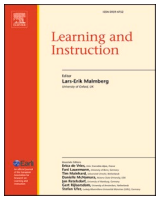
Learning and Instruction 91 (2024) 101871

Contents lists available at ScienceDirect

Learning and Instruction

journal homepage: www.elsevier.com/locate/learninstruc



Prediction versus production for teaching computer programming

Mary C. Tucker [a], Xinran (Wendy) Wang [b,*], Ji Y. Son [c], James W. Stigler [d]

[a] Barracuda Networks, USA
[b] Department of Psychology & Human Development, Vanderbilt University, USA
[c] Department of Psychology, Cal State LA, USA
[d] Department of Psychology, UCLA, USA

ARTICLE INFO

Keywords:
Computational education
Teaching/learning strategies
Generating prediction
Instructional sequence
Coding experience

ABSTRACT

Background: Most students struggle when learning to program.
Aims: In this paper we examine two instructional tasks that can be used to introduce programming: tell-and-practice (the typical pedagogical routine of describing some code or function then having students write code to practice what they have learned) and prediction (where students are given code and asked to make predictions about the output before they are told how the code works).
Sample: Participants were 121 college students with no coding experience.
Methods: Participants were randomly assigned to one of two parallel training tasks: predict, or tell-and-practice.
Results: Participants in the predict condition showed greater learning and better non-cognitive outcomes than those in the tell-and-practice condition.
Conclusions: These findings raise a number of questions about the relationship between programming tasks and students' experiences and outcomes in the early stages of learning programming. They also suggest some pedagogical changes to consider, especially in early introductions to programming.

1. Introduction

Computer programming is a complex skill with cognitive and non-cognitive challenges (McCracken et al., 2001). To develop flexible programming skills, students need to learn programming syntax (Altadmri & Brown, 2015; Sajaniemi & Navarro-Prieto, 2005), understand programming concepts (Bayman & Mayer, 1983; Cañas et al., 1994; Ma, 2007; Ma et al., 2007; Sirkiä & Sorva, 2012), and coordinate and apply this knowledge to solve novel problems (see Qian & Lehman, 2017 for a review). Students also need to regulate emotions that arise during learning (Bosch et al., 2014; D'Mello & Graesser, 2011), maintain motivation and engagement, and persist in the face of failure (Renumol et al., 2010).

The most common pedagogy for helping students to meet these challenges is what we might refer to as tell-and-practice (a phrase coined by Schwartz et al., 2011). With this pedagogy, students first watch as teachers explain some example code, and then try modifying or writing their own code to solve new problems. Tell-and-practice seems like a logical way to introduce students to programming and thus is a dominant strategy. But is it the best pedagogy?

In this paper, we consider the idea that certain aspects of the tell-and-practice pedagogy might negatively impact students' cognitive and non-cognitive outcomes. We start with an informal task analysis considering how tell-and-practice instruction might be experienced by students just beginning to learn to code: where is their attention drawn and what information is encoded as important? We then consider how to draw their attention to more critical information using an alternative pedagogy where students predict the outcome of code rather than practicing writing code. Ultimately, we compare tell-and-practice, in a random-assignment experiment, to a pedagogy designed around making predictions.

1.1. Cognitive and non-cognitive effects of tell-and-practice

Let's consider what students might experience during the "practice" portion of tell-and-practice. If students type in their code and it fails to run, it may be hard for them to identify exactly what went wrong (e.g., was it a missing comma? the wrong function? the algorithm was implemented incorrectly?). The primary focus for beginning students should be comprehending a given code's function and why it produces a particular output, but novices often fail to identify these most critical features (Kaczmarczyk et al., 2010). They may be overly focused on

* Corresponding author. Vanderbilt University, TN37203, USA.
E-mail address: wangxinran02013@gmail.com (X.(W. Wang).

https://doi.org/10.1016/j.learninstruc.2023.101871
Received 4 May 2023; Received in revised form 23 November 2023; Accepted 13 December 2023

smaller errors such as syntactic mistakes.

Using the wrong function should be considered a more fundamental error than a missing comma. However, a beginner may not have sufficient prior experience to make this distinction. Thus, when a novice student writes code that produces an error message, they may not be able to tell whether their error is a major or minor one. Missing a quotation mark and selecting the wrong function may be perceived as equivalent to a novice because both cause the code to fail. Tell-and-practice may not effectively direct students' attention to the parts of the code that are most important to learn. Even if students' code runs successfully during "practice," they might not fully understand why the code succeeds. Studies have found that these minor and major errors are indicative of distinct skills (Corney et al., 2011; Venables et al., 2009). For example, missing a quotation mark only indicates that students do not flawlessly write syntactically correct code (Robins et al., 2003). But selecting the wrong function can be a more critical sign that students are struggling understanding the code's purpose (Whalley et al., 2006).

In the Structure of Observed Learning Outcomes (SOLO) taxonomy (Biggs & Collis, 2014), when a student is able to identify critical features and appreciate the function of code, that aligns with the highest level of this hierarchical framework, the relational level. At this level, students demonstrate that they can "see the forest" and explain the purpose of the code rather than focusing solely on the details ("the trees"). These higher order learning outcomes positively relate to students' expertise and depth of understanding (Lister et al., 2006). Perhaps the pedagogy of introductory programming should consider how to help students achieve these higher order learning outcomes. Without guidance towards these critical learning outcomes, students may practice the details of code writing while leaving gaps in their functional knowledge of coding.

Beyond these cognitive aspects of the learning experience, traditional tell-and-practice also has potential non-cognitive disadvantages. When their written code doesn't run, students might experience negative emotions such as frustration. The inherently high cognitive load of learning to program (Groβe & Renkl, 2007; Sweller, 1994, 2010; Sweller et al., 2019; Zhu & Simon, 1987) may lead students to think the task is "too difficult," or "too complicated," and that the emotional cost of learning programming is too high (Flake et al., 2015). While receiving error messages and debugging code are inherent parts of coding, these negative emotions might lead students to interpret errors as signs of failure and incompetence. Frustration and perceived difficulty may cause students to form negative attitudes towards computer programming as a subject, which may discourage them from future learning.

Some might argue that confusion is a necessary part of learning. D'Mello and Graesser (2012) have posited that students experience cognitive disequilibrium and confusion when encountering impasses, anomalous events, obstacles to goals, and novelty. Faced with cognitive disequilibrium, students attempt to problem-solve. They are able to restore cognitive equilibrium if they effectively resolve the impasses. If they experience obstacles and cannot resolve the impasses (the feeling of being "stuck"), they are more likely to experience frustration which may lead to boredom and disengagement (Larson & Richards, 1991; Robinson, 1975). In learning programming, not being able to figure out why their code does not run may trigger problem solving but may also lead to frustration, boredom, and disengagement. Especially early on in their programming experience, they may lack strategies to problem-solve (e.g., checking for common syntactic errors) or lack the knowledge to interpret error messages and thus be more susceptible to frustration.

The traditional tell-and-practice format could be thought of as a pedagogy that gives students practice in *writing* code before practice in *reading* code. For example, when students learn about a new function, they may focus on reproducing it in practice activities instead of first understanding the code, learning what each part of the code might do, and relating the code to the expected output. While the tasks used in the traditional tell-and-practice approach involve writing code, we could imagine other approaches might focus on analyzing and comprehending code before writing it. There may be advantages, especially in the early stages of learning programming, to using different tasks with different affordances, which primarily focus on understanding code.

1.2. Possible cognitive benefits of predicting

Although instructors may value students learning to understand code, they may be unsure how to help students practice such understanding. One task that engages students in reading and analyzing code before writing it is prediction. In prediction tasks, students read code and then predict what will happen when the code is run. The code students are asked to read and predict could direct their attention to specific features of the code.

This approach aligns with prior studies that have highlighted the benefits of the Predict-Observe-Explain (POE) pedagogy (coined by White & Gunston, 1992). Specifically, the POE pedagogy contains three stages: predict what the code will do, observe what happens when the code is run, and then explain how the code works. To our knowledge, prediction tasks have mostly been studied outside of programming instruction, in domains as varied as reading comprehension, science, and mathematics (Brod, 2021). Prediction tasks generally ask students to make a hypothesis as to the outcome of a process based on their pre-existing knowledge, observe the outcome, and then compare the outcome with their prediction (Brod, 2021). For example, White and Gunstone (1992) used a predict-observe-explain pedagogy to promote conceptual understanding in chemistry. First, students would predict the outcome of a physical experiment, then observe the outcome of the experiment and, finally, reconcile any differences between their prediction and the observed outcome. Also, Miller et al. (2013) documented the benefits of incorporating prediction as a regular teaching routine for large STEM classes. Their results revealed that when students make a prediction before a scientific demonstration, they learn more, regardless of the correctness of their prediction.

Adapting the prediction pedagogy to the domain of programming could potentially address some of the challenges encountered in traditional tell-and-practice instruction. For example, in tell-and-practice, one risk is that students end up focusing on surface features rather than conceptual or functional understanding. However, with predicting, instructors could design effective prediction tasks that help students prioritize understanding and comprehension, direct attention to meaningful features, and encourage semantic processing (Brod, 2021). Predicting or comprehending code has been found to be a precursor skill to code writing (Lopez et al., 2008; Venables et al., 2009) and pedagogy that included code comprehension questions more effectively supports learning from worked examples (Clancy & Linn, 1999). In Xie et al.'s (2019) theory of instruction for introductory programming, they emphasized the importance of teaching programming skills in an appropriate order. They suggested that students should learn tracing before writing correct syntax, and they should learn code comprehension before applying code templates. In an exploratory study, they found that instruction that sequenced skills in this way deepened understanding and decreased errors in writing code on a posttest.

Emphasizing reading and comprehension of code might be of special importance for novices because they lack prior experience and expertise in identifying the important features that need the most attention. They may mistakenly perceive punctuation and syntax, superficial features that can easily trip up novices, to be the most challenging part of programming rather than conceptual understanding. For example, asking students to predict whether simple code such as 'num < - "eleven"' will result in printing num, "eleven", 11, or no output being printed at all may direct students' attention toward the function of the assignment operator (<-). In contrast, if students were asked to write code to save "eleven" into the object 'num', they would have to get many details correct in order for the code to run (e.g., write num in lower case, put quotation marks around the text "eleven", make sure the assignment operator is in the correct direction). Because this is a very simple example, the tell-and-practice pedagogy would result in many students

writing this code correctly. However, they may not notice the special relationship between the assignment operator and the object (num) and the contents ("eleven") in the same way that students who practice predicting might notice.

Prediction may also help students make more effective use of their limited cognitive resources during learning (also called cognitive load, Sweller, 2010a, 2010b). Especially because a novice can easily be overwhelmed by new information, prediction tasks may be a way to focus their attention on critical information without introducing extraneous task demands nor making the task too easy (e.g. Sweller & Cooper, 1985). In order to make a prediction, students do not have to memorize correct programming syntax which may produce extraneous load; but they do have to search for features of the code that would cause a particular outcome, a demand that is germane to the learning task at hand. Although the cognitive load may not be lessened, a novice's limited bandwidth can be deployed more efficiently, focused on critical features.

In addition to cognitive load, other cognitive mechanisms point to the possible benefits of prediction. Asking students to predict an outcome before they learn about it may draw upon the same cognitive mechanisms as the testing effect, a learning benefit seen when students attempt to answer questions about a topic before explicit instruction. Prior studies have found that asking students questions before giving explicit instruction leads to more engagement, more connections between prior knowledge and new information, and ultimately better learning (Pan & Carpenter, 2023; Little & Bjork, 2011; Little & Bjork, 2016). Similar to pretesting, asking students to make predictions may also hold potential to produce these benefits. In both pretesting and making predictions, students retrieve prior knowledge but also attempt to figure out novel aspects of the situation at hand.

Given the potential cognitive benefits to prediction, we want to go beyond proposing prediction as an alternative pedagogy: we want to experimentally compare it with a traditional tell-and-practice approach. The results, although specific to coding and modest in scope, may yield theoretical implications. Although there is a rich tradition in educational theory on how to sequence learning experiences (e.g., what should come first) going back to Bruner (1966) and many domain-specific theories in computer science education proposing what early programming instruction should look like (e.g., Xie et al., 2019 draws upon and also reviews many of them), specific experiments should also be conducted.

1.3. Possible non-cognitive benefits of predicting

Given the emotional states that occur during learning, we should also consider the possible non-cognitive effects of prediction pedagogy: students may create a different definition of "success in learning," give students ways of resolving cognitive disequilibrium, and remove premature threats to self-efficacy.

With a prediction task, students' definition of success may lean away from just whether or not the code runs and towards understanding why the code does what it does. If code does not run as expected (a common occurrence during learning), a student with a focus on understanding rather than "getting the code to run" might interpret that as an opportunity to learn and grow rather than as a sign of failure. For example, incorrectly predicting that the example code 'num < - "eleven"' would result in num being printed and seeing that nothing gets printed could make students more curious about what exactly is going on in this code. Over time, predicting might help students perceive computer programming as a continuous process of incremental learning and revising, thus developing more of a growth mindset (Dweck, 1999; Mueller & Dweck, 1998).

Prediction is also a way to practice the kind of code tracing that expert programmers engage in during debugging; thus teaching students a strategy for alleviating cognitive disequilibrium and resolving future issues. Seeing code outputs that are different from their own predictions may also evoke a different sort of disequilibrium than finding out that their code did not run. Incorrect predictions have the potential to produce more positive emotions such as surprise and curiosity, which can produce engagement and benefit learning (Brod et al., 2018; Miller et al., 2013; Theobald & Brod, 2021). Finally, making an incorrect prediction may not be as threatening to self-efficacy as writing buggy code. Instead of thinking of themselves as failures as coders, they might consider this a failure to understand. Making incorrect predictions also makes students aware of what they do and do not know. This differential may foster curiosity when they receive direct instruction or read explanations.

1.4. The current study

The current study explores predicting as a strategy for teaching programming. In a random-assignment experiment, we compare novice students working on prediction tasks to those working in the more traditional tell-and-practice context, and investigate both the cognitive and non-cognitive effects of the two approaches.

Undergraduate students with no prior experience in coding were randomly assigned to one of two conditions – 1) the *Predict* condition in which participants were given instruction and then asked to make and test predictions about pre-populated code, or 2) the *Traditional* instruction condition in which participants were given instruction and then asked to write or manipulate code on their own. Both conditions received the same explanatory text and instructional content. The only difference between the conditions was the task they were asked to perform: in one condition, students were asked to make predictions and run pre-provided code (Predict condition); in the other condition to write or manipulate and then run code on their own (Traditional condition).

We hypothesized that students assigned to the Predict condition would show increased learning and more positive outcomes on non-cognitive measures than students assigned to the Traditional condition. For cognitive measures, we expected that students in the Predict condition would have higher accuracy in the post-test as well as more correct solutions when multiple answers are allowed. For non-cognitive measures, we hypothesized that students in the Predict condition would report having a better learning experience, including more positive emotions in learning, more positive responses to error messages, less perceived difficulty in making sense of the instructional materials, and less perceived cost of learning.

2. Method

2.1. Participants

Participants were recruited from the online Psychology subject pool, SONA, at the University of California, Los Angeles (UCLA). Participants earned course credit for completing the study. To avoid biasing participants' responses and to ensure recruitment of participants who may not be interested in computer programming, participants were informed that the goal of the research study was to test a new online learning module, but not explicitly told that the task would involve programming. Students who were over 18 years old and not majoring in Computer Science or any other computational field were eligible to participate. To confirm their eligibility, students were asked to confirm this statement, "Yes, I am at least 18 years old and have never taken computer programming classes."

In total, 166 participants signed up for the study. Participants who did not complete the experiment ($n = 21$), who completed the experiment in more than one sitting ($n = 17$), or who took less than 10 min to complete the experiment ($n = 7$) were excluded. The resulting analytic sample comprised 121 students. Before starting the study, participants were randomly assigned by the Qualtrics (Provo, UT) software to either the Predict condition ($n = 60$) or the Traditional condition ($n = 61$). Of the final sample of 121 participants, all participants completed every

activity and answered every question. Seventy-six percent of the sample were female; one participant self-identified as non-binary, and the rest as male. The mean age was 20.35 years, SD = 2.56. Thirty-five percent described themselves as Asian, 23% as White, 17% Latino, 7% Middle Eastern/North African, 4% Black, 13% as more than one race, and one participant as Uzbek.

Chi-square analyses revealed no significant associations between condition and gender ($2(2) = 4.56$, $p = .1$) or condition and race/ethnicity ($2(2) = 6.38$, $p = .382$). Also, one way ANOVA showed that students' age ($F(1,118) = 2.06$, $p = .20$) and GPA ($F(1,118) = 1.91$, $p = .20$) did not significantly differ by condition.

2.2. Learning materials

The learning materials were divided into 4 modules and included a total of 17 brief activities. The modules were designed to mimic a self-paced introductory programming course for students with no coding experience (see Table 1 for a brief summary of the four modules and https://tinyurl.com/TraditionalCondition and https://tinyurl.com/PredictionCondition for the full materials). The modules interleaved explanations, worked examples, and questions (multiple-choice and written response), as well as interactive code blocks. Each module was designed to be completed in 10–15 min.

Table 1
Activities included in each of the 4 online modules.

Module	Topics covered	Example activities	Generate rule/ Summarize Question
Print() [5 activites] [very easy]	basics of the print() functions in R the differences between data types, such as numbers and characters, by using quotation marks	print ("Hello World") print (Hello world) print (1) str ("Good Morning") str (20)	Come up with a rule for when you need to use quotation marks and when you do not.
Arithmetic Operator [3 activites] [easy]	the differences between numbers and strings of characters uses the + operator to show students that R can be used as a calculator.	"7 + 7" 7 + 7 "7" + 7	Create a rule that would help you predict when the operator (like +) will produce a sum and when it will produce an error.
Objects [5 activites] [medium]	reinforce the aforementioned concepts the function of the <- operator in saving a single value and the result of a calculation capital lower case matter in R coding.	num < - 5 num name < - "Eleven" NUM < - 5 num < - 10 Num count < - 0 count < - count +1	Describe what the <- operator does.
Vectors [6 activites] [hard]	using < - operator to save multiple values calculating using vectors viewing a particular value in the vector Boolean comparison saving the result of Boolean comparisons using < - operator.	my.vector < - c (1,2,3,4,5) my.vector my.vector < - c (1,2,3,4,5) my.vector *100 days < - c ("Sunday", "Monday", Tuesday", "Wednesday", "Thursday", "Friday") days [5] = = "Friday" scores < - c (103, 200, 305, 180) high_score <- scores >200 high_score [1]	Come up with a rule for what you need to do to store a list of values in a vector.

2.3. Procedure

Upon signing up for the study, participants were sent a link by email to access the experiment. On clicking the link they were randomly assigned to either the Predict or Traditional instructional conditions. All participants began by reading an overview of the experiment, confirming their eligibility, and filling out a pre-survey. (See below for a complete description of measures.)

Next, participants worked their way through the series of four instructional modules. Participants assigned to the Predict condition were first provided some introductory information, shown some R code, and asked to make a prediction about what the code would do by selecting from one of five multiple-choice options (i.e., "When we click Run, which output do you think we will see?"). After making each prediction, students advanced to the next page, where they were reminded of their prediction and prompted to run the pre-provided code in order to see what happened. After students ran the code, they then reported what the output actually was. On the next page, students were provided text that explained the R code and why it worked the way it did.

Participants assigned to the Traditional condition were provided introductory information and some example code. They were asked to modify the code that was provided in order to produce a specified output, click "Run", and then report the output they actually saw (i.e. "Which output did you see?"). The multiple choice options were the same as those in the Predict condition. On the next page, students were provided an explanation of the correct R code and why it worked. Examples of how an activity appeared in the Predict and Traditional conditions are provided in Fig. 1.

After each of the four modules (each focused on a different concept or R function), students in both the Traditional and Predict conditions were asked a question that required them to generate a rule or summarize the topic of the module.

At the end of each module, participants in both conditions rated their emotions while completing the learning activities (R sentiment), and answered the generate-rule question at the end of the module. After all four modules were completed, participants completed a learning assessment and a flexibility assessment, followed by a post-survey that included questions about their age, gender, race/ethnicity, and GPA. Details regarding the learning assessment and post-survey measures will be discussed in the measures section below. See Fig. 2 for an illustration of the study procedure as well as a timeline of when various measures were administered.

The overall duration of the experiment was similar across conditions. The students in the Predict condition took a little longer overall ($M = 59.34$ min, $SD = 24.87$) than those in the Traditional condition ($M = 51.23$ min, $SD = 22.13$), though the difference was not statistically significant ($p = .6$). Students in the predict condition wrote slightly longer summaries of what they had learned at the end of each module ($M = 74.88$, $SD = 43.47$) than did those in the Traditional condition ($M = 61.42$, $SD = 30.98$), but again this difference was not statistically significant ($p = .6$).

2.4. Measures

A variety of measures were collected, both cognitive and non-cognitive.

2.4.1. Cognitive outcomes

Learning assessment. After the four learning modules and before the post-survey, participants completed a sixteen-item learning assessment ($\alpha = .83$) that covered concepts related to each of the four learning modules (Appendix A). The first fifteen items were used as a measure of learning. The last item was used as the flexibility assessment (see next section). The learning assessment contained five different question types: 1) *error identification questions*, in which participants identified

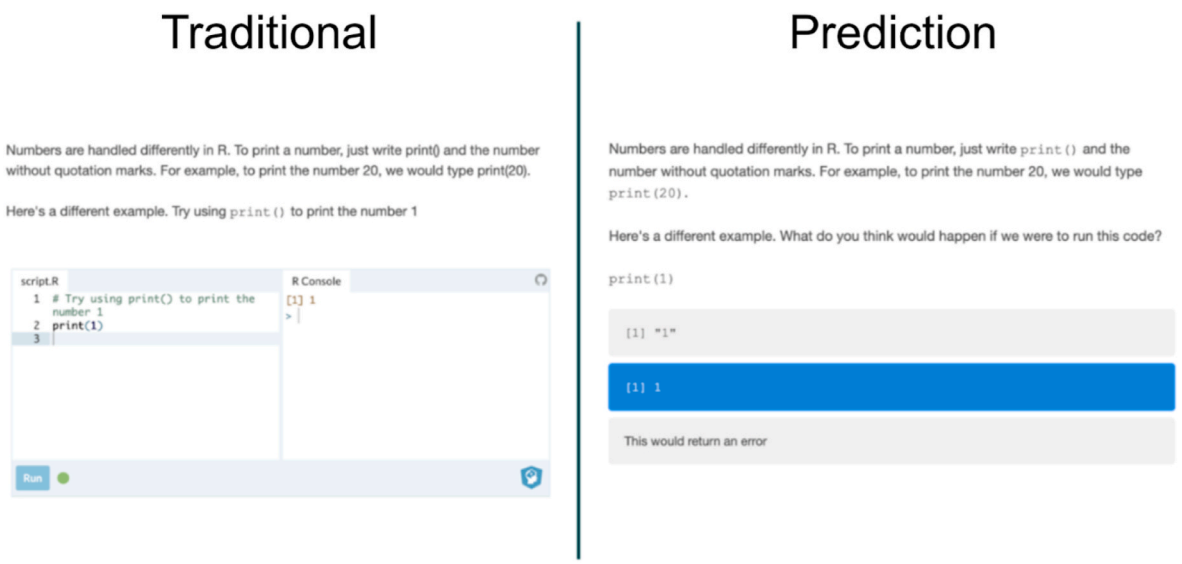


Fig. 1. Comparison of an instructional task in the traditional condition and the predict condition.

which lines of code (if any) from a sample contained an error (questions 1–1, 2–1, 3–1, 4–1, and 5–1); 2) *code writing questions,* in which participants were asked to generate code to produce a specified output (questions 7, 8, and 11) or rewrite a line(s) of code to run without error (questions 2-2, 3–2, and 4–2); 3) *code comprehension questions*, in which participants explained in their own word what a line of code does (questions 1-1, 2–3, and 4) or what output it will produce (question 12); *error interpretation activities*, in which participants were provided a line of incorrect code and asked to explain why it was wrong (question 10); and 5) *code identification tasks,* in which participants were asked to select, from a list, the correct piece of code to perform a given task (question 9).

The learning assessment was graded according to a rubric. For error identification questions, participants were awarded one point for each correct error identified. For code rewriting questions, participants were awarded one point for each error corrected. For code comprehension and error interpretation questions, participants were awarded one point if the explanation included all elements of the model response, half a point if the explanation included some but not all elements of the model response, and no points if the response contained none of the elements included in the model response. For code identification tasks, participants were awarded one point for identifying the correct code in the list and no points for selecting the other incorrect options. Thirteen of the questions were each worth one point each and two of the questions were worth two points each, yielding a total correct score between 0 and 17, with higher scores indicating greater learning.

Flexibility assessment. The last question on the learning assessment was a novel R coding task:

> How many ways can you use R to find the sum of 100 and 200? Write all the code you can think of in the space below. Label each solution with a number (i.e. 1 for solution number #1)

We used the number of unique correct solutions generated by each participant as a measure of flexibility (Guilford, 1967; Kwon et al., 2006).

2.4.2. Non-cognitive measures

R sentiment (measured at six time points). R sentiment (i.e., "How do you feel about R right now?") was measured at six time points throughout the study: t_1 (on the pre-survey), t_2 through t_5 (after each of the 4 learning modules), and t_6 (on the post-survey). Participants used a slider to rate their emotion (Munezero et al., 2014) on a continuous scale from −100 (Extremely negative) to +100 (Extremely positive) (α = .95).

Response to error (post-survey). Participants were shown a screenshot of some incorrect R code and the associated error message. They were asked "Imagine you are the student in the example above. How do you think you would feel in this situation?" They rated how they would feel, using a slider, on a scale from −100 (Extremely negative) to +100 (Extremely positive).

Cognitive Load Component Survey (post-survey). Nine items adapted from the Cognitive Load Component Survey (Morrison et al., 2014) were included in the post-survey to measure participants' perceptions of the activity they just completed. Each item was rated on a scale of 0–10. The 9 items were grouped into three subscales (of two to four items each) and averaged to measure perceived.

- Intrinsic cognitive load (2 items, α = .89):
  - o "The topics covered in the activity were very complex."
  - o "The activity covered code that I perceived as very complex."
- Extraneous cognitive load (3 items, α = .94):
  - o "The instructions and/or explanations during the activity were very unclear."
  - o "The instructions and/or explanations were, in terms of learning, very ineffective."
  - o "The instructions and/or explanations were full of unclear language."
- Germane cognitive load (4 items, α = .97):
  - o "The activity really enhanced my knowledge and understanding of computing/programming."
  - o "The activity really enhanced my understanding of the topic(s) covered."
  - o "The activity really enhanced my understanding of the program code covered."
  - o "The activity really enhanced my understanding of the concepts and definitions."

Perceived cost (pre- and post-survey). Both on the pre-survey and on the post-survey, students were asked on four items to evaluate how costly it would be for them to learn how to program. They were asked to rate their agreement, from "Strongly disagree" to "Strongly agree", with the following four statements on a six-point scale.

- "I think I would have to give up too much to learn programming."
- "For some reason, computer programming seems like it will be particularly hard for me."

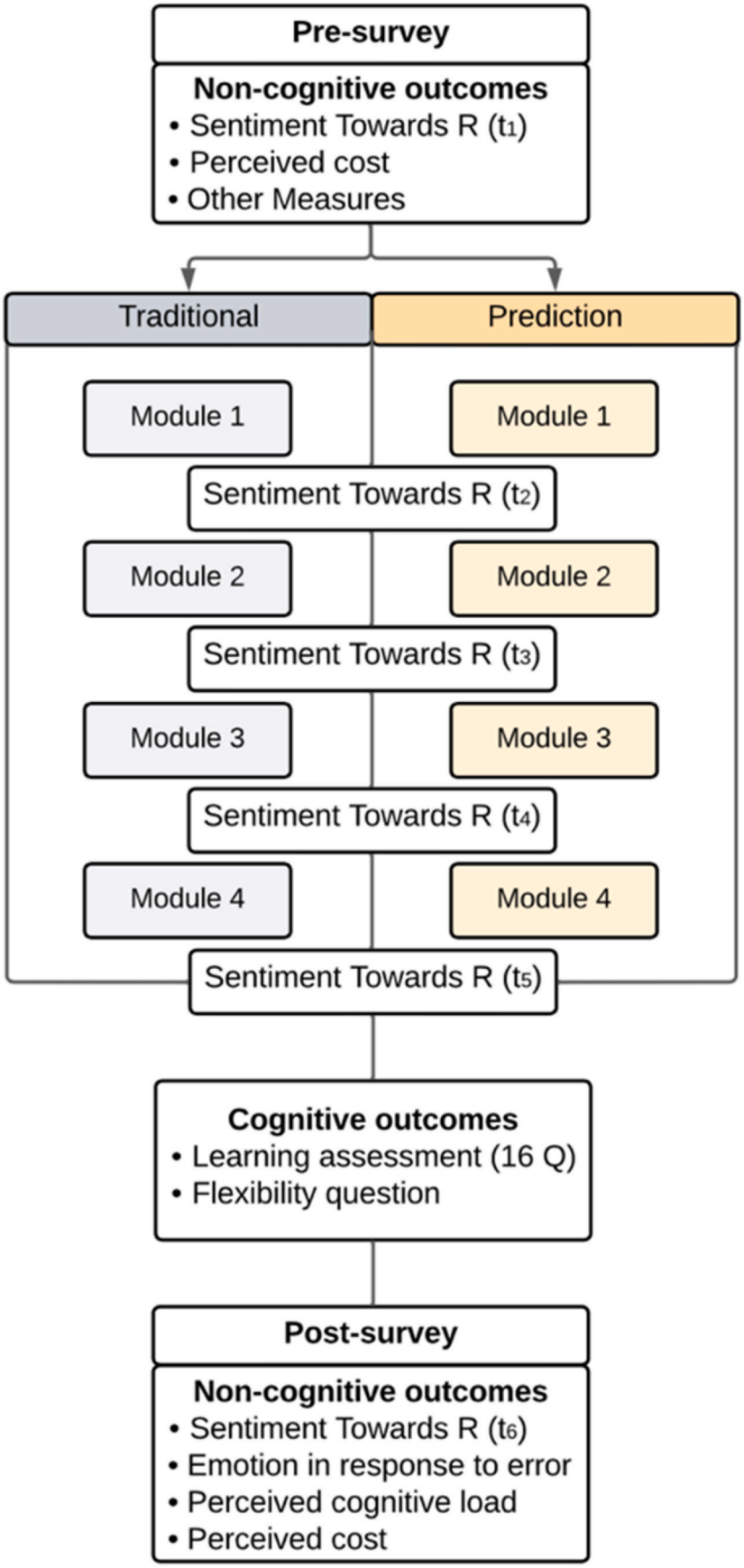


Fig. 2. Experimental procedure and measures.

- "I think learning computer programming would be too stressful for me."
- "I think learning computer programming would take up too much of my time."

Ratings were averaged across the four statements to get an overall indicator of perceived cost.

In addition to these non-cognitive measures, additional measures were collected as part of a larger study (for more information, see [Tucker, 2007]).

3. Results

For all of the following analyses, we used R version 3.6.2 (R Core Team, 2019).

3.1. Effect of condition on cognitive outcomes

To examine whether there was an effect of condition on cognitive outcomes (the learning and flexibility assessments), we conducted separate t-tests (shown in Table 2 along with descriptive statistics). On average, participants in the Predict condition scored significantly higher on the learning assessment and generated more unique solutions on the flexibility assessment than those in the Traditional condition (shown in Figs. 3 and 4).

3.2. Effect of condition on non-cognitive outcomes

3.2.1. R sentiment

As a reminder, students expressed their sentiment towards R by moving a slider on a continuous scale from −100 (extremely negative) to +100 (extremely positive) at six timepoints. As shown in Table 3, students in the two conditions did not differ significantly at the start of the study (t_1) or after the first module (t_2). However, students in the Predict condition demonstrated significantly more positive sentiment than the Traditional condition after the second, third, and fourth modules (t_3, t_4, t_5) and on the post-survey (t_6).

We used Growth curve analysis (Mirman, 2014) to analyze change in sentiment over the course of the six timepoints as shown in Fig. 5. The overall learning curves were modeled with third-order (cubic) orthogonal polynomials and fixed effects of Condition (Traditional vs. Predict) on all time terms. The Traditional condition was treated as the baseline and parameters were estimated for the Predict condition. The model also included random effects of participants on all time terms. The fixed effects of condition were added individually and their effects on model fit were evaluated using model comparisons. Improvements in model fit were evaluated using −2 times the change in log-likelihood, which is distributed as X^2 with degrees of freedom equal to the number of parameters added. Parameter estimates, degrees of freedom, and corresponding p-values were estimated using Satterthwaite's method. This analysis was carried out in R using the lme4 package.

Participants in the Predict and Traditional conditions did not significantly differ in sentiment towards R prior to beginning the activity ($X^2(1) = 3.42$, $p = .064$). However, students in the Predict condition increased their sentiment at higher rates than students in the Traditional condition ($X^2(1) = 9.50$, $p = .002$). There was also a significant effect of Condition on the quadratic term ($X^2(1) = 4.58$, $p = .03$) as well as a significant effect of Condition on the cubic term ($X^2(6) = 214.10$, $p <$ .0001).

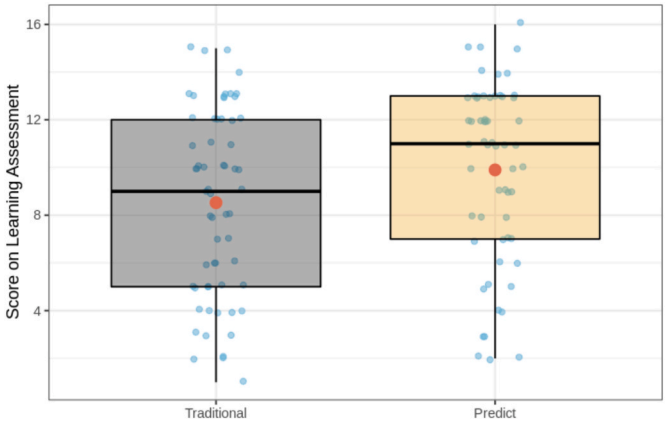


Fig. 3. Distribution of Learning Assessment Scores Broken Down By Condition
Note. Points represent the individual participants' scores. The horizontal line represents the median. The red dot represents the mean. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

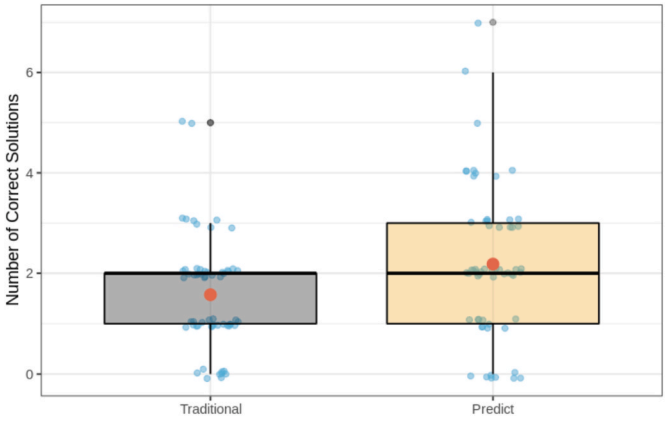


Fig. 4. Distribution of Flexibility Scores Broken Down By Condition
Note. Points represent the individual participants' scores. The horizontal line represents the median. The red dot represents the mean. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

Perhaps because the first module was easier, students in both conditions reported a rise in R sentiment after completing the first module. As the modules became more challenging, however, the Traditional group's R sentiment dropped, on average, while that of the Predict condition remained positive.

3.2.2. Effect of condition on response to hypothetical error

The two conditions differed significantly in how they would feel in a hypothetical situation in which they are shown negative feedback on a programming task (Fig. 6). Participants in the Predict condition rated their emotion, on average, more positively than participants in the Traditional condition ($b_1 = 28.77$, $F(1, 119) = 10.30$, $p = .002$, *95% CI* [11.02, 46.52], *Adjusted R^2* = 0.07248).

Table 2
Descriptive statistics, confidence intervals of mean differences, and T-tests comparing post-survey measures of learning between the predict and traditional instructional conditions.

	Predict *M (SD)*	Traditional *M (SD)*	*b*1 [95% CI]	*t(df)*	*Cohen's d*	*p*
Learning assessment (max = 16)	9.90 (3.72)	8.52 (3.90)	1.38 [0.00, 2.75]	1.99(118.91)	0.36	.05
Flexibility assessment (number of unique correct solutions)	2.18 (1.48)	1.57 (1.09)	2.183 [0.14, 1.08]	2.58(10Con8.32)	0.47	.01

Table 3
Descriptive statistics and T-Tests comparing sentiment ratings between the predict and traditional instructional conditions at each of the six time points.

Time	Predict *M* (*SD*)	Traditional *M* (*SD*)	*b*1 [95% *CI*]	*t*(*df*)	*Cohen's d*	*p*
1	7.65 (49.78)	−0.26 (50.97)	7.91 [-10.22, 26.05]	−0.86 (118.99)	.16	.4
2	46.75 (43.60)	30.93 (51.53)	15.82 [-1.38, 33.01]	−1.82 (116.41)	.33	.07
3	48.88 (40.70)	31.66 (50.25)	17.23 [0.75, 33.70]	−2.07 (114.79)	.38	.04
4	53.96 (43.55)	15.13 (56.55)	38.84 [20.64, 57.02]	−4.24 (112.56)	.77	<.0001
5	46.91 (46.57)	6.54 (60.81)	40.38 [20.86, 59.90]	−4.10 (112.29)	.74	<.0001
6	53.60 (46.11)	25.75 (59.50)	27.85 [8.66, 47.03]	−2.88 (112.86)	.53	.005

3.2.3. The effect of condition on perceived cognitive load

To test the effect of condition on students' perceived cognitive load, we conducted separate t-tests for each measure: intrinsic load, extraneous load, and germane load (descriptive and *t*-test statistics are available in Table 4). Although there was no significant difference between conditions in perceived intrinsic load ($p = .37$), students in the Predict condition perceived the extraneous load to be lower ($p = .005$) and the germane load to be higher ($p = .05$) than did those in the Traditional condition.

3.2.3.1. The effect of condition on cost. In the pre-survey, participants' perceptions of cost did not differ across conditions (Predict condition: $M = 3.54$, $SD = 0.99$, Traditional condition: $M = 3.75$, $SD = 1.1$), $t(118.2) = 1.13$, $p > .05$, *Cohen's d* = .21.

On the post-survey, however, there was a significant effect of condition on cost when controlling for pre-survey cost ratings ($b_1 = 0.40$, $F(2, 118) = 6.33$, $p = .013$, 95% *CI*[0.06, 0.72], eta squared = 0.93). Participants in the Predict condition perceived learning programming to be less costly than participants in the Traditional condition. Pre-survey perceived cost was also a significant covariate ($b_1 = 0.82$, $F(1, 118) = 122.10$, $p < .001$, 95% *CI*[0.66, 0.97], eta squared = 0.05), see Fig. 7.

3.3. Intercorrelations among measures

Finally, we examined the intercorrelations among all of the measures, both overall (Table 5) and separately within each condition (Appendix B). In general, the pattern of correlations is well aligned with expectations of cognitive load theory. Perceived extraneous cognitive load correlated negatively with perceived germane cognitive load, which makes sense. Intrinsic load correlated negatively with post-test scores, which we interpret to mean that participants who perceived the tasks as more complex also struggled more in their learning.

We also note that attitudes toward learning R correlated positively with students' response to the hypothetical error scenario, and that both of these measures were negatively related to perceived cost as reported on the post-survey. Together, these patterns support the internal consistency and validity of the measures within the context of our study.

3.4. Error analysis

We present an informal analysis of errors for further examination and reference in Appendix C.

4. Discussion

Learning programming is a long and challenging process for novice students, lending importance to research on interventions with the potential to enhance this process. The present study illustrates the impact of a simple pedagogical adjustment: having students predict outcomes before viewing the output and explanations. Results indicate that this approach leads to significant improvements in both cognitive and non-cognitive outcomes, compared to traditional code writing tasks. This study extends previous research on the effects of various programming tasks on instruction and demonstrates the potential of prediction as an effective strategy for teaching programming.

These findings align with prior studies that have highlighted the benefits of the Predict-Observe-Explain (POE) pedagogy (coined by White & Gunston, 1992) in subject areas such as biology and physics.

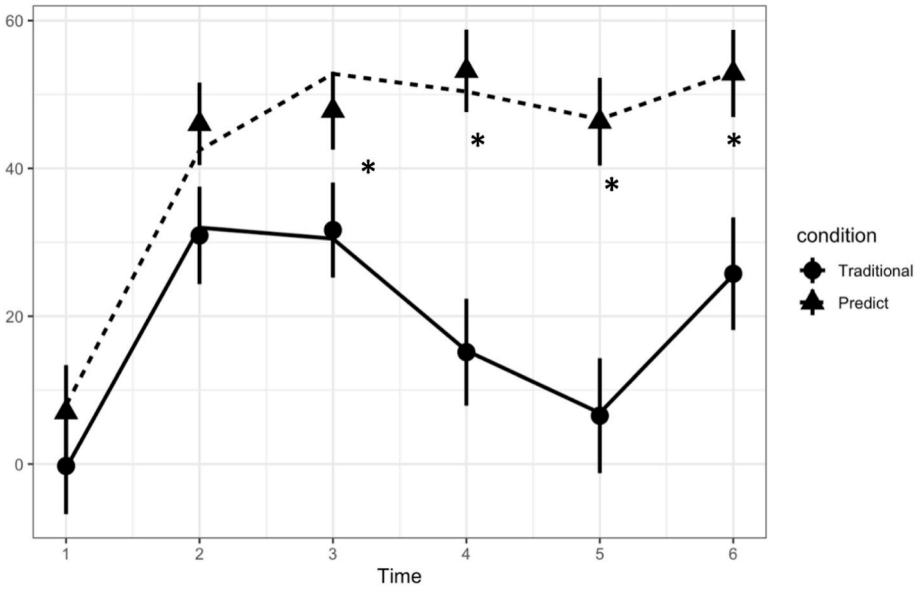


Fig. 5. Mean R Sentiment Across the Six Time Points by Condition
Note. * Indicates a significant mean difference between the two groups ($p < .05$). Triangles and dots indicate group means; vertical lines, standard errors. Dotted and solid lines represent predictions of the growth curve model.

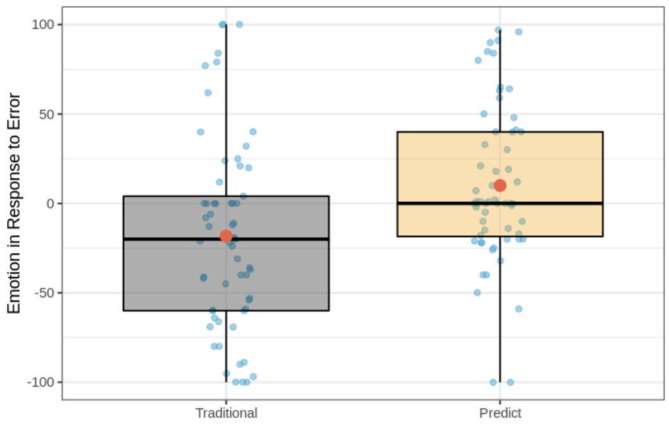


Fig. 6. Distribution of Emotion Ratings in Response to a Hypothetical Error Broken Down By Condition
Note. Points represent individual participants' scores. Horizontal lines represent the group medians, red dots, the means. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

Our brief coding modules each include activities that map onto the three stages of this approach: predict what the code will do, observe what happens when code is run, and then explain how the code works.

4.1. Cognitive outcomes

The results of this study indicate that students who were asked to generate predictions before viewing the output and explanations scored higher on the learning assessment than students who wrote code without making predictions. Additionally, despite having less code-writing practice, students in the Predict condition wrote more coding solutions than those in the Traditional condition on the flexibility assessment. Our study replicated prior research where the utilization of the POE approach in various educational contexts has been found to enhance students' comprehension and conceptual flexibility (Güngör & Özkan, 2016, December; Hong et al., 2021).

One possible explanation for the effectiveness of predicting in our modules is that it may have led to deeper semantic processing (Craik & Lockhart, 1972) of the code elements and syntax. Prior studies have found that directing novices' attention to key features could improve learning (Nadiah et al., 2021; Schnotz & Kürschner, 2007; Son et al., 2008). By requiring students to register their predictions before running the code, they may have paid more attention to key features, such as the purposes of different functions, that would lead to certain output. They may have come up with hypotheses about which of the features of the code would be causally related to potential outputs.

Another potential mechanism may be that this pedagogy helps students make more connections between concepts. The observed effects are consistent with prior research, indicating that the POE approach helps students to actively engage in the learning process by connecting prior knowledge with newly acquired information, leading to improved academic outcomes (Kırılmazkaya & Zengin-Kırbağ, 2015). Also, encouraging students to consider different possible outcomes and compare their predictions to the actual outcomes may have led students to find relations between concepts (Schwartz & Bransford, 1998; Schwartz et al., 2011). This interconnected understanding may have led to greater cognitive flexibility evidenced by the greater number of unique solutions.

4.2. Non-cognitive outcomes

The students in the Predict condition also had more positive

Table 4
Descriptive statistics and T-Tests comparing the predict and traditional conditions on cognitive load measures.

	Predict *M*(SD)	Traditional *M*(SD)	*b*1 [95% CI]	*t*(df)	*Cohen's d*	*p*
Intrinsic load	4.11(2.14)	4.45(2.03)	−0.34[-1.09,0.41]	0.90 (118.37)	.16	.37
Extraneous load	2.38(2.04)	3.45(2.10)	−1.07[-1.81,-0.32]	2.83 (118.98)	.52	.005
Germane load	7.23(1.95)	6.44(2.44)	0.79[-0.01,1.59]	−1.97 (114.17)	.36	.05

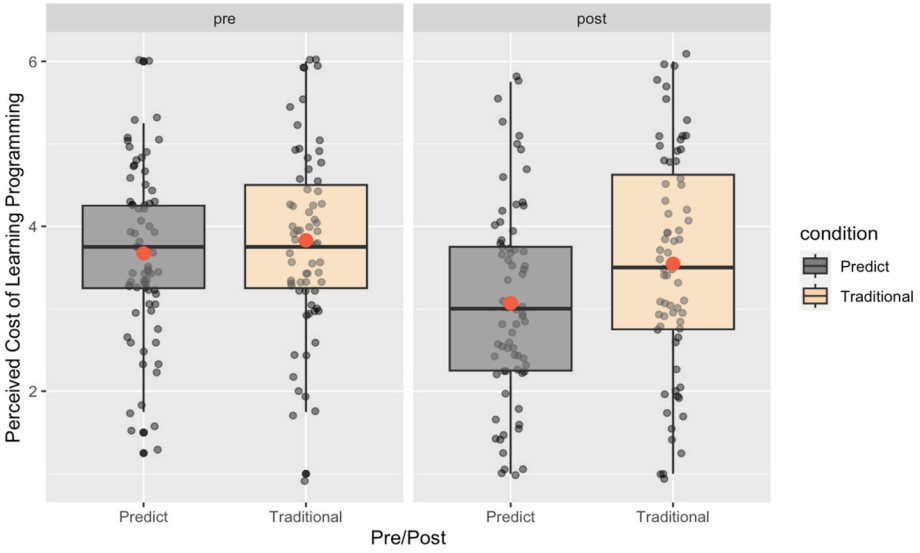


Fig. 7. Distribution of Cost Ratings on the Pre-Survey and Post-Survey Broken Down By Condition
Note. Points represent individual participants' scores. Horizontal lines represent group medians. The red dot represents the mean. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

Table 5
Overall correlation matrix.

Variable	1	2	3	4	5	6	7	8
1. Post-test score	–							
2. Flexibility	.66***	–						
3. R Sentiment avg	.51***	.43***	–					
4. Response to error	.21*	.20*	.32***	–				
5. Intrinsic Load	−.50***	−.46***	−.21*	−.19*	–			
6. Extraeneous Load	−.43***	−.35***	−.46***	−.29***	.45***	–		
7. Germane Load	.47***	.37***	.56***	.38***	−.17	−.38***	–	
8. Pre-test cost	−.15	−.25*	−.27	−.03	.43***	.26*	−.10	–
9. Post-test cost	−.23*	−.29	−.44***	−.08	.30***	.36***	−.22*	.70***

*<0.05, **<0.01, ***<0.001.

emotional responses to R and to error messages. Though the two groups did not differ in their sentiment expressed at the start of the activity, students in the Predict condition demonstrated more positive sentiment than students in the Traditional condition as the lesson progressed. Importantly, the differences between the two groups first emerged as the task difficulty increased, suggesting that the POE pedagogy may help buffer against negative emotions associated with challenges or setbacks. This hypothesis is further supported by the finding that students assigned to the Predict condition exhibited more positive responses when shown a hypothetical programming error on the post-survey. These observed trends are in line with previous studies that found a link between use of the POE strategy and positive attitudes towards learning (Bilen & Aydoğdu, 2010; Köse et al., 2003; Liew, 1995).

These findings suggest that the prediction activities may prepare students more effectively for the cycles of trial and error inherent in coding. This may be critical because students' conceptions of programming can influence their motivation and learning strategies. For instance, in a sample of 421 Taiwanese students majoring in computer science, Liang et al. (2015) found that students who conceived of programming as "memorization" and "rote learning" showed more surface-level motivation and approaches to learning. It may be more adaptive for students to conceptualize coding as cycles of prediction-observation-explanation, iteratively updating their knowledge and code.

Students assigned to the Predict condition rated their perceived extraneous cognitive load to be lower and the germane load to be higher than students assigned to the Traditional condition. They also considered the learning activity to be less costly than students assigned to produce their own code, even though the amount of time and effort (as measured by word count and time spent) did not meaningfully differ across the two conditions. The prediction tasks may have directed students' attention to selected features of the code making the learning experience feel more manageable and less costly, thus leading to more positive sentiment towards R. In general, pedagogies that avoid overwhelming students with multiple simultaneous tasks may lead to more positive attitudes towards the subject (Brod, 2018; Craik & Lockhart, 1972; Prat-Sala & Redford, 2010). Even though the code writing exercises in the Traditional condition were fairly simple (students only had to write one or two lines of code), some aspects of writing and submitting code must have felt more cognitively burdensome than making predictions. This load and cost may have increased negative emotions towards R (such as frustration) during learning.

Existing research has also found the POE pedagogy to result in a range of favorable non-cognitive outcomes beyond our specific focus. For example, studies have reported increased interest in academic subjects as an outcome of POE implementation (Bilen & Aydoğdu, 2010; Köse et al., 2003; Liew, 1995). Additionally, the application of POE pedagogy has been associated with heightened self-confidence among learners (Bilen, 2009; Kırılmazkaya & Zengin-Kırbağ, 2015) and the cultivation of a growth mindset specifically directed toward learning within the relevant domain (Güngör & Özkan, 2016, December). These findings underscore the broader positive impacts of POE pedagogy on various non-cognitive aspects of students' educational experiences.

4.3. Interaction between cognitive and non-cognitive outcomes

Although we have considered cognitive and non-cognitive outcomes separately, the processes leading to each are intimately intertwined during learning. It is not clear whether non-cognitive (i.e., emotional and motivational) factors affect learning, or if learning affects non-cognitive experiences. It is also possible that both are true.

On one hand, more positive non-cognitive experiences may lead to better cognitive outcomes. For example, generating predictions could elicit responses such as curiosity and interest that can facilitate knowledge acquisition and creativity (Isen, 2000). Prior research has shown that emotions like curiosity can lead to greater exploration and engagement (Kosuliev & Stanev, 2020), which could lead to deeper and more connected understanding. Positive emotions during learning can also orient students' attention to feedback (Boekaerts, 2010, pp. 91–111; Gervey et al., 2005). For example, making a prediction might have made students feel more invested in the output of the R code.

Psychological cost – students' perceptions of how costly an activity is in terms of the time it takes, how stressful it is, and how much it takes away from participation in other valued activities) (Barron & Hulleman, 2015) – can also impact engagement and future investment in learning. If students experience learning to program as costly, they may take maladaptive approaches to learning (e.g., getting it over with) and decide not to invest in further learning (e.g., learning on their own, taking more courses, pursuing related majors). Thus, having a more positive emotional and motivational experience of programming instruction might lead students to learn more about programming.

On the other hand, more effective cognitive experiences (that is, better learning) might create better non-cognitive experiences. Implicit in each learning task is a cognitive goal. In the Traditional condition, the goal is to write code that accomplishes some task. Novices engaged in this task might assume that "learning to code" is learning to write flawless code that works on the first try. They might misinterpret code not running as a sign of failure or evidence that they are not good at programming. When students possess a rigid perspective that learning to code should yield immediate results without uncertainty or a need for iterative evaluation (Lee et al., 2023), the inability to produce flawless code may be more discouraging than it should be. In contrast, making incorrect predictions on the prediction task might help novices develop more adaptive epistemic beliefs about what it means to learn to code. Although students' epistemic beliefs about computer programming can predict their self-efficacy in coding (e.g., Lee et al., 2023), future research should be done to determine whether changing these beliefs does indeed impact learning and interest.

Another way that the cognitive outcomes impact non-cognitive ones is that repeated experiences of failure to write code correctly can lead to a chain of negative emotions such as frustration, disengagement, and boredom (D'Mello & Graesser, 2011). These emotions can set off "vicious cycles" that depress learning, which further heightens negative emotions (D'Mello & Graesser, 2011, p. 15).

4.4. Limitations and future directions

This study provides initial evidence of the potential of using predictions as a strategy to improve learning among students in the early stages of learning programming. However, there are a number of limitations that should be considered when interpreting these results.

4.4.1. Participants and study context

Participants were recruited from the Psychology department at a competitive public university, and thus may not be representative of the broader population of novice programming students. We only investigated the effect of predicting versus traditional instruction using the R programming language. Prior research indicates that some programming languages may be easier to learn than others. Because we only looked at one programming language, it's not clear whether or not predicting might similarly benefit learners in the early stages of learning another, more complex language.

Another limitation is that the design of this study did not include a pretest, which limits our ability to capture participants' prior programming knowledge at baseline. We made this decision because we feared that taking a pre-test might impact students' psychological state as they engaged in the learning sessions and thus alter the effect of our experimental manipulation (Opfer & Thompson, 2008). While the study asked participants to self-identify as novice learners of R programming who have not taken any programming courses, it is possible that they might have possessed some other prior experience or knowledge (e.g., mathematical prowess, having taken a logic course), which could have influenced their performance on the posttest.

Although this study provides initial evidence that predicting may influence learning in a controlled experimental setting with self-proclaimed beginners, it does not provide information on whether these findings extend to naturalistic learning contexts. We focused primarily on very early stages of learning programming. However, mastering computer programming takes weeks, months, or years. And most people encounter programming not as a single 1-h session, but as a longer series of sessions and courses). It will be important to study the effects of predicting in learning higher-order programming concepts, in more realistic time spans, and with more developed students. For example, it would be interesting to see whether varying programming instruction in the first few weeks of a course would benefit students as the course progresses over weeks.

4.4.2. Measurement of outcomes

Another important limitation of this study is the measurement of student outcomes. Our posttest involved a range of problem types, such as identifying and correcting code errors, explaining the purpose and function of code snippets in English, selecting code that would achieve a given purpose, and writing correct code to achieve a goal. Even with all of these question types, the post-test does not reflect the true diversity of skills necessary for coding. It will be important in future studies to examine the effect of prediction pedagogy on other measures that have been used in the computer science education literature (e.g., code tracing, parsons problems).

For example, Schulte (2008) proposed three dimensions that should be addressed when teaching programming: understanding structural aspects, such as text surface structure; program execution (e.g., data and control flow); and functional aspects, i.e., understanding what the code does. Future research could benefit from drawing on Schulte's framework to guide a more holistic assessment of students' programming abilities. Moreover, future research should conduct a systematic and formal analysis of errors. This approach would enhance our understanding of the specific nature and patterns of errors, providing valuable insights for refining instructional methodologies and optimizing learning outcomes.

Additionally, because measures of learning included problems that were relatively similar to those used during instruction, it is not clear whether predicting can produce far transfer – improved performance on tasks that share fewer surface-level similarities with the content covered in the lesson. If predicting really does prepare students to engage in more realistic programming practices (e.g., trial and error), it might also prepare them for future learning (Schwartz & Martin, 2004). Future research might consider providing participants with initial instruction using either predicting or traditional methods, then examining their persistence and success in learning novel programming functions on their own.

Measures of non-cognitive outcomes relied on self-report ratings. It is possible that asking students to stop and evaluate their sentiments during learning can disrupt typical learning processes. Less intrusive and more direct measures of emotions such as physiological measures or automated affect detection present a promising alternative to self-reported ratings and would be valuable to include in future research.

4.4.3. Mechanisms for the benefits of prediction

Although this study did not attempt to explain *how* predicting influences students' cognitive and non-cognitive outcomes, it does provide a basis for future studies of these mechanisms. For example, one conjecture is that prediction benefits learners by directing their attention to specific aspects of the task. The materials in this study were carefully designed to highlight specific features and concepts. Would students experience similar benefits if asked to make a more general prediction that would not direct their attention to key features? We are currently conducting a follow-up study where students are asked to make open-ended predictions about the code rather than specific predictions (e.g., through a multiple-choice format). Our hypothesis is that if attention direction is the key mechanism behind prediction, students making open-ended predictions would not benefit as much as those making specific predictions.

4.5. Learning programming in the age of AI

Furthermore, it is crucial to acknowledge the evolving landscape of instructional tools in the field of programming education. One of the recent advancements is Chat-GPT, an artificial intelligence model that can generate code for learners. However, the mere availability of this tool does not automatically suggest its superiority as an educational resource. While these models may offer valuable support to novice students with limited programming experience, it remains uncertain whether they represent the most optimal approach to learning programming. Effective programming education entails more than just providing code; it requires a solid comprehension of coding fundamentals. Students must be able to interpret the code generated by Chat-GPT, assess its relevance to their programming goals, and determine its suitability in specific contexts. Future research should delve deeper into the implications of integrating AI-generated code within programming education, shedding light on its potential impacts on students' learning experiences and programming proficiency.

5. Conclusion

In this randomized experiment, we found evidence that generating predictions can lead to more positive emotional experiences, increased motivation, and better learning outcomes among beginning students learning computer programming, compared to modifying or writing code. The findings of this study raise questions about the effectiveness of commonly used instructional strategies for teaching programming, particularly for novice programmers in the early stages of learning. While traditional "tell-and-practice" methods can be effective, educators may benefit from incorporating other approaches, such as prediction, into their instructional designs.

The findings of this study not only have implications for computer programming education but also provide insight into how learning tasks can impact multiple processes involved in learning from cognitive to

affective to motivational. Given the importance of these interacting processes in learning any complex skill or knowledge, it is crucial to consider how different instructional approaches impact these processes simultaneously when designing instruction.

CRediT authorship contribution statement

Mary C. Tucker: Conceptualization, Methodology, Software, Investigation, Data curation, Visualization, Writing – original draft, Project administration. **Xinran (Wendy) Wang:** Investigation, Data curation, Writing – review & editing, Visualization, Project administration. **Ji Y. Son:** Writing – review & editing, Supervision. **James W. Stigler:** Writing – review & editing, Supervision.

Declaration of competing interest

We have no conflict of interest to disclose.

Acknowledgements

This article is adapted from a doctoral dissertation submitted to the UCLA Psychology Department by the first author (Tucker, 2022). We gratefully acknowledge the support of the Chan Zuckerberg Initiative DAF, an advised fund of Silicon Valley Community Foundation (DRL-1229004) and the California Governor's Office of Planning and Research (contract OPR18115).

Appendix A. Supplementary data

Supplementary data to this article can be found online at https://doi.org/10.1016/j.learninstruc.2023.101871.

References

Altadmri, A., & Brown, N. C. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM technical Symposium on computer science education* (pp. 522–527). https://doi.org/10.1145/2676723.2677258

Barron, K. E., & Hulleman, C. S. (2015). Expectancy-value-cost model of motivation. *Psychology, 84*, 261–271. https://doi.org/10.1016/B978-0-08-097086-8.26099-6

Bayman, P., & Mayer, R. E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM, 26*(9), 677–679. https://doi.org/10.1145/358172.358408

Biggs, J. B., & Collis, K. F. (2014). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

Bilen, K., & Aydoğdu, M. (2010). The use of POE (guess-by-eye-explain) strategy in teaching the concepts of photosynthesis and respiration in plants. *Journal of Social Sciences Institute, 7*(14), 179–194.

Boekaerts, M. (2010). The crucial role of motivation and emotion in classroom learning. *The nature of learning: Using research to inspire practice*.

Bosch, N., Chen, Y., & D'Mello, S. (2014). It's written on your face: Detecting affective states from facial expressions while learning computer programming. In S. Trausan-Matu, K. E. Boyer, M. Crosby, & K. Panourgia (Eds.), *Intelligent tutoring systems* (pp. 39–44). Springer International Publishing. https://doi.org/10.1007/978-3-319-07221-0_5.

Brod, G. (2021). Predicting as a learning strategy. *Psychonomic Bulletin & Review, 28*(6), 1839–1847. https://doi.org/10.3758/s13423-021-01904-1

Brod, G., Hasselhorn, M., & Bunge, S. A. (2018). When generating a prediction boosts learning: The element of surprise. *Learning and Instruction, 55*, 22–31. https://doi.org/10.1016/j.learninstruc.2018.01.013

Bruner, J. S. (1966). *Toward a theory of instruction*. Harvard University Press.

Cañas, J. J., Bajo, M. T., & Gonzalvo, P. (1994). Mental models and computer programming. *International Journal of Human-Computer Studies, 40*(5), 795–811. https://doi.org/10.1006/ijhc.1994.1038

Clancy, M. J., & Linn, M. C. (1999). Patterns and pedagogy. *ACM SIGCSE Bulletin, 31*(1), 37–42. https://doi.org/10.1145/384266.299673

Corney, M., Lister, R., & Teague, D. (2011). Early relational reasoning and the novice programmer: Swapping as the'hello world'of relational reasoning. In *Proceedings of the thirteenth australiasian computing education conference* (pp. 95–104). Australian Computer Society.

Craik, F. I., & Lockhart, R. S. (1972). Levels of processing: A framework for memory research. *Journal of Verbal Learning and Verbal Behavior, 11*(6), 671–684. https://doi.org/10.1016/S0022-5371(72)80001-X

D'Mello, S., & Graesser, A. (2011). The half-life of cognitive-affective states during complex learning. *Cognition & Emotion, 25*(7), 1299–1308. https://doi.org/10.1080/02699931.2011.613668

D'Mello, S., & Graesser, A. (2012). Dynamics of affective states during complex learning. *Learning and Instruction, 22*(2), 145–157. https://doi.org/10.1016/j.learninstruc.2011.10.001

Dweck, C. (1999). Mindset: The new Psychology of success (ballentine, New York, 2006). In *Self-theories: Their role in motivation, personality, and development*. Psychology press.

Flake, J. K., Barron, K. E., Hulleman, C., McCoach, B. D., & Welsh, M. E. (2015). Measuring cost: The forgotten component of expectancy-value theory. *Contemporary Educational Psychology, 41*, 232–244. https://doi.org/10.1016/j.cedpsych.2015.03.002

Gervey, B., Igou, E. R., & Trope, Y. (2005). Positive mood and future-oriented self-evaluation. *Motivation and Emotion, 29*, 267–294. https://doi.org/10.1007/s11031-006-9011-3

Große, C. S., & Renkl, A. (2007). Finding and fixing errors in worked examples: Can this foster learning outcomes? *Learning and Instruction, 17*(6), 612–634. https://doi.org/10.1016/j.learninstruc.2007.09.008

Guilford, J. P. (1967). Creativity: Yesterday, today and tomorrow. *Journal of Creative Behavior, 1*(1), 3–14. https://doi.org/10.1002/j.2162-6057.1967.tb00002.x

Güngör, S. N., & Özkan, M. (2016). Teaching enzymes to pre-service science teachers through POE (predict, observe, explain) method: The case of catalase. In *Asia-pacific forum on science learning & teaching* (Vol. 17), 2.

Hong, J. C., Hsiao, H. S., Chen, P. H., Lu, C. C., Tai, K. H., & Tsai, C. R. (2021). Critical attitude and ability associated with students' self-confidence and attitude toward "predict-observe-explain" online science inquiry learning. *Computers & Education, 166*, Article 104172. https://doi.org/10.1016/j.compedu.2021.104172

Isen, A. M. (2000). Some perspectives on positive affect and self-regulation. *Psychological Inquiry, 11*(3), 184–187. https://www.jstor.org/stable/1449800.

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 107–111). https://doi.org/10.1145/1734263.1734299

Kırılmazkaya, G., & Zengin-Kırbağ, F. (2015). Investigation of the effect of guess-observe-explain method on secondary school students' academic achievement and attitudes towards science. *International Journal of Social Studies, 8*(41), 975–981.

Köse, S., Coştu, B., & Keser, Ö. F. (2003). Identifying misconceptions in science subjects: POE method and sample activities. *Journal of PAU Education Faculty, 13*(1), 43–53.

Kosuliev, A., & Stanev, E. (2020). Betting on answers as a way of engaging STUDENTS1. In *59th annual scientific conference - University of Ruse and Union of scientists* (pp. 127–131). Bulgaria, 2020.

Kwon, O. N., Park, J. H., & Park, J. S. (2006). Cultivating divergent thinking in mathematics through an open-ended approach. *Asia Pacific Education Review, 7*(1), 51–61. https://doi.org/10.1007/BF03036784

Larson, R. W., & Richards, M. H. (1991). Boredom in the middle school years: Blaming schools versus blaming students. *American Journal of Education, 99*(4), 418–443. https://www.jstor.org/stable/1085554.

Lee, S. W. Y., Liang, J. C., Hsu, C. Y., & Tsai, M. J. (2023). Students' beliefs about computer programming predict their computational thinking and computer programming self-efficacy. *Interactive Learning Environments*, 1–21. https://doi.org/10.1080/10494820.2023.2194929

Liang, J. C., Su, Y. C., & Tsai, C. C. (2015). The assessment of Taiwanese college students' conceptions of and approaches to learning computer science and their relationships. *The Asia-Pacific Education Researcher, 24*, 557–567. https://doi.org/10.1007/s40299-014-0201-6

Liew, C. W. (1995). A Predict-Observe-Explain teaching sequence for learning about students' understanding of heat. *Australian Science Teachers Journal, 41*(1), 68–72.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin, 38*(3), 118–122. https://doi.org/10.1145/1140123.1140157

Little, J., & Bjork, E. (2011). Pretesting with multiple-choice questions facilitates learning. In *Proceedings of the annual meeting of the cognitive science society* (Vol. 33), 33 https://escholarship.org/uc/item/9xn3f39q.

Little, J. L., & Bjork, E. L. (2016). Multiple-choice pretesting potentiates learning of related information. *Memory & Cognition, 44*, 1085–1101. https://doi.org/10.3758/s13421-016-0621-z

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. *Proceedings of the Fourth International Workshop on Computing Education Research*, 101–112. https://doi.org/10.1145/1404520.1404531

Ma, L. (2007). *Investigating and improving novice programmers' mental models of programming concepts*. Doctoral dissertation, University of Strathclyde.

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2007). Investigating the viability of mental models held by novice programmers. In *Proceedings of the 38th SIGCSE technical symposium on computer science education* (pp. 499–503). https://doi.org/10.1145/1227310.1227481

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE on innovation and technology in computer science education* (pp. 125–180). https://doi.org/10.1145/572133.572137

Miller, K., Lasry, N., Chu, K., & Mazur, E. (2013). Role of physics lecture demonstrations in conceptual learning. *Physical Review Special Topics - Physics Education Research, 9*(2), Article 020113. https://doi.org/10.1103/PhysRevSTPER.9.020113

Mirman, D. (2014). Growth curve analysis: A hands-on tutorial on using multilevel regression to analyze time course data. In *Proceedings of the annual meeting of the cognitive science society* (Vol. 36), 36). Retrieved from https://escholarship.org/uc/item/1dp5q4k2.

Morrison, B. B., Dorn, B., & Guzdial, M. (2014). Measuring cognitive load in introductory CS: Adaptation of an instrument. *Proceedings of the Tenth Annual Conference on International Computing Education Research*, 131–138. https://doi.org/10.1145/2632320.2632348

Mueller, C. M., & Dweck, C. S. (1998). Praise for intelligence can undermine children's motivation and performance. *Journal of Personality and Social Psychology, 75*(1), 33. https://psycnet.apa.org/doi/10.1037/0022-3514.75.1.33.

Munezero, M., Montero, C. S., Sutinen, E., & Pajunen, J. (2014). Are they different? Affect, feeling, emotion, sentiment, and opinion detection in text. *IEEE Transactions on Affective Computing, 5*(2), 101–111. https://doi.org/10.1109/TAFFC.2014.2317187

Nadiah, N., Salleh, S., & Laxman, K. (2021). THE impact of VIDEO-BASED predict-observe-explain (POE) on secondary school students' scientific literacy. *International Journal on E-Learning, 20*(3), 295–321. https://www.learntechlib.org/primary/p/219040/.

Opfer, J. E., & Thompson, C. A. (2008). The trouble with transfer: Insights from microgenetic changes in the representation of numerical magnitude. *Child Development, 79*(3), 788–804. https://doi.org/10.1111/j.1467-8624.2008.01158.x

Pan, S. C., & Carpenter, S. K. (2023). Prequestioning and pretesting effects: A review of empirical research, theoretical perspectives, and implications for educational practice. *Educational Psychology Review, 35*(4), 97. https://doi.org/10.1007/s10648-023-09814-5

Prat-Sala, M., & Redford, P. (2010). The interplay between motivation, self-efficacy, and approaches to studying. *British Journal of Educational Psychology, 80*(2), 283–305. https://doi.org/10.1348/000709909X480563

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education, 18*(1). https://doi.org/10.1145/3077618, 1:1-1:24.

Renumol, V. G., Janakiram, D., & Jayaprakash, S. (2010). Identification of cognitive processes of effective and ineffective students during computer programming. *ACM Transactions on Computing Education, 10*(3), 1–21. https://doi.org/10.1145/1821996.1821998

Robinson, W. P. (1975). Boredom at school. *British Journal of Educational Psychology, 45*(2), 141–152.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

Sajaniemi, J., & Navarro-Prieto, R. (2005). Roles of Variables in Experts' Programming KnowledgeP. Romero, J. Good, E. Acosta Chaparro, & S. Bryant (Eds.). *Proc. PPIG, 17*, 145–159.

Schnotz, W., & Kürschner, C. (2007). A reconsideration of cognitive load theory. *Educational Psychology Review, 19*, 469–508. https://doi.org/10.1007/s10648-007-9053-4

Schulte, C. (2008). Block model: An educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the fourth international workshop on computing education research* (pp. 149–160).

Schwartz, D. L., & Bransford, J. D. (1998). A time for telling. *Cognition and Instruction, 16*(4), 475–5223. https://doi.org/10.1207/s1532690xci1604_4

Schwartz, D. L., Chase, C. C., Oppezzo, M. A., & Chin, D. B. (2011). Practicing versus inventing with contrasting cases: The effects of telling first on learning and transfer. *Journal of Educational Psychology, 103*(4), 759. https://psycnet.apa.org/doi/10.1037/a0025140.

Schwartz, D. L., & Martin, T. (2004). Inventing to prepare for future learning: The hidden efficiency of encouraging original student production in statistics instruction. *Cognition and Instruction, 22*(2), 129–184. https://doi.org/10.1207/s1532690xci2202_1

Sirkiä, T., & Sorva, J. (2012). Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th koli calling international Conference on computing education research* (pp. 19–28). https://doi.org/10.1145/2401796.2401799

Son, J. Y., Smith, L. B., & Goldstone, R. L. (2008). Simplicity and generalization: Short-cutting abstraction in children's object categorizations. *Cognition, 108*(3), 626–638. https://doi.org/10.1016/j.cognition.2008.05.002

Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction, 4*(4), 295–312. https://doi.org/10.1016/0959-4752(94)90003-5

Sweller, J. (2010a). *Cognitive load theory: Recent theoretical advances*. https://doi.org/10.1017/CBO9780511844744.004

Sweller, J. (2010b). Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review, 22*(2), 123–138. https://doi.org/10.1007/s10648-010-9128-5

Sweller, J., & Cooper, G. A. (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction, 2*(1), 59–89. https://doi.org/10.1207/s1532690xci0201_3

Sweller, J., van Merriënboer, J. J., & Paas, F. (2019). Cognitive architecture and instructional design: 20 years later. *Educational Psychology Review, 31*, 261–292. https://doi.org/10.1007/s10648-019-09465-5

Theobald, M., & Brod, G. (2021). Tackling scientific misconceptions: The element of surprise. *Child Development, 92*(5), 2128–2141. https://doi.org/10.1111/cdev.13582

Tucker, M. C. (2022). *Prediction versus production for teaching computer programming* (Publication No. 29393682.) [Doctoral Dissertation. Los Angeles: University of California. ProQuest Disserations and Theses database.

Venables, A., Tan, G., & Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, 117–128. https://doi.org/10.1145/1584322.1584336

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Ajith Kumar, P. K., & Prasad, C. (2006). *An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies*. https://opus.lib.uts.edu.au/handle/10453/5050.

White, R. T., & Gunstone, R. F. (1992). *Probing understanding*. London: The Falmer Press.

Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., … Ko, A. J. (2019). A theory of instruction for introductory programming skills. *Computer Science Education, 29*(2–3), 205–253.

Zhu, X., & Simon, H. A. (1987). Learning mathematics from examples and by doing. *Cognition and Instruction, 4*(3), 137–166. https://doi.org/10.1207/s1532690xci0403_1