9-2025

Automatic generation of introductory programming exercises with large language models

Nguyen Binh Duong TA
Singapore Management University, donta@smu.edu.sg

Hua Gia Phuc NGUYEN
Singapore Management University, hgpnguyen.2019@phdcs.smu.edu.sg

GOTTIPATI Swapna
Singapore Management University, SWAPNAG@smu.edu.sg

Citation

Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models

Sami Sarsa
Aalto University
Espoo, Finland
sami.sarsa@aalto.fi

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

Paul Denny
The University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

ABSTRACT

This article explores the natural language generation capabilities of large language models with application to the production of two types of learning resources common in programming courses. Using OpenAI Codex as the large language model, we create programming exercises (including sample solutions and test cases) and code explanations, assessing these qualitatively and quantitatively. Our results suggest that the majority of the automatically generated content is both novel and sensible, and in some cases ready to use as is. When creating exercises we find that it is remarkably easy to influence both the programming concepts and the contextual themes they contain, simply by supplying keywords as input to the model. Our analysis suggests that there is significant value in massive generative machine learning models as a tool for instructors, although there remains a need for some oversight to ensure the quality of the generated content before it is delivered to students. We further discuss the implications of OpenAI Codex and similar tools for introductory programming education and highlight future research streams that have the potential to improve the quality of the educational experience for both teachers and students alike.

CCS CONCEPTS

• **Social and professional topics** → *Computing education*; • **Computing methodologies** → *Natural language generation*.

KEYWORDS

Natural language generation, OpenAI Codex, GPT-3, CS1, Programming exercises, Code explanations, Robosourcing, Exercise generation, Resource generation, Automated feedback, Large language models

1 INTRODUCTION

Creating an introductory programming course involves designing course materials, developing assignments, creating feedback opportunities, and planning out the flow of the course [25]. Providing opportunities for active learning – that involves both doing and reflecting – is especially important for learning programming [72]. One particular approach which has gained popularity due to the wide availability of auto-grading tools that can generate immediate feedback is the use of many short programming exercises which students use to develop mastery through regular practice [4, 21, 85]. Such approaches have become so popular that Finnie-Ansley et al. suggest they may form a signature pedagogy within computing education [27, 76].

Developing suitable programming exercises involves multiple facets, including creating problem statements, sample code solutions and automated tests. Creating a sufficient quantity of novel exercises to form a useful resource is a significant challenge for educators, and particularly difficult in an age where solutions to existing exercises are quickly published and shared online. Indeed, writing good questions and good tests to verify them is a fundamental challenge [53, 91]. It is thus not surprising that researchers have explored collaborative approaches to resource generation, such as crowdsourcing, but these are not free from problems as they suffer from limitations around student motivation and content quality [15, 65, 71].

Let us consider a possible problem statement for a programming exercise that could appear early in a typical CS2 course focused on object-oriented programming [67]. In this example, the language is Python and the goal is to familiarize students with the basic concepts of attributes, methods and method calling, as well as to practice manipulating built-in dictionaries.

```
Write a class called Fisherman that is initialized
    ↪  with a dictionary of fish, e.g. {"Salmon
    ↪ ": 10, "Trout": 20, "Tuna": 5}. The class
    ↪ should have a method called catch, which
    ↪ takes in one parameter, a fish name, and
    ↪ returns the number of that fish caught.
    ↪ The class should also have a method called
    ↪  throw_away, which takes in a fish name,
    ↪ and decreases the number of that fish in
    ↪ the dictionary by 1.

As an example, the code:
fisherman = Fisherman({"Salmon": 10, "Trout": 20,
    ↪ "Tuna": 5})
fisherman.catch("Salmon")
fisherman.throw_away("Trout")
fisherman.catch("Tuna")
print(fisherman.fish)

should print out:
{"Salmon": 11, "Trout": 19, "Tuna": 6}
```

A student that chooses to work on this programming exercise might produce a solution similar to the following.

```
class Fisherman():
  def __init__(self, fish):
    self.fish = fish

  def catch(self, fish):
    if fish in self.fish:
      self.fish[fish] += 1
    else:
      self.fish[fish] = 1

  def throw_away(self, fish):
    if fish in self.fish:
      self.fish[fish] -= 1
      if self.fish[fish] == 0:
        del self.fish[fish]
```

Once the student submits their solution, they would typically receive some kind of feedback from the auto-grader. If there is an error in their code, the student may attempt a line-by-line walk-through of their solution to help them locate a bug. If they are stuck, they may reach out to a teaching assistant for help, in which case the assistant may ask the student to explain their code and jointly walk through it with them step by step. This type of communication, similar to rubber-duck debugging, can be an effective way of exploring one's own understanding [64]. If the student needs a great deal of help, the teaching assistant may even help generate an explanation for them. A stepwise explanation of the above program might be as follows.

(1) We create a dictionary called fish where we'll store our fishes and their quantity.
(2) We create a class called Fisherman.
(3) We create a constructor for the class Fisherman where we initialize the fish dictionary.
(4) We create a method called catch which takes a fish as a parameter.

(5) If the fish is already in the fish dictionary, we increment its value by 1.
(6) If the fish is not in the fish dictionary, we add it and set its value to 1.
(7) We create a method called throw_away which takes a fish as a parameter.
(8) If the fish is in the fish dictionary, we decrement its value by 1.
(9) If the fish's value reaches 0, we delete the fish from the dictionary.

Generating these types of explanations and jointly helping students trace through their own code can take a considerable amount of time and effort for teachers. This effort grows proportionally with the cohort size, as does the challenge of developing repositories of practice exercises that suit the varied learning needs of larger and more diverse cohorts. This makes the idea of automatically generating these kinds of materials an exciting prospect. As a matter of fact, *the above programming exercise, its solution, and the code explanation were all generated automatically by OpenAI Codex*, which is a generative NLP model for creating code and code related texts. In addition to the exercise, solution and code explanation, OpenAI Codex also generated a suite of test cases for the exercise that could be used to automatically verify attempted solutions. We discuss the generation of test suites later in the paper. Given the known difficulties around maintaining the integrity of existing question banks [3], it is notable that the problem description itself is, as far as we can tell, entirely novel. At the time of writing, the problem description returns no relevant matches (using any combination of sentence fragments from the description) on search engines like Google or on websites like Chegg or StackOverflow that are frequently used by students to find solutions from problems statements.

Very recent work by Finnie-Ansley et al. has explored the implications of OpenAI Codex on programming education, but from the perspective of assessing the accuracy of source code generated by the model to solve typical CS1 test and exam questions [27]. This prior study focused primarily on the challenges that the technology poses to educators, including serious academic integrity issues, over-reliance by novices, and confusion caused by the generation of incorrect code or code with poor style. In this work we explore the opportunities that are provided by this new technology. Instead of focusing on the generation of source code, which is the most publicized functionality of OpenAI Codex, we primarily investigate the generation of natural language artefacts – both programming exercises and explanations of code – that may offer value to both instructors and students.

RQ1 To what extent are programming exercises created using OpenAI Codex sensible, novel, and readily applicable?
RQ2 How comprehensive and accurate are OpenAI Codex natural language explanations of code solutions to introductory programming exercises?

Our work provides insight into the utility of OpenAI Codex as one part of the toolbox of a teacher of an introductory programming course and discusses the further potential of such tools. In this work, we focus on the applicability of OpenAI Codex for the generation of programming exercises and for creating feedback
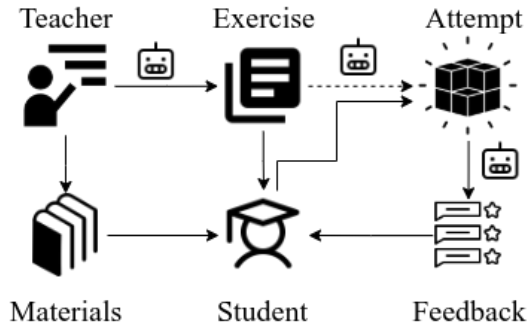
Figure 1: Lifecycle of a programming exercise. Teacher creates programming exercises and learning materials. Students study the materials and the exercises, and create exercise attempts. Students receive feedback on their attempts. In this work, denoted by the solid arrows with a robot, we explore the use of OpenAI Codex for the creation of programming exercises and for providing feedback on students' programming exercise attempts. The dashed arrow with a robot represents prior work by Finnie-Ansley et al. [27] who explored how well Codex can solve introductory programming exercises.

from student attempts to programming exercises. Figure 1, which illustrates a simplistic lifecycle for a programming exercise, provides some context for our contributions.

2 BACKGROUND

2.1 Practice and Feedback in Introductory Programming Courses

Introductory programming courses around the world interleave theory and practice, providing students opportunities for learning how to write programs guided by exercise statements, automated assessment systems, and course staff. Courses and course assignments are typically written so that they are increasingly complex, gradually introduce new concepts and seek to avoid overwhelming students, i.e. seek to avoid cognitive overload [20]. Such a design can be seen as scaffolding that supports students in their zone of proximal development [87], that is, their area of skills and knowledge where they cannot yet succeed on their own, but where they can succeed with guidance. As a student learns, the student's zone of proximal development also changes.

The design of programming courses and course assignments often reflects the idea of deliberate practice [23], which is a systematic and purposeful type of practice that focuses on improvement of performance in a specific task. Continuing deliberate practice is sustained with grit [19], i.e. passion and perseverance for long-term goals, even when pursuing those goals feels difficult. Motivation towards assignments is influenced by the design of assignments; while very easy assignments have a high expectancy for success, their utility value is low, and as per expectancy-value theory [69], students may have little motivation to work with them. Conversely, assignments that are too difficult also have little utility and can lead to low motivation [69], and likely contribute negatively towards

feelings of self-efficacy [6]. One practice in teaching introductory programming courses that has sought to avoid students prematurely encountering assignments that are too complex is the use of many small programming exercises which help develop mastery through regular practice [4, 21, 85]. As students have different backgrounds, different skills, and differently evolving zones of proximal development, each student would likely benefit from a tailored set of assignments, maybe even with contextual cues tuned to their own interests. This latter point is supported by prior work in computing education suggesting that students' familiarity with the context of a problem can potentially be helpful [44]. However, such large exercise pools would be very tedious to develop [53, 91].

In addition to creating programming assignments, teachers often design feedback opportunities to course assignments. One common approach for providing feedback in introductory programming courses is the use of automated assessment systems [2, 35, 60], which at the minimum provide feedback on the correctness of programming assignments submitted for evaluation. As feedback plays a considerable role in learning [32], in addition to influencing approaches to learning by simply being offered [86], it should be given with care; feedback can both improve self-efficacy and decrease self-efficacy [32]. In general, formative feedback – feedback given as a part of the learning process – is preferred over summative feedback, i.e. feedback given after the learning process [40, 77]. In particular, formative feedback can be used to aid self-regulated learning and metacognition, helping students in becoming better learners [77].

Classroom practices and the way that programming instruction is organized also matters [84]. In particular, feedback opportunities can be included into classroom and lab sessions. For example, both peer instruction [13] and pair programming [90] create opportunities for reflection. In peer instruction, the reflection is partially guided by the teacher responsible for the peer instruction questions, while in pair programming, students interact and reflect on the program that is being worked on. Students tend to enjoy pair programming [1] and also learn to reason and explain code.

2.2 Code Explanations and Their Assessment

The ability to reason about code and explain its purpose is a key skill that novices develop as they gain expertise [56]. The relationship between a student's ability to explain code, and related skills such as code tracing and code writing, have been the focus of much prior research in computing education [51, 52, 75, 83]. The evidence from this body of work generally suggests that competence at explaining code develops after lower-level code tracing skills and before higher-level code writing skills. This hierarchy forms the basis of a recently proposed theory of programming instruction by Xie et al. in which learners first develop the ability to explain the purpose of reusable code templates before writing code to solve new problems [92].

Assessing both code tracing and code writing skills is generally straightforward as answers are objective and thus can be readily automated. Code tracing questions typically ask students to determine the output or the value stored in a variable after a provided code block is executed and are often presented in multiple-choice format [50]. Hassan and Zilles propose a novel 'reverse-tracing'

question format which is non-trivial even when students have access to a computer [31], and Lehtinen et al. explored automatically creating multiple choice questions from students' own code [43]. A variety of approaches and tools for helping students develop code tracing skills have also been reported [68, 93]. Code writing questions require students to produce code from a problem description. Many tools for automatically grading code writing questions have emerged as institutions shift away from paper-based exams [57, 81], and look to provide immediate feedback on programming assignments and tasks [7, 45, 82].

Code explanation skills are less straightforward to assess because explanations are usually given in natural language and can be provided at varying levels of abstraction. A popular method for evaluating code explanation ability is the 'explain in plain English' question format, where students are asked to explain the purpose of a provided code block. This type of question was first studied as part of the BRACElet project [88], where student responses were classified by researchers according to the first four levels of the SOLO taxonomy [8]. The highest of these levels, relational, characterized responses that described at a high-level of abstraction how the code would behave over all possible inputs. A classic example from the BRACElet work is the response "it checks to see if the array is sorted" for describing code that compares adjacent elements in an array within a loop [49]. The next highest level, multistructural, was used to classify responses that gave a line-by-line description of the code but failed to succinctly state its purpose. Subsequent research has established a strong correlation between code writing skills and the ability to construct responses to explain in plain English questions at the relational level [56]. More recently, approaches for grading explain in plain English questions on exams have been explored, including a validated rubric to inform manual grading [10] and an automated tool which exhibited similar accuracy to that of trained teaching assistants [29]. Both approaches, like the SOLO classification commonly used in research, differentiate between responses at an abstract level and those which are line-by-line descriptions of the code.

Clearly, the ability to explain the purpose of *correct* code at an abstract level is an important skill for novices to develop. However, when code is *incorrect* the situation is more complex. For code that contains bugs, attempts to describe its purpose at the relational level are premature and will likely not identify the errors. Indeed, Perkins et al. argue that the ability to read what a piece of code actually does, rather than what we think it might do on a quick first inspection, is an important debugging skill [63]. They describe line-by-line walkthroughs of code as 'close tracking', which mirrors to some extent the 'mental simulations' that Soloway argues should be taught explicitly to students [79]. Therefore it is possible that multistructural explanations of code, at a line-by-line level, may provide some benefit for the purposes of debugging. Given that code walkthroughs can be mentally demanding, being presented with an explanation of one's own code may reduce the cognitive demands associated with debugging [55] and evidence from other educational domains suggests that being presented with explanations produced by others can improve learning [89]. Within computing education, techniques like pair programming [30] and misconception-based peer feedback [39] provide some opportunities for walking through code with others, but are not always

feasible and may not be suitable for individual student assessments. Therefore, the automatic generation of code explanations, particularly for supporting student debugging, is an attractive idea and one which is made feasible with the introduction of tools like OpenAI Codex.

2.3 Machine Learning Models for Code Generation

Recently, there has been great progress on generative natural language models, such as OpenAI's GPT-3 [9], that are capable of generating text that can be hard to distinguish from text written by humans [9]. These are deep learning models and their performance relies on both a vast number of parameters for the models (175 billion in the case of GPT-3) as well as an extensive corpus of text for training (570GB of text for GPT-3). Codex [11], also by OpenAI, is a GPT-model similar to GPT-3 but has been fine-tuned using publicly available code from GitHub with the goal of translating natural language to source code and vice versa, and to generate or auto-complete source code given source code as input.

In addition to OpenAI's Codex, other generative machine learning models capable of generating natural language from source code and/or vice versa have been developed. One of the earliest such models is Microsoft's CodeBERT [26]. CodeBERT has been trained with natural language - programming language pairs and is capable of generating source code documentation automatically similar to Codex. Another recently presented model is DeepMind's AlphaCode [48], which is capable of performing on par with a median competitor when presented with problem prompts at the programming competition level.

These recent models have multiple applications. One that has been proposed is to help programmers fix insecure code. Pearce et al. [61] analyzed the performance of Codex and similar models for repairing source code containing security flaws and found that through providing a carefully constructed prompt for the model, they were able to patch security issues in programs in some cases. Another study by Pearce et al. [62] analyzed the possibility of utilizing Codex for reverse engineering. In their study, they provided Codex decompiled code and prompted Codex to explain the purpose of the code. Their results indicated that there is some potential in utilizing models such as Codex for reverse engineering as slightly over half of the questions authors asked were answered correctly. However, they suggest there is a need for ongoing work, and propose fine-tuning the model by providing it context-specific training data (in their case, decompiled source code).

Finally, relevant to the current paper, very recent work in the domain of mathematics has shown that large language models can successfully solve and generate new problems [17]. Interestingly, in order to solve the mathematics problems, the authors use Codex to generate code-based solutions from natural language prompts which are then executed in order to perform the calculations.

2.4 Potential of Codex in Computing Education

The most common use case for Codex is generating new code from either a provided code fragment or from a natural language description of a problem. GitHub Copilot, which is an editor plug-in that is powered by OpenAI Codex, promises to generate code

suggestions "for whole lines or entire functions right inside your editor". Indeed, the tagline for Copilot is: "your AI pair programmer". A developer using this plug-in would typically receive real-time code suggestions as they are typing, or would explicitly provide a natural language description (for example, as a code comment) and then receive more comprehensive suggestions, such as entire functions, almost immediately. In many cases, multiple suggestions are provided which the developer can simply cycle through and accept or reject.

This code-generation use case could be applied productively in several ways in computing education contexts. For example, as model solutions have been proposed as a support mechanism in introductory programming [58, 59], students could generate model solutions with Codex for historical assignment, test and exam problems, where solutions may not otherwise exist. They could also generate alternative correct solutions for a problem they have solved, to reflect on their own solution and to compare different algorithms and language constructs. As the accuracy of Codex improves over time, introductory computing pedagogy may shift away from low-level coding and towards problem decomposition and problem solving.

However, Codex is not limited to code-generation tasks, and can generate natural language output from code-based or prose-based input. In the current paper, we explore how this capability can be used to support two novel use cases that relate to the programming exercise lifecycle (see Figure 1). The first of these relates to the generation of programming exercises by the instructor. Given an existing exercise as input, we explore whether Codex can generate novel variations of the exercise that could then be deployed to students. The second of these relates to the generation of feedback to students. Given source code as input, we explore whether Codex can produce useful natural language feedback on that code, particularly in terms of helping students detect bugs prior to submission for grading. In general, the use of a tool like Codex to generate practice problems for computing students in various formats, and to provide useful feedback to students on their progress on those problems, appears to offer great potential.

In the context of programming education, Finnie-Ansley et al. [27] studied the potential of Codex for solving introductory programming assignments. They found that Codex was able to correctly answer most introductory programming problems and that when given typical exam questions, Codex performed better than the average student. The authors note that considering the performance of Codex, and especially that the progress in this area has been rapid, there are clear consequences for introductory programming courses. For example, when models such as Codex that are capable of performing well on programming assignments become more and more common, it becomes increasingly easy for students to use these models to write code for them, essentially engaging in a new type of plagiarism, which might require the utilization of process-based plagiarism detection [33, 46, 54]. While Finnie-Ansley et al. focused mostly on potential challenges Codex-like models will introduce to introductory programming classrooms, our focus in this article is exploring potential opportunities these models can provide in programming education.

3 METHODOLOGY

3.1 Using Codex

Similar to OpenAI's GPT-3 models, Codex can be used both programmatically through an API or through a web-UI. The user provides a priming, i.e. a prompt, to Codex as input and Codex generates new content as output based on the given priming. For example, given a natural language description of desired behavior, Codex will often generate source code for a program that provides that functionality.

For generating content, a custom "stop sequence" can be specified, which causes the generation of text to stop upon creating such a sequence. Other relevant options that we leveraged in this study include maximum token count that controls the length of the generated content and "temperature" that controls the "creativity" or "randomness" of the model. A lower temperature value will further reduce the chances of the model generating less probable tokens, reducing randomness in the creation process. With any temperature value, however, the model is not deterministic and there can be differences in the created content between runs, although this is more common with higher temperature values.

Since the priming given to Codex primes the model on what content should be generated, in addition to generating code, we can for instance prime Codex with an existing programming exercise and some context related words. This guides Codex to try and create content similar to the priming, which in this case would be a similar exercise but with a specified context. For reference, considering e.g. the natural language model GPT-3 (which Codex is based on), using a priming about dogs will likely lead to output related to dogs.

3.2 Creating Programming Exercises and Code Explanations

3.2.1 Choosing inputs for Codex. For the purposes of the analyses in this article, we selected a small set of exercises that have been featured in computing education research and that are often used in the teaching contexts of the researchers, who use the many small exercises approach [4]. We focused on four programming exercises: 1) a variant of the speeding problem [83] where students work with conditionals and returning values, 2) a variant of FizzBuzz [5] where students work with conditionals and lists, and study the importance of ordering of conditional statements, 3) a variant of the Rainfall Problem [80] that has been a recurring problem in computing education research [28, 74], and 4) a currency converter application used in our contexts where students work with objects, methods, and dictionaries. A sample solution for each of the four programming exercises is shown in Appendix A.

Since OpenAI Codex has primarily been evaluated with the Python programming language in prior work [11] and reportedly works best in Python[1], all of our exercises that we use to prime OpenAI Codex are in Python. In our explorations, we used the `code-davinci-001` Codex model, which was the most capable (albeit slowest) version when these experiments were conducted.

3.2.2 Creating programming exercises. We explored a range of priming approaches for creating programming exercises. In the

---

[1]As noted in the OpenAI Codex Beta documentation, last accessed 2022-03-25: https://beta.openai.com/docs/engines/codex-series-private-beta

end, the priming that we found most reliable for creating new programming exercises contained a problem description, a sample solution, and automated tests. In addition, we explored adding programming-related concepts (e.g. conditional, loop) and contextual concepts (e.g. hiking, fishing) to the priming. In general, we observed that introducing concepts led to OpenAI Codex taking these into account when creating programming exercises, although the programming exercises created without the concepts were also meaningful. To see how these primings are formatted, refer to Appendix B. In addition to the examples in Appendix B, when providing the samples as an input to Codex, the samples were suffixed with a stop sequence ("""). After the stop sequence, the priming included the text "Exercise 2", the concepts desired in the created exercise and the identifier for the problem statement (--Problem statement--). An example of a complete priming (i.e. the input to Codex) and one example of the corresponding output generated by Codex can be found in Appendix C.

We generated exercises using the two priming exercises in Appendix B, varying both the programming-related concepts and the contextual concepts (see Table 1). Using a total of nine contextual concepts (and an extra for leaving out the contextual concept) and two programming-related concept sets (and an extra for leaving out the programming-related concepts), we generated a total of $10 \times 3 \times 2 = 60$ different combinations of inputs (contextual concepts $\times$ programming-related concept sets $\times$ exercise primings). In addition, we explored two values for Codex's temperature parameter (0 and 0.75) and created two exercises for each parameter combination. In total, this led to a sample of $60 \times 2 \times 2 = 240$ programming exercises.

3.2.3 Creating code explanations. Similar to creating programming exercises, we explored different types of priming approaches for creating code explanations. We identified three types of primings that led to different types of code descriptions: 1) a high-level description of the code, 2) a problem statement-like description of the code, and 3) a step-by-step explanation of the code. In this work, we focus on the last code explanation type, i.e. the step-by-step explanation of code, as it aligns with the multistructural level of the SOLO taxonomy and is often produced by students when prompted to explain code [52].

In our experiments, using a priming that consisted of the source code, followed by a stop sequence and the text "Step-by-step explanation of the above program:", and a number one followed by a dot, tended to produce step-by-step explanations. As an example, the priming for a simple "Hello world!" program would look as follows:

```
print("Hello world!")

"""Step-by-step explanation of the above program:
1.
```

With the above priming, Codex would create a step-by-step explanation of the code `print("Hello world!")`. For the step-by-step code explanation analysis, we created five explanations for each of the four programming exercise sample solutions in Appendix A, leading to a total of 20 code explanations. Since we were interested in precise explanations instead of creative ones, we used the temperature value 0 to generate each of the explanations.

3.3 Evaluation

3.3.1 Programming exercises. The evaluation of the programming exercises was conducted as mixed-methods research, where the exercises were evaluated both qualitatively and quantitatively.

In the qualitative analysis, we focused on a random sample of 120 programming exercises. Our focus was on the sensibleness, novelty and readiness for use of the created programming exercises, as outlined in RQ1. When assessing *sensibleness*, we study whether the programming exercise represents a sensible problem for students – does the problem statement describe a practical problem that could be given to students to solve? When assessing *novelty*, we study whether the verbatim copy of the programming exercise or a similar programming exercise already exists and can be found online (we used both Google and GitHub for searching). Related to novelty, we also examine the *topicality* of the exercises – how are the different priming concepts accounted for in the created exercises? When assessing *readiness for use*, we consider the amount of manual work a teacher would have to make to the exercises and the associated sample solution and tests.

The qualitative analysis was conducted by four researchers, who first practiced the assessment of sensibleness, novelty, and readiness for use jointly, discussing identified issues and corner cases. The analysis was conducted individually using the rubric outlined in Table 2, where each researcher worked on a subsample of the programming exercises, and assessed the focused items with Yes / No / Maybe statements and added notes whenever needed. All the answers with Maybe were then jointly analyzed by at least two researchers working in tandem to form a consensus on whether they should be considered as Yes or No.

We then quantitatively analysed the Yes / No / Maybe answers and report and discuss the results. For the quantitative analysis, we explore three further questions related to the readiness of use of the exercises, which were calculated from the total body of 240 programming exercises. These questions are outlined in Table 3 and the answers to the questions were obtained programmatically; 1) we tested whether the sample solutions could be run, 2) tested whether the sample solution passed the automated tests, and 3) checked for the statement coverage of the automated tests[2].

3.3.2 Code explanations. Similar to the generated exercises, we analyzed the capability of Codex for generating natural language explanations of code samples typically seen in introductory programming classes.

We analyzed the 20 generated code explanations by inspecting what kinds of mistakes were present and how common they were in the explanations for the different priming programs. When analyzing the code explanations, we answered the question "Are all parts of the code explained?" (Yes / No) and counted the proportion of correctly explained lines out of all the generated explanation lines.

It was feasible for all four researchers to collaboratively assess all of the explanations under evaluation. We discussed each generated explanation in turn, and developed a shared understanding of what it meant for a single line within an explanation to be correct. We decided to be rather strict in our assessment so as to not artificially

---

[2] Analysis of statement coverage of automated tests was conducted using Coverage.py version 6.3.2 (https://coverage.readthedocs.io/).

Table 1: Keywords used for priming exercise generation. The programming-related concepts are placed in two sets to reduce the number of possible combinations.

contextual concepts	programming-related concept set 1: "function"	programming-related concept set 2: "class"
hiking, fishing, relationships, football, music, health, ice hockey, books, cooking	function parameters dictionary dict comprehension arithmetics	class list list comprehension conditional

Table 2: Manual assessment rubric

Aspect	Question	Options
Sensibleness	Does the problem statement describe a sensible problem?	Yes / No / Maybe
Novelty	Are we unable to find the programming exercise via online search (Google and GitHub) of the problem statement?	Yes / No / Maybe
Readiness: problem and solution	Does the problem statement match the model solution?	Yes / No / Maybe
Topicality: function / class	Is the problem statement about a function or class when that concept is provided as a priming concept?	Yes / No / Maybe
Topicality: list / dictionary	Does the problem statement incorporate a list or a dictionary when that concept is provided as a priming concept?	Yes / No / Maybe
Topicality: context	Does the problem statement topic match the given context priming concept?	Yes / No / Maybe
Free-form notes	Notes	Free-form text

Table 3: Automated assessment rubric

Aspect	Question	Answer
Readiness: solution runnability	Can we run the sample solution without errors?	Yes / No / NA
Readiness: solution and tests	Does the sample solution pass the unit tests?	Yes / No / NA
Readiness: test coverage	To what extent do the unit tests cover the model solution (statement coverage)?	0 to 100% / NA

report stronger results, and required the language in each line to be precise. For example, we judged an explanation to be incorrect if it stated "less than or equal to x" where the corresponding code was checking "less than x". Similarly, if there was ambiguity as to whether the "else" part in the explanation of an "elif" was accounted for, we deemed that to be incorrect. For example, in a FizzBuzz program, a line such as "elif number % 3 == 0:" would be classified as incorrect if the explanation of the line began directly with "if the number is divisible by 3" and did not attempt to qualify the description with "otherwise" or a similar phrase to denote its logical relationship to the matching "if". We chose to be lenient only in the case where explanations did not explicitly mention the initialization of variables, even though it could be argued that this may be relevant in a comprehensive explanation.

4 RESULTS

4.1 Programming Exercises

In total, we randomly selected and evaluated 120 of the 240 programming exercises created by OpenAI Codex. Evaluating the programming exercises included assessing their sensibleness, novelty, readiness, and also marking down any additional notes during the process. In addition, for all of the 240 programming exercises, we programmatically assessed whether the sample solutions could be run, whether the automated tests passed, and calculated the statement coverage of the automated tests.

The statistics for sensibleness, novelty, and readiness of the evaluated programming exercises are presented in Table 4. Of these, 75.0% were sensible, 81.8% were novel[3], and 76.7% had a matching sample solution. The free-form notes mostly discussed issues which included existence of redundant information, missing information, missing or incorrect values in sample inputs and/or outputs (some

---

[3]Note that by our definition of novelty, non-sensical problem statements are rather certainly classified as novel.

Table 4: Summary of the manually evaluated programming exercises. An exercise is *sensible* if the requirements are described clearly within a context that makes logical sense, *novel* if the exercise description returns no valid matches when used as the input for a search using Google or GitHub, and has a *matching sample solution* if the generated code solution matches the description.

Exercises	Sensible	Novel	Matches sample solution	Matches priming topic	Matches priming concept function/class	Matches priming concept list/dictionary
120	75.0%	81.8%	76.7%	79.2%	78.3%	75.8%

Table 5: Summary of programmatic analysis of generated programming exercises

	Has sample solution?	Can run the sample solution?	Has tests?	All tests pass?	Test coverage
Percentages	84.6%	89.7%	70.8%	30.9%	98.0%
n out of N	203 / 240	182 / 203	170[1] / 240	51 / 165[1]	48[2] / 51

[1]Five of the generated exercises contained --Tests-- but not --Sample solution-- (needed for automated extraction of the content parts)

[2]The n out of N for test coverage is counted as the number of full coverage (100%) cases out of the number of all test suites that did not fail (i.e. when coverage can be computed)

of the problem statements featured sample inputs and outputs), discussed mismatches between the problem statement and a sample solution, and outlined reasons for why the automated tests would not pass.

The statistics for the programmatic analysis that was conducted on all of the 240 created programming exercises are presented in Table 5. Out of the 240 programming exercises, 203[4] had a sample solution (84.6%). From the 203 sample solutions, 182 (89.7%) could be executed (i.e. running the code did not produce any errors). A total of 170[4] programming exercises had automated tests, while 165 programming exercises had both a sample solution and automated tests. From these 165 programming exercises, 51 had a sample solution that passed the automated tests. Out of the 51 programming exercises with a working sample solution and automated tests, 48 had a 100% statement coverage, and the statement coverage averaged over all the 51 programming exercises was 98.0%. When inspecting the notes for the exercises with automated tests that did not pass the tests, we observed that the most common issue was not related to the code logic, but in how the outputs were handled. In those cases, the sample solution printed a value, while the automated tests expected that the sample solution would return a value (e.g. the tests called a function and expected that the function would return a value, but the function printed a value). We note, of course, that a confusion between printing and returning values is a commonly cited error made by novices [24, 36]. In addition, a common issue was that the tests expected specific numbers that were not possible with the inputs (e.g. checking whether a program correctly extracted and returned a list of even values from a list received as a parameter, a test provided the list [1, 2, 3] as an input to the function and expected that the function would return the list [2, 4]).

In the programmatic analysis results on readiness, presented in Table 5, we see that around 90% of the time the generated sample solutions are valid runnable code, tests are generated and auto-extractable roughly 70% of the time, while only around 30% percent

[4]The actual count is slightly higher, since this value is computed from programmatically extractable content (requires the relevant keyword wrapped with double dashes "--" from priming in the generated content).

of the generated solutions pass the tests (this requires both a sound solution and sound tests). Surprisingly enough, when there are passing generated tests, on average we got 98% test coverage and 48 out of the 51 passing test sets covered 100% of the sample solution statements. Further, we noted that in multiple cases only minor tweaks would have been necessary to transform failing tests into passing ones. In the cases where tests were missing, we could simply add the generated exercise to the initial priming and the tests would likely be generated on a "second" run (we tested this behavior directly when exploring the output).

4.2 Code Explanations

A total of 20 code explanations created by OpenAI Codex from the source code available in Appendix A were jointly analyzed by the researchers. When evaluating the code explanations, we studied whether all parts of the code were explained, and whether each line was correctly explained. Table 6 provides statistics for the analysis. From the 20 code explanations, 90% explained all parts of the code. In total, the code explanations had 174 line-by-line explanations, out of which 117 were correct (67.2%).

The incorrect explanations were mostly related to incorrect explanation of comparison and branching conditionals, e.g. Codex often explained speed > 100 as "if speed is less than 100" or elif number % 3: as "if number is divisible by three". We consistently found these problems in each of the explanations generated for two of our four code samples used for priming explanations. Another recurring, although less persistent, incorrect line was one that included the phrase "program ends if user inputs" when in actuality, a while loop was ended and the program still executed remaining lines after the while loop. Notably, for the fourth of our priming codes, the one which contained a currency converter class and usage of the class, none of the five generated explanations contained incorrect lines and the explanations covered every part of the program.

Table 6: Code explanation results

Code explanations	All parts of code explained	Total lines	Lines correctly explained
20	90%	174	117 (67.2%)

5 DISCUSSION

5.1 Programming Exercises

Most of the generated exercises appeared to be both sensible and novel, and included a sample solution that could be executed. Much less impressive was the quality of the test suites and the performance of the code against those tests. Only around 70% of exercises included tests at all and of those, the set of tests passed successfully in less than a third of cases. However, in practice, it may be possible to address this shortcoming in two ways. Firstly, we tasked Codex with generating all parts of the programming exercise in a single output step. OpenAI's demonstration of Codex illustrated particularly good performance when working interactively with Codex, and prompting it step by step[5]. Therefore, we may have had better success in generating good test cases by explicitly prompting for them. This could be achieved by providing a problem description and a sample solution as input to Codex, and priming it to generate only the tests. We tested this in practice by using Codex to successfully create tests for a handful of programming exercises that were created with Codex. Secondly, given that it is possible to automate the verification of tests by running the sample solution, simply regenerating an output repeatedly until a set of successful tests is produced could be a valid strategy in practice.

While our main focus in this paper has been on the generation of exercises and their readiness for use, we believe there is value even in those exercises that have room for improvement. In particular, they may provide inspiration to instructors who can easily modify the problems by hand, and they could even form the basis for new kinds of student activities. For example, many educators will appreciate removing some of the frustration of needing to write 'yet another' practice problem or exam question, and this has led to some community work around sharing programming exercises [22, 34]. The ease with which novel exercises can be generated with a tool like Codex, even if the resulting exercises are not used verbatim, can help instructors brainstorm ideas quickly and overcome the computing educators' version of writer's block. After all, modifying an existing programming exercise is easier than writing one from scratch. Another issue that we observed, even in generated exercises that were novel, sensible and had tests, was that they were sometimes under-specified. That is, the problem description did not explicitly specify how boundary cases should be handled. A good example of this is the 'Fisherman' class that is shown at the very start of this paper. The problem description does not state what should happen if the count for a particular type of fish reaches zero when the `throw_away()` method is called. In this case, the sample solution to the problem provides the answer, which is that the fish type should be removed from the dictionary (rather than remaining and being displayed with a value of zero). Such problems may provide a good starting point for student discussions around

program testing, and could form the basis for new tasks where students must improve problem specifications.

One aspect of the exercise generation that we found particularly surprising was how well the contextual concepts and the programming related concepts were incorporated into novel problem descriptions. Table 7 shows the problem descriptions for two of the exercises that were generated with Codex. Each exercise was generated from a different programming prime (see Appendix B) but bear little resemblance to those primes. The generated problem statements were not just trivial variations (such as grammatical changes) of the priming exercises but were materially different and incorporated the contextual themes quite naturally, such as computing a list of friends or calculating the elevation change of a hiker for the 'relationship' and 'hiking' themes respectively, as shown in Table 7. Exercises generated using the contextual theme of 'books' included test cases involving popular titles and authors such as 'Ender's Game', 'Rainbow Six' and 'J.R.R. Tolkien', the theme 'football' resulted in tests involving 'Lionel Messi' and 'Cristiano Ronaldo', and the theme 'health' resulted in exercises where smoking cigarettes and eating apples were contrasted as unhealthy and healthy activities, respectively.

This ability to automatically contextualize problem statements may have useful applications in practice. For teachers, it offers the potential to generate programming exercises that target specific constructs and require certain kinds of solutions. For students, prior work exploring the problem description effect in computing education has shown that a familiar context within the narrative of a problem statement might have a positive effect on performance [44]. It is not possible for a teacher to select appropriate contexts that are both familiar and of interest to all students, especially given the diversity of backgrounds in large first-year cohorts. Our results suggest that it may be possible for individual students to provide their own keywords and have tailored exercises generated for their personal use, possibly using teacher created exercises as primes. Exploring this in more detail, and in particular collecting students' thoughts on the suitability of tailored questions compared to more generic sets of problems, is a fascinating avenue for future work.

5.2 Code Explanations

In this paper, our investigation of code explanations generated with Codex focused on line-by-line descriptions of code. As discussed in Section 2.2, these kinds of descriptions align with the multistructural level of the SOLO taxonomy and are commonly produced by students when asked to explain code, especially lower performing students who stand to benefit the most from some help with code explanation and reflection [52]. The Codex generated explanations were quite thorough in that all essential parts of the code were usually addressed in the explanation, but they often contained minor inaccuracies. This does raise questions about the utility of the explanations for helping students understand or debug their own

---
[5]https://openai.com/blog/openai-codex/

Table 7: Examples of problem statements where contextual concepts (relationships and hiking) and programming concepts (class and function) have been successfully incorporated. The source code for the primes ('speeding_check' and 'Converter') can be found in Appendix B. Whitespace and other structural formatting has been removed for space reasons.

Prime: 'speeding_check' Contextual concept: 'relationships' Programming concept: 'class'	Prime: 'Converter' Contextual concept: 'hiking' Programming concept: 'function'
Write a class called Person that has a list of friends. It has methods to add a friend and remove a friend. Write a function called find_pals that takes a single parameter called person and that will list the friends of this person. Use the Person class to create two persons and add friends to them. Print out all friends of the first person.	Write a function called hiking called with these parameters: 'elevation_chart' is a dictionary containing the elevation in meters of various locations in the world; 'path' is a list of tuples, where each tuple contains two names (strings) of locations in the chart. The first name is the location where the path starts and the second name is the location where the path ends; 'uphill_hiking' is a number that represents how much the hiker is willing to walk up hill. In other words, it is the maximum percentage of an elevation that the hiker is willing to climb; 'downhill_hiking' is a number that represents how much the hiker is willing to walk down hill. In other words, it is the maximum percentage of an elevation that the hiker is willing to climb. The function should compute the total vertical distance traveled by the hiker on the path and return the distance in meters.

code. However, as a prompt for a discussion between a teaching assistant and a student, the generated explanations may still provide a useful starting point. Other kinds of technological scaffolds for supporting sit-down conversations between students and teachers, such as the Pensieve tool [94], have proven valuable.

We explicitly primed Codex to produce multistructural level explanations using a prompt that asked for a 'step-by-step explanation' of the code and ended with the initial enumerated list item (i.e. '1.'). An interesting related question is the extent to which Codex can generate higher-level explanations, at the relational level of the SOLO taxonomy, that successfully abstract the purpose of the code. When experimenting with many other kinds of priming statements for generating the explanations, we found that Codex very rarely provided high-level descriptions. This supports the findings of Kojima et al. who found that large language models seem to perform better in reasoning tasks when priming them to "think step by step" [42]. Even a very explicit prompt, such as "A high-level description of the above program:", would still usually result in a line-by-line explanation being produced. An example of this is the following explanation generated in response to the 'speeding_check program', which also illustrates a subtle error in that the cases where the input is equal to 120 or 100 are not correctly accounted for:

- The speed_check function checks the speed of the car and returns the required result. The function first compares the speed value with 120, and if the value is more than 120, it returns "You are fined for $200". If the speed value is more than 100 but less than 120, it returns "You are fined for $100", and if the value is less than 100, it will return "All good, race ahead".

We occasionally observed responses that were at a more abstract level. Several examples are illustrated in Table 8. Examples A and B show relational level responses to the 'speeding_check' and 'fizz_buzz' prompts. Example C includes some interesting background information on the FizzBuzz problem. Example D is nonsensical output that was generated once in response to the Rainfall problem prompt.

5.3 Future Work

We see great potential for future work utilizing Codex and other similar models in the context of programming education. Given the positive results we have observed in terms of programming exercise generation, we are interested in developing an automated exercise generator powered by Codex that could be used by instructors. The tool could provide a validation layer on top of Codex enabling teachers to filter out any questions that do not include valid sample solutions or a comprehensive set of accurate tests. In the current work, our focus was on introductory programming exercises, but it would be interesting to explore the generation of exercises of greater complexity. For example, investigating whether Codex is capable of generating accurate specifications for larger assignments or projects, or for those that relate to more advanced computing concepts.

With respect to the code explanations, future work should explore whether these could be used as the basis for generating multiple-choice questions related to the student's own code, similar to prior work [43], which could serve as a reflection task. For example, one could create an explanation of the student's program as well as several other explanations for slight modifications to this program, similar in methodology to mutation testing [38] (e.g. with relational operators flipped). This set of explanations could then be shown to the student, with their task being to select the explanation that best matches their code. In a similar vein, the explanations created with Codex could be turned into Parsons problems [18], for example where each line of a line-by-line explanation is presented to the student in a randomized order for them to unscramble. Although we did observe inaccuracies in the code explanations generated in this study that may constrain such ideas for now, models like Codex are likely to continue to improve over time.

In this work, we qualitatively analyzed the code explanations created with Codex. Future work should explore how such explanations could be used by students in practice, for example, by having students assess the quality and usefulness of the created explanations. One instructional approach that has become increasingly common in computing education is learnersourcing [41] where students participate in the creation and evaluation of course materials such as questions and exercises (see e.g. [14, 16, 47, 65]).

Table 8: Examples of uncommon code explanations produced in response to the speeding_check (A) and FizzBuzz (B, C) problems. (D) is an example of a nonsensical explanation of the Rainfall problem.

A	Takes the input of the speed of the car. Checks the speed and prints a fine according to the speed of the driver
B	is a simple program that checks if a number is divisible by 3, 5, or both.
C	This program is a variation of the FizzBuzz which is a popular coding challenge. It can be found here: https://blog.codinghorror.com/why-cant-programmers-program/. The above program reads an array of integers, performs an if-else conditional check on the numbers and prints the result
D	Has many global and local variable lookup. Has an initial and final node. Has a number of variables, like variable names, that are used to access the values and are used to group the list of methods.

For example, the Quizius tool described by Saarinen et al. has students contribute questions to a repository, and their answers are used to produce statistical estimates of the prevalence of topic misconceptions [70]. A novel approach to learnersourcing could have students focus on evaluating Codex-created artefacts. We envision a new type of learnersourcing we coin "robosourcing", where Codex-like machine learning models are used to automatically create artefacts similar to traditional crowdsourcing, but where these "robosourced" learning materials are then evaluated by students. This would address one of the major challenges related to the use of learnersourcing which is that students tend to be much more inclined to use and evaluate resources created by others than they are to create resources themselves [66, 78].

There are also obvious applications of Codex that we did not evaluate in this work, that have important implications for computing education. In particular, the real-time auto-completion and auto-generation of existing source code. One potential use of Codex in programming education could be a tool similar to GitHub Copilot[6] which presents students with hints or suggestions on code improvements. The tool could enable an instructor to tune this feedback in a way that is suitable pedagogically, rather than unleashing the full power of these tools on students. This avenue of research maps to the Student → Attempt pathway on the model we present in Figure 1.

Lastly, the combination of GPT-3 and Codex could facilitate the whole course material creation process. To be clear, we believe it is unlikely that large language models such as GPT-3 and Codex could fully replace teachers as the creators of learning material. However, as the development in natural language processing is rapid and the capabilities of the models are still improving, it is possible that in the near future an instructor could expedite the creation of both textual materials and programming exercises through carefully constructed prompts to these models, where the output of the models would need only minor changes before being published to students.

5.4 Threats to Validity

There are some threats to the validity of this work which we discuss here. Firstly, regarding our qualitative analysis of the Codex-created programming exercises and code explanations, we had a relatively small set of created examples, and we explored only a relatively few different types of prompts: four different exercises for code explanations, and two different exercises as prompts when creating

new exercises. It is possible that different prompts could have led to outputs of different quality, and an evaluation of a wider variety of inputs is warranted.

Additionally, in our qualitative analysis of the created programming exercises, we did not calculate an inter-rater reliability. However, the researchers worked closely together on a subset of the evaluations and discussed all unclear cases, partly addressing this concern.

When considering the novelty of the created programming exercises, we searched both Google and GitHub for possible matches. It is possible that this analysis misses some sources such as password-protected sites that are not indexed by Google. It is also possible that some repositories that were used by Codex during training (and from which Codex could technically produce verbatim content) may have been deleted or made private between the time Codex was fine-tuned and our analysis. However, we consider this possibility very remote [12]. In addition, our definition of novelty mostly relied on the exercises being novel in the sense that they are not direct copies of existing exercises. Future work should study novelty with a more broad definition, for example, studying whether Codex combines programming concepts in novel ways.

One potential issue related to the generalizability of our results is that we focused on creating programming exercises and code explanations in English. It may be that the creation of these in languages other than English is harder (e.g. that the created exercises are more likely to be nonsensical). To address this concern, we conducted a brief exploration of how well Codex can create exercises in Finnish, a language with approximately 5.8 million native speakers and which is the first language of three of the authors. Based on this brief exploration, the created exercises were sensible and the language in the accompanied text (e.g. problem statement) was generally good.

When considering the performance of Codex at solving programming problems [27], a question that might arise is whether any value added by this tool for the instructor will immediately be negated by its use by students for plagiarism. However, students will be able to use these types of models regardless of work exploring potential benefits. Additionally, other fields such as mathematics and physics have suffered from the problem of automatically solvable exercises for decades [37] – it is also a common practice in such disciplines to provide solutions to problems at the end of textbooks. Being able to solve exercises automatically or having solutions available does not prevent those who want to learn from doing so.

---

[6]https://copilot.github.com/

We acknowledge that large language models have been shown to suffer from similar biases to humans [73]; this is to be expected as they have been trained with human-generated data. Thus, it is possible that, for example, using these models for creating exercises could lead to exercises that perpetuate biases. We believe the human-in-the-loop approach is essential in order to moderate such biases when utilizing large language models to generate learning materials.

Lastly, we mostly analyzed the created exercises through the lens of the "many small exercises" pedagogical approach, and did not, for example, explore the creation of larger programming exercises. Thus, whether Codex is applicable in contexts with larger exercises remains unknown. Similarly, we only studied Python exercises – it is possible that Codex is not as proficient in creating new exercises in some other programming languages as it has been reported that Codex is most proficient in Python[7].

6 CONCLUSION

In this work, we explored to what extent OpenAI Codex could 1) support instructors in creating programming exercises and 2) generate useful explanations of source code. We studied this through two research questions which we answer as follows:

RQ1: To what extent are programming exercises created using OpenAI Codex sensible, novel, and readily applicable?

A: We found that the majority of the programming exercises created by Codex were sensible, novel, and included an appropriate sample solution. Additionally, we observed that both the programmatic topic as well as the contextual topic of the created exercises could be easily influenced. This result suggests that Codex could indeed be a useful tool for instructors to facilitate the exercise creation process. We did, however, observe that the programming exercises were rarely in a state where one could directly – without any adjustments – add them to a course. In particular, problem statements did not always discuss corner cases and many exercises lacked tests or had faulty tests. We see that the corner cases could be easily added by a teacher, or adding them could be turned into a learning activity. Similarly, in the case of missing tests, we note that tests can be easily generated with Codex, and many of the faulty tests were related to issues that would be easy to fix (e.g. by adding a number, or by returning a value instead of printing it).

RQ2: How comprehensive and accurate are OpenAI Codex natural language explanations of code solutions to introductory programming exercises?

A: Our results suggest that the explanations created by Codex cover a majority (90%) of the code, although contain some inaccuracies (67.2% of explanation lines were correct). We observed that in most cases, the erroneous lines contained only minor mistakes that could easily be fixed by an instructor or by teaching assistants. Assessing the value of such explanations in practice would be interesting future work, for example, whether they could be used by teaching assistants to expedite the process of helping novice programmers.

In summary, our results support earlier findings that large language models are zero-shot [42] and few-shot learners [9], meaning that they perform well in tasks even when not given any, or given just a few, task-related examples as input. Our work suggests that modern machine learning models such as OpenAI Codex provide many opportunities for programming course designers, although potential challenges outlined in prior work [27] should not be ignored. Our present analysis showed remarkable results in creating novel and sensible programming exercises with ready-made sample solutions and automated tests, despite the presence of some accuracy and quality issues (that could be easily fixed by human hands). We also saw promise in the created code explanations. We foresee that the affordances of generative models for computing education practice and research will only improve over time with the continuing evolution of these models.

ACKNOWLEDGMENTS

REFERENCES

[1] Onni Aarne, Petrus Peltola, Juho Leinonen, and Arto Hellas. 2018. A study of pair programming enjoyment and attendance using study motivation and strategy metrics. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education.* 759–764.

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

[3] Ibrahim Albluwi. 2019. Plagiarism in Programming Assessments: A Systematic Review. *ACM Trans. Comput. Educ.* 20, 1, Article 6 (dec 2019), 28 pages. https://doi.org/10.1145/3371156

[4] Joe Michael Allen, Frank Vahid, Kelly Downey, and Alex Daniel Edgcomb. 2018. Weekly programs in a CS1 class: Experiences with auto-graded many-small programs (MSP). In *2018 ASEE Annual Conference & Exposition.*

[5] Cory Althoff. 2022. *The Self-Taught Programmer: The Definitive Guide to Programming Professionally.* Hachette UK.

[6] Albert Bandura. 1977. Self-efficacy: toward a unifying theory of behavioral change. *Psychological review* 84, 2 (1977).

[7] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. 2021. STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21).* Association for Computing Machinery, New York, NY, USA, 1062–1068. https://doi.org/10.1145/3408877.3432430

[8] John B. Biggs and K. F. Collis. 1982. *Evaluating the quality of learning : the SOLO taxonomy (structure of the observed learning outcome) / John B. Biggs; Kevin F. Collis.* Academic Press New York. xiii, 245 p. : pages.

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in neural information processing systems.* 1877–1901.

[10] Binglin Chen, Sushmita Azad, Rajarshi Haldar, Matthew West, and Craig Zilles. 2020. *A Validated Scoring Rubric for Explain-in-Plain-English Questions.* Association for Computing Machinery, New York, NY, USA, 563–569. https://doi.org/10.1145/3328778.3366879

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[12] Matteo Ciniselli, Luca Pascarella, and Gabriele Bavota. 2022. To What Extent do Deep Learning-based Code Recommenders Generate Predictions by Cloning Code from the Training Set? *arXiv preprint arXiv:2204.06894* (2022).

[13] Catherine H Crouch and Eric Mazur. 2001. Peer instruction: Ten years of experience and results. *American journal of physics* 69, 9 (2001), 970–977.

[14] Paul Denny, Diana Cukierman, and Jonathan Bhaskar. 2015. Measuring the Effect of Inventing Practice Exercises on Learning in an Introductory Programming Course. In *Proceedings of the 15th Koli Calling Conference on Computing Education*

---

[7]As noted in the OpenAI Codex Beta documentation, last accessed 2022-03-25: https://beta.openai.com/docs/engines/codex-series-private-beta

Research (Koli, Finland) *(Koli Calling '15)*. Association for Computing Machinery, New York, NY, USA, 13–22.   https://doi.org/10.1145/2828959.2828967

[15] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Codewrite: supporting student-driven practice of java. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. 471–476.

[16] Paul Denny, Ewan Tempero, Dawn Garbett, and Andrew Petersen. 2017. Examining a Student-Generated Question Activity Using Random Topic Assignment. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 146–151.   https://doi.org/10.1145/3059009.3059033

[17] Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, Roman Wang, Nikhil Singh, Taylor L. Patti, Jayson Lynch, Avi Shporer, Nakul Verma, Eugene Wu, and Gilbert Strang. 2021. A Neural Network Solves, Explains, and Generates University Math Problems by Program Synthesis and Few-Shot Learning at Human Level.   https://doi.org/10.48550/ARXIV.2112.15594

[18] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. 2020. A review of research on Parsons problems. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. 195–202.

[19] Angela Lee Duckworth and Lauren Eskreis-Winkler. 2013. True grit. *Aps Observer* 26 (2013).

[20] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. 2021. Cognitive Load Theory in Computing Education Research: A Review. (2021). http://rodrigoduran.net/papers/CLT_in_CER.pdf Preprint.

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

[22] Stephen H. Edwards, Jürgen Börstler, Lillian N. Cassel, Mark S. Hall, and Joseph Hollingsworth. 2008. Developing a Common Format for Sharing Programming Assignments. *SIGCSE Bull.* 40, 4 (nov 2008), 167–182.   https://doi.org/10.1145/1473195.1473240

[23] K Anders Ericsson, Ralf T Krampe, and Clemens Tesch-Römer. 1993. The role of deliberate practice in the acquisition of expert performance. *Psychological review* 100, 3 (1993), 363.

[24] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common Logic Errors Made by Novice Programmers. In *Proceedings of the 20th Australasian Computing Education Conference* (Brisbane, Queensland, Australia) *(ACE '18)*. Association for Computing Machinery, New York, NY, USA, 83–89.   https://doi.org/10.1145/3160489.3160493

[25] Katrina Falkner and Judy Sheard. 2019. *Pedagogical approaches* (1st ed.). Cambridge University Press, United Kingdom, 445–480.   https://doi.org/10.1017/9781108654555.016

[26] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[27] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Australasian Computing Education Conference*. 10–19.

[28] Kathi Fisler. 2014. The recurring rainfall problem. In *Proceedings of the tenth annual conference on International computing education research*. 35–42.

[29] Max Fowler, Binglin Chen, Sushmita Azad, Matthew West, and Craig Zilles. 2021. Autograding "Explain in Plain English" Questions Using NLP. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1163–1169.   https://doi.org/10.1145/3408877.3432539

[30] Brian Hanks, Sue Fitzgerald, Renée McCauley, Laurie Murphy, and Carol Zander. 2011. Pair programming in education: a literature review. *Computer Science Education* 21, 2 (2011), 135–173.   https://doi.org/10.1080/08993408.2011.579808 arXiv:https://doi.org/10.1080/08993408.2011.579808

[31] Mohammed Hassan and Craig Zilles. 2021. Exploring 'Reverse-Tracing' Questions as a Means of Assessing the Tracing Skill on Computer-Based CS 1 Exams. In *Proceedings of the 17th ACM Conference on International Computing Education Research* (Virtual Event, USA) *(ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 115–126.   https://doi.org/10.1145/3446871.3469765

[32] John Hattie and Helen Timperley. 2007. The power of feedback. *Review of educational research* 77, 1 (2007), 81–112.

[33] Arto Hellas, Juho Leinonen, and Petri Ihantola. 2017. Plagiarism in take-home exams: help-seeking, collaboration, and systematic cheating. In *Proceedings of the 2017 ACM conference on innovation and technology in computer science education*. 238–243.

[34] David Hovemeyer, Matthew Hertz, Paul Denny, Jaime Spacco, Andrei Papancea, John Stamper, and Kelly Rivers. 2013. CloudCoder: Building a Community for Creating, Assigning, Evaluating and Sharing Programming Exercises (Abstract Only). In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 742.   https://doi.org/10.1145/2445196.2445451

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

[36] Cruz Izu and Peter Dinh. 2018. Can Novice Programmers Write C Functions?. In *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. 965–970.   https://doi.org/10.1109/TALE.2018.8615375

[37] MA Jenkins and Joseph Frederick Traub. 1967. *An algorithm for an automatic general polynomial solver*. Citeseer.

[38] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[39] Cazembe Kennedy, Aubrey Lawson, Yvon Feaster, and Eileen Kraemer. 2020. Misconception-Based Peer Feedback: A Pedagogical Technique for Reducing Misconceptions. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 166–172.   https://doi.org/10.1145/3341525.3387392

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

[41] Juho Kim. 2015. *Learnersourcing: improving learning with collective learner activity*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[42] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. *arXiv preprint arXiv:2205.11916* (2022).

[43] Teemu Lehtinen, André L Santos, and Juha Sorva. 2021. Let's Ask Students About Their Programs, Automatically. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 467–475.

[44] Juho Leinonen, Paul Denny, and Jacqueline Whalley. 2021. Exploring the Effects of Contextualized Problem Descriptions on Problem Solving. In *Australasian Computing Education Conference* (Virtual, SA, Australia) *(ACE '21)*. Association for Computing Machinery, New York, NY, USA, 30–39.   https://doi.org/10.1145/3441636.3442302

[45] Juho Leinonen, Paul Denny, and Jacqueline Whalley. 2022. A Comparison of Immediate and Scheduled Feedback in Introductory Programming Projects. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 885–891.   https://doi.org/10.1145/3478431.3499372

[46] Juho Leinonen, Krista Longi, Arto Klami, Alireza Ahadi, and Arto Vihavainen. 2016. Typing patterns and authentication in practical programming exams. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. 160–165.

[47] Juho Leinonen, Nea Pirttinen, and Arto Hellas. 2020. Crowdsourcing Content Creation for SQL Practice. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 349–355.

[48] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-Level Code Generation with AlphaCode. *arXiv preprint arXiv:2203.07814* (2022).

[49] Raymond Lister. 2020. On the Cognitive Development of the Novice Programmer: And the Development of a Computing Education Researcher. In *Proceedings of the 9th Computer Science Education Research Conference* (Virtual Event, Netherlands) *(CSERC '20)*. Association for Computing Machinery, New York, NY, USA, Article 2, 15 pages.   https://doi.org/10.1145/3442481.3442498

[50] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) *(ITiCSE-WGR '04)*. Association for Computing Machinery, New York, NY, USA, 119–150.   https://doi.org/10.1145/1044550.1041673

[51] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further Evidence of a Relationship between Explaining, Tracing and Writing Skills in Introductory Programming. *SIGCSE Bull.* 41, 3 (jul 2009), 161–165.   https://doi.org/10.1145/1595496.1562930

[52] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. *SIGCSE Bull.* 38, 3 (jun 2006), 118–122.   https://doi.org/10.1145/1140123.1140157

[53] Richard Lobb and Jenny Harlow. 2016. Coderunner: A Tool for Assessing Computer Programming Skills. *ACM Inroads* 7, 1 (feb 2016), 47–51.   https://doi.org/10.1145/2810041

[54] Krista Longi, Juho Leinonen, Henrik Nygren, Joni Salmi, Arto Klami, and Arto Vihavainen. 2015. Identification of programmers from typing patterns. In *Proceedings of the 15th Koli Calling conference on computing education research*. 60–67.

[55] Renée McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: a review

of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92. https://doi.org/10.1080/08993400802114581 arXiv:https://doi.org/10.1080/08993400802114581

[56] Laurie Murphy, Sue Fitzgerald, Raymond Lister, and Renée McCauley. 2012. Ability to 'explain in Plain English' Linked to Proficiency in Computer-Based Programming. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (Auckland, New Zealand) *(ICER '12)*. Association for Computing Machinery, New York, NY, USA, 111–118. https://doi.org/10.1145/2361276.2361299

[57] Terence Nip, Elsa L. Gunter, Geoffrey L. Herman, Jason W. Morphew, and Matthew West. 2018. Using a Computer-Based Testing Facility to Improve Student Learning in a Programming Languages and Compilers Course. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 568–573. https://doi.org/10.1145/3159450.3159500

[58] Henrik Nygren, Juho Leinonen, and Arto Hellas. 2019. Non-restricted Access to Model Solutions: A Good Idea?. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 44–50.

[59] Henrik Nygren, Juho Leinonen, Nea Pirttinen, Antti Leinonen, and Arto Hellas. 2019. Experimenting with model solutions as a support mechanism. In *Proceedings of the 1st UK & Ireland Computing Education Research Conference*. 1–7.

[60] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education (TOCE)* (2022).

[61] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2021. Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs? *arXiv preprint arXiv:2112.02125* (2021).

[62] Hammond Pearce, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Brendan Dolan-Gavitt. 2022. Pop Quiz! Can a Large Language Model Help With Reverse Engineering? *arXiv preprint arXiv:2202.01142* (2022).

[63] D. N. Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1986. Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research* 2, 1 (1986), 37–55. https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL

[64] Robert Phillips, Dan Lockton, Sharon Baurley, and Sarah Silve. 2013. Making Instructions for Others: Exploring Mental Models through a Simple Exercise. *Interactions* 20, 5 (sep 2013), 74–79. https://doi.org/10.1145/2505290

[65] Nea Pirttinen, Vilma Kangas, Irene Nikkarinen, Henrik Nygren, Juho Leinonen, and Arto Hellas. 2018. Crowdsourcing programming assignments with Crowd-Sorcerer. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 326–331.

[66] Nea Pirttinen and Juho Leinonen. 2022. Can Students Review Their Peers? Comparison of Peer and Instructor Reviews. In *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol 1*.

[67] Leo Porter, Daniel Zingaro, Cynthia Lee, Cynthia Taylor, Kevin C. Webb, and Michael Clancy. 2018. Developing Course-Level Learning Goals for Basic Data Structures in CS2. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 858–863. https://doi.org/10.1145/3159450.3159457

[68] Ruixiang Qi and Davide Fossati. 2020. *Unlimited Trace Tutor: Learning Code Tracing With Automatically Generated Programs*. Association for Computing Machinery, New York, NY, USA, 427–433. https://doi.org/10.1145/3328778.3366939

[69] Emily Q Rosenzweig, Allan Wigfield, and Jacquelyne S Eccles. 2019. Expectancy-value theory and its relevance for student motivation and learning. (2019).

[70] Sam Saarinen, Shriram Krishnamurthi, Kathi Fisler, and Preston Tunnell Wilson. 2019. Harnessing the Wisdom of the Classes: Classsourcing and Machine Learning for Assessment Instrument Generation. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 606–612. https://doi.org/10.1145/3287324.3287560

[71] Kate Sanders, Marzieh Ahmadzadeh, Tony Clear, Stephen H Edwards, Mikey Goldweber, Chris Johnson, Raymond Lister, Robert McCartney, Elizabeth Patitsas, and Jaime Spacco. 2013. The Canterbury QuestionBank: Building a repository of multiple-choice CS1 and CS2 questions. In *Proceedings of the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports*. 33–52.

[72] Kate Sanders, Jonas Boustedt, Anna Eckerdal, Robert McCartney, and Carol Zander. 2017. Folk Pedagogy: Nobody Doesn't Like Active Learning. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 145–154. https://doi.org/10.1145/3105726.3106192

[73] Patrick Schramowski, Cigdem Turan, Nico Andersen, Constantin A Rothkopf, and Kristian Kersting. 2022. Large pre-trained language models contain human-like biases of what is right and wrong to do. *Nature Machine Intelligence* 4, 3 (2022), 258–268.

[74] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do we know how difficult the rainfall problem is?. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. 87–96.

[75] Judy Sheard, Angela Carbone, Raymond Lister, Beth Simon, Errol Thompson, and Jacqueline L. Whalley. 2008. Going SOLO to Assess Novice Programmers. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education* (Madrid, Spain) *(ITiCSE '08)*. Association for Computing Machinery, New York, NY, USA, 209–213. https://doi.org/10.1145/1384271.1384328

[76] Lee S Shulman. 2005. Signature pedagogies in the professions. *Daedalus* 134, 3 (2005), 52–59.

[77] Valerie J Shute. 2008. Focus on formative feedback. *Review of educational research* 78, 1 (2008), 153–189.

[78] Anjali Singh, Christopher Brooks, Yiwen Lin, and Warren Li. 2021. What's In It for the Learners? Evidence from a Randomized Field Experiment on Learnersourcing Questions in a MOOC. In *Proceedings of the Eighth ACM Conference on Learning@ Scale*. 221–233.

[79] E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (sep 1986), 850–858. https://doi.org/10.1145/6592.6594

[80] Elliot Soloway and Kate Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Transactions on software engineering* 5 (1984), 595–609.

[81] Ben Stephenson. 2018. An Experience Using On-Computer Programming Questions During Exams. In *Proceedings of the 23rd Western Canadian Conference on Computing Education* (Victoria, BC, Canada) *(WCCCE '18)*. Association for Computing Machinery, New York, NY, USA, Article 11, 6 pages. https://doi.org/10.1145/3209635.3209639

[82] Zahid Ullah, Adidah Lajis, Mona Jamjoom, Abdulrahman Altalhi, Abdullah Al-Ghamdi, and Farrukh Saleem. 2018. The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education* 26, 6 (2018), 2328–2341. https://doi.org/10.1002/cae.21974 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.21974

[83] Anne Venables, Grace Tan, and Raymond Lister. 2009. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (Berkeley, CA, USA) *(ICER '09)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1584322.1584336

[84] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. 2014. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the tenth annual conference on International computing education research*. 19–26.

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

[86] Regina Vollmeyer and Falko Rheinberg. 2005. A surprising effect of feedback on learning. *Learning and instruction* 15, 6 (2005), 589–602.

[87] Lev Semenovich Vygotsky and Michael Cole. 1978. *Mind in society: Development of higher psychological processes*. Harvard university press.

[88] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 2006. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) *(ACE '06)*. Australian Computer Society, Inc., AUS, 243–252.

[89] Joseph Jay Williams, Juho Kim, Anna Rafferty, Samuel Maldonado, Krzysztof Z. Gajos, Walter S. Lasecki, and Neil Heffernan. 2016. AXIS: Generating Explanations at Scale with Learnersourcing and Machine Learning. In *Proceedings of the Third (2016) ACM Conference on Learning @ Scale* (Edinburgh, Scotland, UK) *(L@S '16)*. Association for Computing Machinery, New York, NY, USA, 379–388. https://doi.org/10.1145/2876034.2876042

[90] Laurie Williams, Robert R Kessler, Ward Cunningham, and Ron Jeffries. 2000. Strengthening the case for pair programming. *IEEE software* 17, 4 (2000), 19–25.

[91] John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 51–59. https://doi.org/10.1145/3230977.3230999

[92] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J. Ko. 2019. A Theory of Instruction for Introductory Programming Skills. *Computer Science Education* 29, 2-3 (2019), 205–253. https://doi.org/10.1080/08993408.2019.1565235

[93] Benjamin Xie, Greg L. Nelson, and Amy J. Ko. 2018. An Explicit Strategy to Scaffold Novice Program Tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 344–349. https://doi.org/10.1145/3159450.3159527

[94] Lisa Yan, Annie Hu, and Chris Piech. 2019. Pensieve: Feedback on Coding Process for Novices. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 253–259.

A   SAMPLE SOLUTIONS TO PROGRAMMING EXERCISES OUTLINED IN 3.2.1

```python
def speeding_check(speed):
  if speed > 120:
    return "You are fined for $200"
  elif speed > 100:
    return "You are fined for $100"
  else:
    return "All good, race ahead"

print(speeding_check(88))
print(speeding_check(110))
print(speeding_check(130))
```

```python
def fizz_buzz(numbers):
  for number in numbers:
    if number % 3 == 0 and number % 5 == 0:
      print("FizzBuzz")
    elif number % 3 == 0:
      print("Fizz")
    elif number % 5 == 0:
      print("Buzz")
    else:
      print(number)
```

```python
total = 0
count = 0

while True:
  value = int(input("Write value, 9999 ends."))
  if value == 9999:
    break

  if value < 0 or value > 1000:
    print("Invalid input")
    continue

  total += value
  count += 1

if count == 0:
  print("No inputs")
else:
  print(f"Average: {total/count}")
```

```python
class Converter():
  def __init__(self, exchange_rates):
    self.exchange_rates = exchange_rates

  def convert(self, from_currency, to_currency,
    ↪ amount):
    amount_in_usd = amount / self.exchange_rates[
    ↪ from_currency]
    return amount_in_usd * self.exchange_rates[
    ↪ to_currency]

converter = Converter({"USD": 1, "EUR": 0.9, "GBP"
    ↪ : 0.75})
print(converter.convert("EUR", "GBP", 100))
```

B SAMPLE PRIMINGS FOR PROGRAMMING EXERCISE GENERATION

```
"""Exercise 1
−−Keywords−−
cars
function
parameters
conditional
−−Problem statement−−
Write a function called speeding_check that takes
    ↪ a single parameter speed and prints out "
    ↪ You are
fined for $200" if the speed is above 120, "You
    ↪ are fined for $100" if the speed is above
    ↪ 100 but
below 120 and otherwise prints "All good, race
    ↪ ahead".
−−Sample solution−−
def speeding_check(speed):
  if speed > 120:
    return "You are fined for $200"
  elif speed > 100:
    return "You are fined for $100"
  else:
    return "All good, race ahead"
−−Tests−−
class Test(unittest.TestCase):
  def test_speeding_check(self):
    self.assertEquals(speeding_check(100), 'All
    ↪ good, race ahead')
    self.assertEquals(speeding_check(101), 'You
    ↪ are fined for $100')
    self.assertEquals(speeding_check(121), 'You
    ↪ are fined for $200')
```

```
"""Exercise 1
−−Keywords−−
currency
class
function
parameters
dictionary
arithmetics
−−Problem statement−−
Write a class called Converter that is initialized
    ↪ with a dictionary of exchange rates for
    ↪ currencies
against the USD, e.g. {"USD": 1, "EUR": 0.9, "GBP
    ↪ ": 0.75}. The class should have a method
    ↪ called
convert, which takes in three parameters:
    ↪ from_currency, to_currency, and amount.
    ↪ The function should
return the given amount converted from the first
    ↪ currency (first parameter) to the second
    ↪ currency
(second parameter) using the exchange rate
    ↪ dictionary given in the class constructor.

As an example, the code
converter = Converter({"USD": 1, "EUR": 0.9, "GBP
    ↪ ": 0.75})
in_euros = converter.convert("USD", "EUR", 100)
print(in_euros)
should print out 90.0
−−Sample solution−−
class Converter():
  def __init__(self, exchange_rates):
    self.exchange_rates = exchange_rates

  def convert(self, from_currency, to_currency,
    ↪ amount):
    amount_in_usd = amount / self.exchange_rates[
    ↪ from_currency]
    return amount_in_usd * self.exchange_rates[
    ↪ to_currency]
−−Tests−−
class TestConverter(unittest.TestCase):
  def test_converter(self):
    converter = Converter({"USD": 1, "EUR": 0.8})
    self.assertEquals(converter.convert("USD", "
    ↪ EUR", 100), 80)

  def test_converter2(self):
    converter = Converter({"USD": 1, "EUR": 0.9, "
    ↪ GBP": 0.75, "SEK": 9.71})
    self.assertEquals(converter.convert("USD", "
    ↪ USD", 100), 100)
    self.assertEquals(converter.convert("USD", "
    ↪ EUR", 100), 90)
    self.assertEquals(converter.convert("GBP", "
    ↪ EUR", 10), 12)
    self.assertEquals(converter.convert("EUR", "
    ↪ GBP", 10), 8.333333333333332)
```

C   COMPLETE PRIMING EXAMPLE WITH TOPIC THEMING FOR PROGRAMMING EXERCISE GENERATION

Example priming (input)

```
"""Exercise 1
--Keywords--
cars
function
parameters
conditional
--Problem statement--
Write a function called speeding_check that takes
    ↪ a single parameter speed and prints out "
    ↪ You are
fined for $200" if the speed is above 120, "You
    ↪ are fined for $100" if the speed is above
    ↪ 100 but
below 120 and otherwise prints "All good, race
    ↪ ahead".
--Sample solution--
def speeding_check(speed):
  if speed > 120:
    return "You are fined for $200"
  elif speed > 100:
    return "You are fined for $100"
  else:
    return "All good, race ahead"
--Tests--
class Test(unittest.TestCase):
  def test_speeding_check(self):
    self.assertEquals(speeding_check(100), 'All
    ↪ good, race ahead')
    self.assertEquals(speeding_check(101), 'You
    ↪ are fined for $100')
    self.assertEquals(speeding_check(121), 'You
    ↪ are fined for $200')
"""Exercise 2
--Keywords--
ice hockey
--Problem statement--
```

Generated exercise (output with temperature 0)

```
Write a function called ice_hockey_check that
    ↪ takes a single parameter
called score and prints out "You are awarded a
    ↪ penalty for unsportsmanlike conduct"
if the score is greater than 5 and otherwise
    ↪ prints "All good, keep playing".
--Sample solution--
def ice_hockey_check(score):
  if score > 5:
    return "You are awarded a penalty for
    ↪ unsportsmanlike conduct"
  else:
    return "All good, keep playing"
--Tests--
class Test(unittest.TestCase):
  def test_ice_hockey_check(self):
    self.assertEquals(ice_hockey_check(5), 'All
    ↪ good, keep playing')
    self.assertEquals(ice_hockey_check(6), 'You
    ↪ are awarded a penalty for unsportsmanlike
    ↪ conduct')
```