post-revision sections were considered. Based on a series of t-tests, we identified significant differences in the number of submissions across all modules with sufficient student data (Table 2). Compared to the pre-revision group, students in the post-revision group made many more submissions in Module 1 ($t = -11.95,\ p < .001$), Module 2 ($t = -8.71, p < .001$) and Module 3 ($t = -5.05, p < .001$), but fewer submissions in Module 4/6 ($t = 2.32,\ p = .02$).

Table 2. The mean and standard deviation, expressed in $M(SD)$ format, of students' submission counts in each module.

Module	Pre-revision	Post-revision
Module 1: Types, Variables, and Functions	15.06 (17.75)	59.50 (41.73)
Module 2: Iteration, Conditionals, Strings	27.23 (30.07)	107.69 (104.34)
Module 3: Lists, Sets, Tuples, Dictionaries	9.02 (20.77)	23.89 (26.06)
Module 4/6: Classes, Objects, Attributes and Methods	24.85 (37.78)	16.11 (14.10)

RQ3: Did the Effects of the Course Revision Vary Based on Students' Frame of Mind and Background? To determine the relationship between students' background factors, frame of mind constructs and the course revisions in their joint effects on learning outcomes, we constructed a series of regression models with 11 input features, which can be divided into four groups:

(1) The *Course revision* flag (Pre-revision = 0, Post-revision = 1), which indicates whether a student was in a pre-revision or post-revision section.
(2) Two background features, *Age* and *Full-time status* (Part-time = 0, Full-time = 1). The other background features were heavily skewed and therefore could not be used in the model.
(3) Frame-of-mind ratings at the start of the class, covering three constructs: *STEM career self-efficacy*, *Course LO self-efficacy* and *STEM identity*.
(4) Interaction terms between *Course revision* and each of the features in groups (2) and (3).

Correlation analysis showed no issues with multi-collinearity for the above set of features. Each model was then fit on students' final module scores in a particular module, and the significant predictor features are listed in Table 3. Overall, two features – *full-time status*, and the *interaction between full-time status and course revision*, were consistently predictive of students' learning performance across four modules. Figure 4 further illustrates the role of the interaction term, whereby the performance of full-time students improved after the course revision, but the performance of part-time students did not.

5 Discussion

In this work, we examined the effects of integrating an in-browser code editor and curriculum revision into the initial modules of an introductory computer science class. Our research was motivated by the potential trade-offs between having students use realistic tools from the outset and scaffolding them with simpler coding workflows. To this end, we compared the learning outcomes, learning behaviors and frame-of-mind constructs between students who took the class before the course revision and those who took it after. We discuss our findings and their implications on computer science instructions as follows.

Table 3. Regression features that significantly predicted students' final scores in each module ($p < .05$).

Module	Significant predictors	
Module 1: Types, Variables, and Functions	Full-time status,	$\beta = 22.77$
	Course revision × Full-time status,	$\beta = -33.27$
Module 2: Iteration, Conditionals, Strings	Course revision × Full-time status,	$\beta = -39.14$
Module 3: Lists, Sets, Tuples, Dictionaries	Full-time status,	$\beta = 27.84$
	Course revision × Full-time status,	$\beta = -35.97$
	STEM career self-efficacy,	$\beta = 9.17$
Module 4/6: Classes, Objects, Attributes and Methods	Full-time status,	$\beta = 29.28$
	Course revision × Full-time status,	$\beta = -43.18$

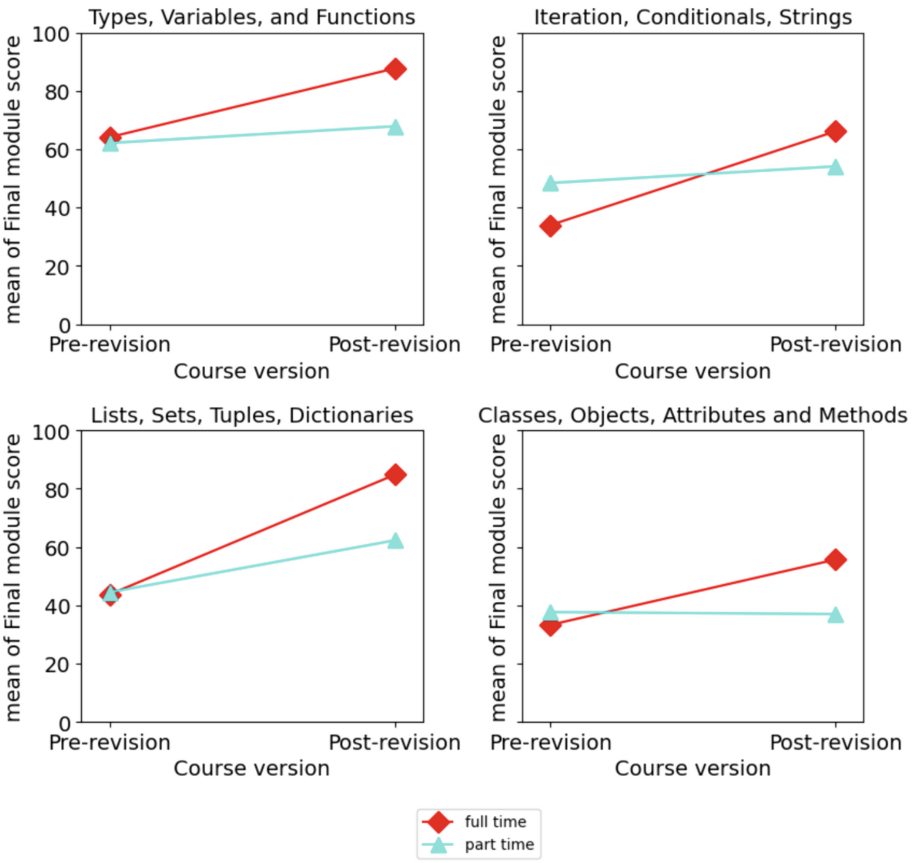

Fig. 4. Interaction plots showing the relationship between course revision, full-time status and final module scores.

First, we were able to confirm the learning benefits of the course revision. Compared to the pre-revision group, the post-revision group had higher scores in the first three modules, which utilized the in-browser code editor feature for code experimentations and submissions. In addition, when moving from the in-browser code editor to standard development tools (i.e., Visual Studio Code and the terminal) in a subsequent module, the post-revision group had comparable performance to the pre-revision group, suggesting that the simplified workflows in the first three modules did not hinder students' ability to use realistic tools. These results are encouraging and consistent with our motivation for the course revision, which was to reduce extraneous cognitive loads at the initial stages of learning. In the context of Python programming, significant cognitive overload could result from technical topics such as IDE functionalities, virtual environment setup and file system management, which are typically exposed to students at the same time they start learning the language syntax and develop computational thinking skills [30,37]. Towards addressing this issue, our findings provide support for an alternate strategy, which centered the students' learning experience on the core Python programming before introducing the intricacies of realistic development. We plan to continue evaluating this approach in subsequent iterations of our course, as well as in other introductory programming classes which may involve different languages and development tools.

When investigating the effects of the course revision on students' frame of mind, we noted several interesting patterns. First, there were no differences in STEM career self-efficacy and STEM identity between the start and end of class, or between the pre-revision and post-revision sections. This finding is consistent with the results from a prior work, which show that these constructs are relatively stable [7]. On the other hand, students' course LO self-efficacy, which measured their confidence in meeting the course's learning objectives, significantly improved from the start to the end of class, indicating that they perceived the benefits from the learning materials. However, this improvement did not differ between the pre-revision and post-revision group. In other words, while the post-revision class sections yielded better learning outcomes, they did not improve students' self-efficacy with respect to the course's learning objectives more than the pre-revision sections. One possible explanation is that, while the students' workflow in the initial modules were greatly simplified with the course revision, the content difficulties and amount of instructional support remained unchanged; thus, students' gain in self-efficacy was consistent in both the pre-revision and post-revision offerings. At the same time, given the identified connections between self-efficacy, knowledge acquisition and transfer [26], a promising venue of future research is to enhance the existing learning system to also promote self-efficacy. As an example, with the in-browser code editor, we are able to collect finer-grained data about students' progress through each assignment, including their coding samples and any errors they encountered. These types of data could be used to provide immediate affective, cognitive and metacognitive feedback, with the goal of helping students better navigate their learning trajectories and improve in self-efficacy [5,10,32].

In terms of learning behaviors, we observed significantly more submissions to the first three modules in the post-revision sections. This pattern is likely due to the ease of submission provided by the in-browser code editor, which only involved a button click. In contrast, students in the pre-revision sections needed to run a submitter script locally and then navigate to the learning platform to see their feedback (Figs. 1, 2). However, the higher number of submissions by the post-revision group, which exceeded even 100 on average in Module 2, could indicate that students were making minimal changes between submissions and using the auto-grading service to debug their code. This behavior is similar to "gaming the system," whereby students abuse the features of an interactive learning system to avoid thinking through a problem [3,17]. Given its potential detrimental effects, future iterations of the course should implement features to discourage students from making rapid submissions, for example by providing a reminder or enforcing a time gap between submissions in the learning platform. On the other hand, when students in the post-revision group transitioned into realistic development tools in a subsequent module, they made fewer submissions but still achieved comparable scores as the pre-revision group. In other words, the post-revision group had higher learning efficiency, which further supports the benefits of the course revision.

Additionally, when examining how the effects of the course revision on learning outcomes interact with different student factors, we found that it benefited full-time students more than part-time students, as full-time students did significantly better in the post-revision sections, while part-time students did not. This difference could be due to full-time students being able to spend more time with the class materials, which affords them more opportunities to benefit from the in-browser editor and curriculum revision. As identified in prior research, time management is a crucial strategy for success in community college and is especially more important for part-time than for full-time students [18]. At the same time, there may be other underlying factors contributing to the full-time and part-time gap, such as issues of identity and marginalization [35]. It is possible that our course revision may not adequately address, or even worsen, such issues. Thus, we plan to further investigate differences in learning difficulties, perspectives and attitudes between full-time and part-time students to identify ways in which the course revisions could benefit both groups equally.

Finally, we should note certain limitations that could impact the interpretations of our findings and merit additional investigations. First, we have analyzed student data across several institutions that differ in learning modalities and resource availability. These are potential confounding factors that should be explicitly modeled in future research with a larger sample size to fully understand the effects of the course revision. Likewise, we currently have insufficient data to compare student performance in later modules of the class, due to these modules being made optional. We plan to continue co-operating with instructors in the adoption of these modules to provide students with a uniform learning experience and gather more data for our research. Moving forward, we also plan to expand the range of analyses, taking into account students' work habits, code

samples, reflections and discussions, to better capture their learning experiences. These analyses have been conducted in prior research on other computer science domains, such as data science, software engineering and cloud computing education [2,24,28]. Thus, we expect that they will also be insightful in the context of our course, which utilizes a similar centralized learning platform to engage with students.

6 Conclusion

In summary, our work has identified the benefits of integrating an in-browser code editor to simplify the initial workflows for students in an introductory Python programming class. This feature was shown to improve learning performance while not hindering the transition to a realistic IDE later on. We therefore recommend that, instead of deciding between simplified and realistic coding environments, computer science instructors should start with simplified development tools before introducing students to more realistic and complex workflows, so as to lower the entry barrier and reduce cognitive load. In the broader sense, this research showcases how innovative uses of technology-enhanced learning tools can improve students learning in a complex technical domain such as project-based programming, where attrition rates remain high. As computer science education continues to grow in scale and complexity, further refinements of such tools are essential to maintaining quality and accessible learning experiences.

References

1. Al-Imamy, S., Alizadeh, J., Nour, M.A.: On the development of a programming teaching tool: the effect of teaching by templates on the learning process. J. Inf. Technol. Educ. Res. **5**(1), 271–283 (2006)
2. An, M., Zhang, H., Savelka, J., Zhu, S., Bogart, C., Sakr, M.: Are working habits different between well-performing and at-risk students in online project-based courses? In: Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1, pp. 324–330 (2021)
3. Baker, R., Walonoski, J., Heffernan, N., Roll, I., Corbett, A., Koedinger, K.: Why students engage in "gaming the system" behavior in interactive learning environments. J. Interact. Learn. Res. **19**(2), 185–224 (2008)
4. Bennedsen, J., Caspersen, M.E.: Failure rates in introductory programming: 12 years later. ACM Inroads **10**(2), 30–36 (2019)
5. Berger, N., Hanham, J., Stevens, C.J., Holmes, K.: Immediate feedback improves career decision self-efficacy and aspirational alignment. Front. Psychol. **10**, 429533 (2019)
6. Bettini, L., Crescenzi, P.: Java-meets eclipse: an ide for teaching java following the object-later approach. In: 2015 10th International Joint Conference on Software Technologies (ICSOFT), vol. 2, pp. 1–12. IEEE (2015)
7. Bogart, C., An, M., Keylor, E., Singh, P., Savelka, J., Sakr, M.: What factors influence persistence in project-based programming courses at community colleges? In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, pp. 116–122 (2024)

8. Chen, X.: Stem attrition: College students' paths into and out of stem fields. statistical analysis report. nces 2014-001. National Center for Education Statistics (2013)
9. Chen, Z., Marx, D.: Experiences with eclipse ide in programming courses. J. Comput. Sci. Coll. **21**(2), 104–112 (2005)
10. Halmo, S.M., Yamini, K.A., Stanton, J.D.: Metacognition and self-efficacy in action: How first-year students monitor and use self-coaching to move past metacognitive discomfort during problem solving. CBE-Life Sci. Educ. **23**(2), ar13 (2024)
11. Hou, X., Ericson, B.J., Wang, X.: Integrating personalized parsons problems with multi-level textual explanations to scaffold code writing. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2, pp. 1686–1687 (2024)
12. Hou, X., Ericson, B.J., Wang, X.: Using adaptive parsons problems to scaffold write-code problems. In: Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1, pp. 15–26 (2022)
13. Hou, X., Ericson, B.J., Wang, X.: Parsons problems to scaffold code writing: Impact on performance and problem-solving efficiency. In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2, pp. 665 (2023)
14. Hou, X., Ericson, B.J., Wang, X.: Understanding the effects of using parsons problems to scaffold code writing for students with varying cs self-efficacy levels. In: Proceedings of the 23rd Koli Calling International Conference on Computing Education Research, pp. 1–12 (2023)
15. Jenkins, J.T., Terwilliger, M.G.: Examining strategies to improve student success in cs1. J. Comput. Sci. Coll. **35**(4), 124–132 (2019)
16. Kelleher, C., Pausch, R.: Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers. ACM Comput. Surv. (CSUR) **37**(2), 83–137 (2005)
17. Li, Y., Zou, X., Ma, Z., Baker, R.S.: A multi-pronged redesign to reduce gaming the system. In: International Conference on Artificial Intelligence in Education, pp. 334–337. Springer (2022)
18. MacCann, C., Fogarty, G.J., Roberts, R.D.: Strategies for success in education: time management is more important for part-time than full-time community college students. Learn. Individ. Differ. **22**(5), 618–623 (2012)
19. Marra, R., Bogue, B., Rodgers, K., Shen, D.: Self efficacy of women engineering students? three years of data at us institutions. In: 2007 Annual Conference & Exposition pp. 12–1262 (2007)
20. McDonald, M.M., Zeigler-Hill, V., Vrabel, J.K., Escobar, M.: A single-item measure for assessing stem identity. In: Frontiers in Education, vol. 4, p. 78. Frontiers Media SA (2019)
21. Milne, I., Rowe, G.: Difficulties in learning and teaching programming-views of students and tutors. Educ. Inf. Technol. **7**, 55–66 (2002)
22. Mohamed, A.: Teaching highly mixed-ability cs1 classes: a proposed approach. Educ. Inf. Technol. **27**(1), 961–978 (2022)
23. Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C., Balik, S.: Improving the cs1 experience with pair programming. ACM Sigcse Bull. **35**(1), 359–362 (2003)

24. Nguyen, H., Lim, M., Moore, S., Nyberg, E., Sakr, M., Stamper, J.: Exploring metrics for the analysis of code submissions in an introductory data science course. In: LAK21: 11th International Learning Analytics and Knowledge Conference, pp. 632–638 (2021)
25. Qian, Y., Lehman, J.: Students' misconceptions and other difficulties in introductory programming: a literature review. ACM Trans. Comput. Educ. (TOCE) **18**(1), 1–24 (2017)
26. A Ramalingam, V., LaBelle, D., Wiedenbeck, S.: Self-efficacy and mental models in learning to program. In: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, pp. 171–175 (2004)
27. Sands, P.: Addressing cognitive load in the computer science classroom. Acm Inroads **10**(1), 44–51 (2019)
28. Sankaranarayanan, S., Kandimalla, S.R., Bogart, C.A., Murray, R.C., Hilton, M., Sakr, M.F., Rosé, C.P.: Collaborative programming for work-relevant learning: Comparing programming practice with example-based reflection for student learning and transfer task performance. IEEE Trans. Learn. Technol. **15**(5), 594–604 (2022)
29. Santana, B.L., Bittencourt, R.A.: Increasing motivation of cs1 non-majors through an approach contextualized by games and media. In: 2018 IEEE Frontiers in Education Conference (FIE), pp. 1–9. IEEE (2018)
30. Stachel, J., Marghitu, D., Brahim, T.B., Sims, R., Reynolds, L., Czelusniak, V.: Managing cognitive load in introductory programming courses: a cognitive aware scaffolding tool. J. Integr. Des. Process. Sci. **17**(1), 37–54 (2013)
31. Tang, L.: A Browser-based Program Execution Visualizer for Learning Interactive Programming in Python. Ph.D. thesis, Rice University (2015)
32. Valencia-Vallejo, N., López-Vargas, O., Sanabria-Rodríguez, L.: Effect of a metacognitive scaffolding on self-efficacy, metacognition, and achievement in e-learning environments. Knowl. Manage. E-Learning **11**(1), 1–19 (2019)
33. Weintrop, D.: Block-based programming in computer science education. Commun. ACM **62**(8), 22–25 (2019)
34. Weintrop, D., Wilensky, U.: Comparing block-based and text-based programming in high school computer science classrooms. ACM Trans. Comput. Educ. (TOCE) **18**(1), 1–25 (2017)
35. Williams, J., Kane, D.: The part-time student's experience 1996–2007: An issue of identity and marginalisation? Tert. Educ. Manag. **16**(3), 183–209 (2010)
36. Xhakaj, F., Aleven, V.: Towards improving introductory computer programming with an its for conceptual learning. In: Artificial Intelligence in Education: 19th International Conference, AIED 2018, London, UK, June 27–30, 2018, Proceedings, Part II 19. pp. 535–538. Springer (2018)
37. Yousoof, M., Sapiyan, M., Kamaluddin, K.: Measuring cognitive load-a solution to ease learning of programming. Int. J. Comput. Syst. Eng. **1**(2), 32–35 (2007)