Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming

Majeed Kazemitabaar
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
majeed@dgp.toronto.edu

Justin Chow
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
justinchow@dgp.toronto.edu

Carl Ka To Ma
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
cma@dgp.toronto.edu

Barbara J. Ericson
School of Information, University of Michigan
Ann Arbor, Michigan, USA
barbarer@umich.edu

David Weintrop
College of Education, University of Maryland
College Park, Maryland, USA
weintrop@umd.edu

Tovi Grossman
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
tovi@dgp.toronto.edu

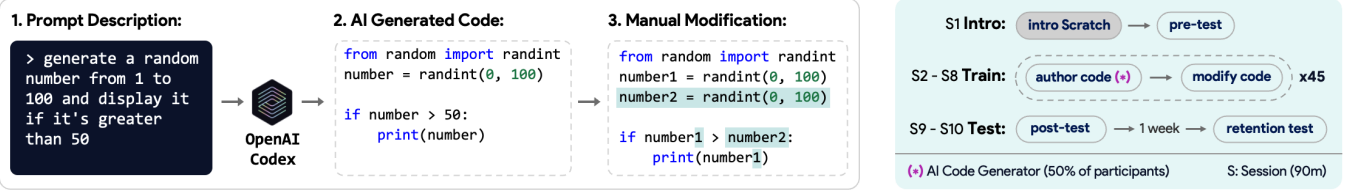

Figure 1: Left) Generate-modify usages with AI code generators. Right) Summary of our controlled study over 10 sessions.

ABSTRACT

AI code generators like OpenAI Codex have the potential to assist novice programmers by generating code from natural language descriptions, however, over-reliance might negatively impact learning and retention. To explore the implications that AI code generators have on introductory programming, we conducted a controlled experiment with 69 novices (ages 10-17). Learners worked on 45 Python code-authoring tasks, for which half of the learners had access to Codex, each followed by a code-modification task. Our results show that using Codex significantly increased code-authoring performance (1.15x increased completion rate and 1.8x higher scores) while not decreasing performance on manual code-modification tasks. Additionally, learners with access to Codex during the training phase performed slightly better on the evaluation post-tests conducted one week later, although this difference did not reach statistical significance. Of interest, learners with higher Scratch pre-test scores performed significantly better on retention post-tests, if they had prior access to Codex.

CHI '23, April 23–28, 2023, Hamburg, Germany

ACM ISBN 978-1-4503-9421-5/23/04…$15.00
https://doi.org/10.1145/3544548.3580919

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**; • **Social and professional topics** → **Computing education**.

KEYWORDS

Large Language Models, AI Coding Assistants, AI-Assisted Pair-Programming, OpenAI Codex, GPT-3, ChatGPT, Copilot, Introductory Programming, K-12 Computer Science Education

ACM Reference Format:
Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23), April 23–28, 2023, Hamburg, Germany.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3544548.3580919

1 INTRODUCTION

Powered by the recent advancements in Deep Learning [88], Large Language Models that are trained on millions of lines of code, such as OpenAI Codex [14], can generate code from natural language descriptions (Figure 1, Left). In addition to enabling natural language programming, these AI coding assistants can perform numerous operations including code-to-code operations like code completion, translation, repair, and summarization, along with language-to-code operations such as code explanation and search [58, 79]. By generating code from simple sentences instead of formal and syntactically fixed specifications, these AI Coding Assistants may lower the barriers to entry into programming.

In the context of computer science education, AI code-generators could potentially support both learners and educators. For instance, code generators could automatically fix semantic bugs and syntax errors, and allow learners to focus more on theoretical and problem-solving aspects of computational thinking [91], instead of struggling with syntax. Additionally, these tools could support educators in curriculum development by generating programming exercises and explanations to solutions [79].

However, there are also several potential drawbacks in using such tools in instructional contexts: learners might become dependent on these tools and not be able to author similar code without them; they might not know how to best express their intentions in order to generate their desired code [86] and/or, they might not understand the AI-generated code and how they could modify it if needed. From the educator's point of view, there are also concerns related to academic integrity and plagiarism [29].

Prior work has explored AI code-generators from a usability perspective with experienced programmers [86], usage barriers for novice programmers, and advantages to curriculum development for educators [79]. However, AI code generators have not been studied from a learning point of view in an introductory programming context, thus leaving essential and foundational questions unanswered. Central among them is understanding whether novices who have never written text-based code are able to understand the code generated by these tools, and if they are able to modify or extend the generated code. It is also important to understand if using such tools will form a reliance, or help learners write code without such tools being present. Specifically, we are interested in answering the following questions:

- **RQ1:** Are novices able to utilize AI code generators to solve introductory programming tasks?
- **RQ2:** How do learners' task performance measures (e.g., correctness score, completion time, and error rate) differ with and without AI code generators?
- **RQ3:** How does learners' ability to manually modify code differ with and without the use of AI code generators?
- **RQ4:** What are the effects on learning performance and retention from using an AI code generator versus not?
- **RQ5:** What is the relationship between existing programming competency and the effectiveness of using AI code generators in introductory programming?

To answer these questions, we conducted a controlled study with 69 young learners, ages 10-17 (M=12.5, SD=1.8) with no prior text-based programming experience. Half of them used an AI Coding Assistant (Codex) to learn Python, and the other half did not. Specifically, we built Coding Steps, a self-paced learning environment that included novice-friendly documentation in which learners worked on small and increasingly complex programming tasks to learn the basic concepts of Python programming. Concepts including variables, operators, data-types, conditionals, loops, and arrays. During the three-week study, learners went through three phases (Figure 1, Right): (i) a single *introduction* session to the basic concepts of programming using Scratch, followed by a pre-study evaluation, (ii) seven *training* sessions on introductory-level Python programming topics by working on 45 code-authoring tasks, each followed by a code-modification task, and (iii) two *evaluation* sessions, including an immediate post-test session, followed by a retention test conducted a week later.

Our results show that learners who had access to the AI code generator (the Codex group) were able to successfully generate code and showed evidence of understanding the generated code during the training phase. They performed significantly better on code-authoring tasks (1.15x increased progress, 0.59x less errors, 1.8x higher correctness, and 0.57x less time) with no decrease in performance on the following manual code-modification tasks, in which both groups performed similarly. Furthermore, during the evaluation phase and on the immediate post-test, learners from the Codex group were able to perform similar to the Baseline group despite not having access to the AI code generator. In the retention test which was conducted one week later, learners from the Codex group performed slightly better on coding tasks and multiple-choice questions, although these results did not reach statistical significance. Finally, our analysis indicates that learners with more prior programming competency may benefit more from AI code generators. After discussing these results, we conclude with a discussion of limitations of our work and the implications of our results.

2 RELATED WORK

Our work draws from several areas of prior research: natural language programming, AI Coding Assistants powered by large language models, and introductory programming. We review each in turn.

2.1 Natural Language Programming

Historically, many computer scientists have been arguing that programming a computer should remain a completely formal process with syntactically fixed specifications, similar to math, as it would encourage people to think more like *computers* and therefore, perform better [19]. In contrast, there are also arguments supporting the idea of bringing programming closer to how *people* think [63, 65] through natural language programming. From an HCI point of view, expressing algorithms, data manipulation tasks, and even debugging, require many transformations from how people think to how such concepts are specified in the language of the machine [64]. By reducing this gap, and utilizing principles like direct manipulation [36], the transformation effort could potentially be reduced and lighten up the burden of programming.

Generating code from natural language, instead of syntactically fixed and formal specifications, comes with many challenges like handling ambiguity and abstract language. Various methods have been employed, such as using semantic parsers [4, 9, 11, 18, 33, 34, 47, 50, 51, 56, 68, 80] and machine learning [5, 54, 71–73, 93, 95]. However, initial approaches were mostly designed to handle line-by-line explanations instead of abstract explanations.

2.2 AI Coding Assistants

Recently, there have been significant advancements in Deep Learning and Large Language Models (LLM) like GPT-3 [12]. These models have been able to generate code from natural language descriptions when they are trained on large corpora of source code. For example, models like OpenAI Codex [14], Microsoft CodeBERT [27],

Google PaLM [16], DeepMind AlphaCode [53] have been trained on millions of lines of publicly available code (e.g., on Github). In addition to generating code from descriptions, these tools can perform code completion, translation, repair, summarization, and explanation [58]. Currently, these models power AI Coding Assistants including Github Copilot [31], CodeWhisperer [2] Tabnine [85] that provide code-completion functionality.

To explore the explainability needs of AI code generators, Jiao et al. [83] conducted 9 semi-structured workshops with 43 software engineers using AI Coding Assistants and identified 11 categories of explainability needs such as types of input that the model can take, and the data that these models are trained on. In a study with 24 experienced programmers that used Github Copilot to complete real-world programming tasks, Vaithilingam et al. [86] reported that using Copilot did not improve task completion time or success rate, however, most programmers preferred to use it in their daily programming tasks as it provided a useful starting point. Jiang et al. created GenLine [39] and conducted a study with 12 participants that worked on two web-programming tasks to explore the user experience of natural language programming with AI code generators. Their results indicate that participants generally felt that they needed to learn the syntax of natural language programming. Finally, in a recent study to explore the learnability of program synthesizers, Jayagopal et al. [38] found that participants preferred program synthesis tools where users can also write code manually and liked the *triggerless initiation* and *triggerless result communication* mechanisms of Copilot. Unlike these previous studies, we are particularly interested in the impact that AI Coding Assistants have on novices when they are first learning text-based programming languages and conduct the first study comparing learning experiences with and without an AI code generator in a controlled study.

2.3 Introductory Programming

Computing is now being integrated into K-12 education of many countries [67, 90]. Research has provided evidence that computing education offers a platform to practice and improve problem-solving [28, 41, 62, 69], critical thinking [26], collaboration [77], and active learning [40] skills. One of the core components of CT is *programming*, which supports the cognitive tasks involved in CT and demonstrates computational competency [32]. A full review of introductory programming research is beyond the scope of this paper, and instead we refer the reader to recent surveys on the topic [7, 59]. Here we review introductory programming research that is most relevant to our own work: learning challenges, assistive programming environments, and AI code generators.

Programming is not easy to learn [21, 70]. Novices can be overwhelmed by the complexity of coding tasks [44, 87] and spend an unexpectedly large amount of time on them [10]. This can be a frustrating experience [78] for students and repeated failures, especially early on, can lower their self-efficacy with respect to programming [45]. Cognitive load theory defines the demand that a situation or task places on a learner's working memory [84] and is based in part on the learner's prior knowledge and the complexity of the task or material [22]. Several approaches have been developed to reduce this cognitive load in introductory programming, such as the use of worked examples and similar practice problems [74, 87], or mixed-up code (Parsons) problems [66], where learners are given mixed up fragments that must be placed in the correct order to solve a problem. Learner's typically complete Parsons problems significantly more quickly than fixing or writing code and have similar learning gains [24, 25, 94]. Similarly, AI code generators may reduce the cognitive load when learning to program, but its impact on the learning experience has not been studied.

One main approach to improving the learning experience is the use of assistive programming environments, which can help alleviate misconceptions in syntax [1, 35], and conceptual knowledge [81]. For example, many introductory programming courses for K-12 learners start with block-based programming environments (BBPEs) like LogoBlocks [8], Alice [17], Scratch [76], and AppInventor [92]. BBPEs have been designed to eliminate syntax errors and enable students to work on personally meaningful projects [75]. These characteristics lower the barrier of entry to programming and allow learners to focus on learning how to formulate a solution that a machine can execute. However, learners may perceive BBPEs to be less powerful and wish (or need) to transition to text-based programming languages. This transition, however, comes with its own challenges. To support this transition, various tools have been developed, including: dual-modality programming environments like PencilCode [6] and MakeCode [3], and hybrid environments such as Frame-based editing [48] and CodeStruct [42, 43]. In our study, we examine if AI code generators may have similar learning benefits to BBPEs and evaluate a subsequent transition to manual text programming.

With the increased availability of AI code generators like OpenAI Codex and Github Copilot, researchers have started to explore the implications of such tools on introductory programming. For example, Finnie-Ansley et al. [29] showed that OpenAI Codex outperforms most students on code writing questions that are used in typical first-year programming exams. From an educator's point of view, Sarsa et al. [79] qualitatively analyzed the novelty, sensibleness, and readiness of 120 programming exercises that were generated by OpenAI Codex after priming it with a few samples. Their results showed that the majority of the programming exercises created by Codex were sensible, but generated exercises needed some adjustments. From the learner's perspective, there are still many open questions: What happens if learners have access to AI Coding Assistants during their training? Are novices able to use these tools and understand the generated code? Will they become over-reliant on AI-generated code? We seek an answer these, and other related research questions, in this paper.

3 AI-ASSISTED LEARNING ENVIRONMENT

To investigate the effect of using AI Coding Assistants when learning to code, we built Coding Steps, a web-based application that enables learning of basic Python programming. The system allows learners to work on a series of programming tasks that were designed to gradually introduce new concepts. The system includes functionality to submit code to remote instructors that grade submitted work and provide feedback through a grading dashboard. Figure 2 shows the programming environment in Coding Steps that includes: (i) a description of the task and a set of examples, (ii)

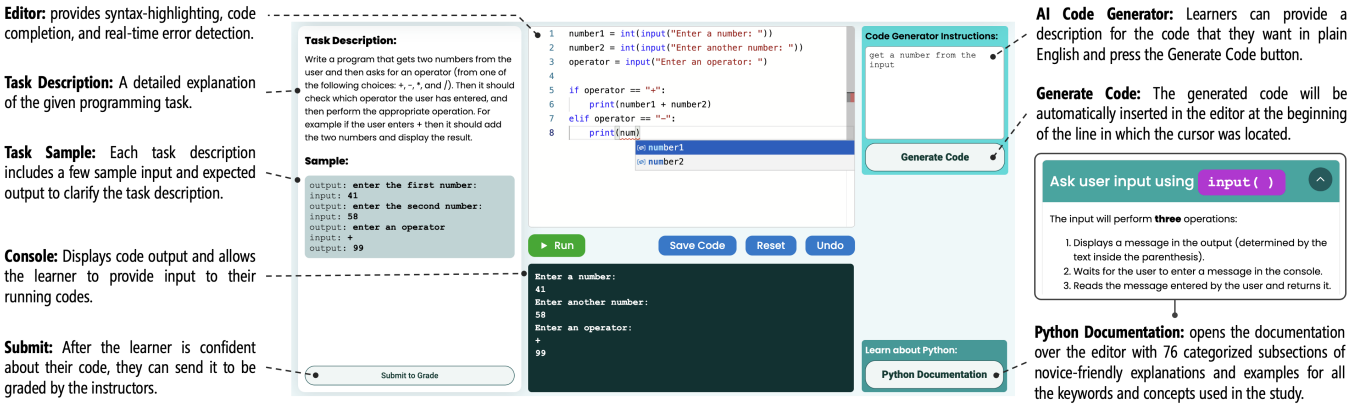

Figure 2: Coding Steps Programming environment, AI code generator, and python documentation.

a code editor with syntax highlighting, real-time error detection, and autocomplete functionality, (iii) a console to display the output and provide input to the running code, (iv) an embedded Python documentation with mini examples specifically written for novices, and (v) a code generator for inserting AI-generated code into the code editor. The integration of all these features in one single application allows learners to independently progress through the training tasks, while also supporting the collection of usage data for our study. The source code of Coding Steps is available as an open-source repository including both client and server-side code: https://github.com/MajeedKazemi/coding-steps.

3.1 Implementation

Coding Steps is written in TypeScript and has a client-server architecture that enables authentication and storing user progress, collecting logs, providing personalized feedback, executing code, and communicating with OpenAI Codex, the AI code generator used for our study. The server was implemented using Node.js, specifically: Express.js for REST API used in client-server communication, Mongoose to interact with a cloud-based instance of MongoDB for storing user progress, Passport.js for user authentication, python-shell for running Python code on the server, and a custom Python language server for real-time autocomplete suggestion and error detection. The client-side code was developed using the React Framework And included the Monaco Editor for editing code and syntax-highlighting. The Monaco Editor communicated with the Python Language Server to provide code completion, real-time error detection, definitions, hover, and references inside the editor. The Python documentation was designed as a pop-up window that would cover most of the interface. Its content was broken down into multiple subsections, each covering a specific concept of Python programming. This allowed for detailed collection of documentation usage metrics during the study.

For AI code generation, we included a textbox and Generate Code button next to the code editor (Figure 2, top right). Users could type their desired code behavior into the textbox using natural language. After clicking on the generate button, the code generated from OpenAI Codex would be inserted at the beginning of the line in which the cursor was located (and shifting any existing code below). The code generator uses the **code-davinci-002** model from OpenAI's code completion API that generates code from the provided prompt. The prompt message was modified on each API call to include the following three parts to help seed the code generation process (Figure 3): (i) six static examples of short prompt messages followed by desired output code (ii) the current code in the editor, and finally (iii) the user's requested behavior. This would condition the AI model on generating python code from simple and novice-level explanations and allows the code generator to be aware of the user's context. Coding Steps was approved by the OpenAI App Review team prior to running the study.

3.2 Data Instrumentation

The system was instrumented to collect timestamped usage data. From the code editor, all edits (including insertions, deletions, or replacements) were collected. From the console, all standard inputs, outputs, and errors were collected. From the Python documentation, all open and close events were collected from each of the subsections in the documentation. Finally, from the AI code generator, all prompt messages and generated code were collected. Data logs were periodically sent to a remote server (every 30 seconds), or upon submitting a task, to be stored in a database.

3.3 Programming Task Design

The learning environment included 45 two-part programming tasks that were designed to avoid overwhelming learners through too much cognitive load [23] by gradually increasing complexity and introducing new concepts as they made progress. This was intended to keep learners in the Zone of Proximal Development [89], which means learners were challenged to do more than they could without help. Learners' Zone of Proximal Development expands as they learn, and therefore, they can work on more complicated tasks after they master easier tasks. The tasks were organized into five groups that each focused on one specific topic in Python. Each group of programming tasks was followed by several multiple-choice questions (MCQs) for additional practice (though the correct answer was never given). See Table 1 for more information about the tasks and Appendix A for further detail on each of the task descriptions.

Table 1: Tasks used in Coding Steps broken down by topic

Topic	# Coding Tasks	# MCQs	Python Concepts
Basics	8	6	input / output, variables, basic operators, joining strings, random numbers
Data-Types	4	4	type conversions (from string to integer and vice versa)
Conditionals	8	10	conditional statements, logical expressions, comparators
Loops	18	9	iterators, loops, and conditional loops
Arrays	7	10	indexing lists, appending items to lists, obtaining the length of lists

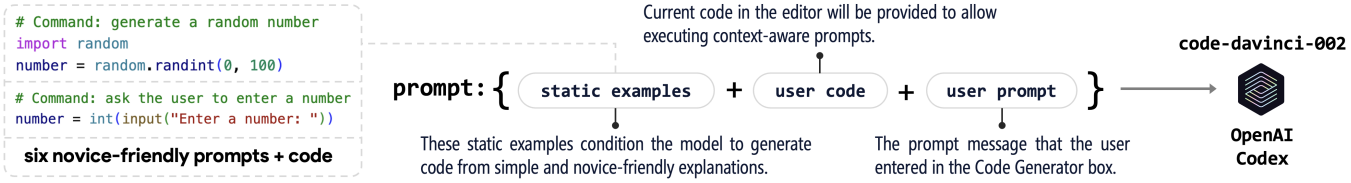

Figure 3: Prompts sent to the OpenAI Codex API were modified to additionally include current user code and six samples.

3.4 Quality of AI Generated Code

To measure the quality of the AI code generator, we entered the task description of the 45 code-authoring tasks into the pre-conditioned model (described in Figure 3) and evaluated the accuracy and amount of required manual modifications on the generated code. Codex was able to correctly solve 41/45 of the tasks with no changes to the task description. For three tasks, it required minor modifications and rewordings to the prompt message, and for one task it did not import the random module when randint was used in the generated code (however for 11/12 of the other tasks that the random module was used, Codex did correctly import the random module). This shows that the AI code generator produces high quality results, but this depends on the quality of the prompt message.

4 USER STUDY

We now present our study to investigate the impact of using AI Coding Assistants when learning to code. The study was specifically focused on novice programmers and learning introductory Python programming skills. Students used Coding Steps to learn independently throughout the study, receiving assistance from the experimenters (first three authors) when needed. Half of the students had access to the AI code generator during the training phases of the study.

4.1 Study Procedure

A between-subject design was used, with two conditions: one in which students had access to the AI code generator (Codex) and one in which the code generator was removed (Baseline). The study lasted three weeks and included ten 90-minute sessions, with one session per day, spanning over three (consecutive) weeks. The study was broken into three main phases (Figure 4): an introduction phase, a training phase, and an evaluation phase. The study was conducted remotely over the Google Meet platform.

4.1.1 Introduction Phase. The first session in the training phase was 2-hours long and included a one-hour introduction to the basic concepts of programming and computational thinking using Scratch. The lecture covered the following topics: sequence, input and output, random numbers, joining texts, arithmetic operations, conditionals, logical expressions, loops, conditional loops, and lists. During the second hour of the first session all participants worked on a pre-test including 25 multiple-choice questions about Scratch programming (Figure 6a).

4.1.2 Training Phase. The training phase consisted of seven sessions of using Coding Steps. Learners from both conditions worked on 45 programming tasks and 40 multiple-choice questions. Students progressed at their own pace through five main topics during these sessions: basics (including variables, input/output, operators, and random numbers), data-types, conditionals, loops, and lists. Learners were encouraged to ask questions from the instructors if they needed help and the instructors were trained to guide the learners to the specific parts of the documentation to resolve their issues. Additionally, learners received personalized feedback for each of their incorrect submissions that were rejected. This feedback was displayed inside the Coding Steps environment for easy access.

Learners began the training phase by watching a video that explained how the Coding Steps system worked. The Codex group watched a different video that included several examples of AI code generation. After watching the video, learners could start working on the tasks. Each programming task in the learning phase had two parts: authoring code (Figure 6b) and modifying code (Figure 6c). For both types of tasks, learners first read the task description and a few examples of input and expected output. They were then asked to write code in the code editor and were free to use the Python documentation when needed. Additionally, learners in the Codex condition had access to the AI code generator. The tasks and the topics were presented in a fixed order for all students in both groups, as they were designed to gradually increase in complexity.

The code submission and grading process is illustrated in Figure 5, Left. After running the code to check that it worked, learners submitted the code in order to advance to the next task. At this point, the researchers conducting the study would synchronously, either

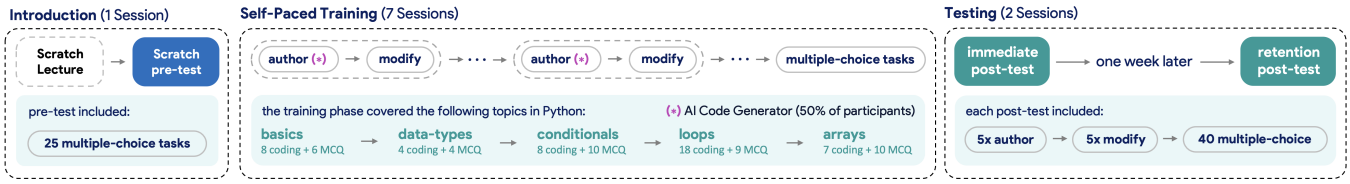

Figure 4: Study procedure over the 10 sessions (three weeks).

accept a correct solution or send an incorrect or incomplete solution back to the learner along with personalized feedback. Learners were required to continue working on the task before advancing to the next task. However, learners were also able to skip tasks after exceeding the minimum amount of time (which varied between 3 to 12 minutes based on task difficulty). If the learner skipped an authoring task, they would be shown the correct solution which was then used as the starter code for the associated code modification task. If their submission was correct, their own accepted submission was used as the starter code for the code modification task. To ensure both groups received consistent and unbiased feedback on their submissions, the grading dashboard did not display any information that would reveal the identity of learners or their group (Figure 5, Right). A first-in, first-out mechanism was used to ensure submissions were graded in order of submission. For each submitted task that was incorrect, graders were instructed to initially just explain what was wrong with the submission. However, as the number of re-submissions or the amount of time that a learner had spent working on a task increased, graders were instructed to provide more direct feedback that included some hints.

4.1.3 Evaluation Phase. The immediate post-test was conducted one day after the training phase. The Coding Steps application was still used for the evaluation phase, but learners did not have access to either the code generator or the Python documentation, or any feedback after submitting the tasks. The tasks in this phase included five code-authoring tasks, five code modification tasks, and then 40 multiple-choice questions (Figure 6d). In the code-authoring tasks, learners had to manually write all code, while in the modification tasks, starter code was provided, and learners were asked to change it. Generally, the evaluation tasks were analogous to the tasks in the training phase in terms of difficulty and topics. The second evaluation session, the retention test, was conducted one week later and included a new set of tasks and questions with a similar number and order of tasks to the immediate test.

4.2 Participants

Participants were recruited through multiple coding camps located in two major North American cities. From more than 200 sign-ups, we contacted 90 learners that reported no prior text-based programming experience to participate in the study. From the 90 participants that started the study, 69 learners (21 female/48 male) ages 10-17 (M=12.5; SD=1.8) successfully completed all three phases of the study (11 only participated in the first session, four participated in less than half of the sessions, and six missed one of the last two evaluation sessions). No common factors were identified among the 21 students who dropped out in terms of disability, native language (six non-English) and computer or internet access. Only the results from the 69 learners that completed the study are included. Participants received a $50 gift card as compensation and the study protocol was approved by our institution's Research Ethics Board. To accommodate all learners with different time constraints, three sections were offered each day of the study and learners could choose which one to join each day. The first nine sessions were conducted every day across two calendar weeks (there was no session on weekends), with the final retention post-test conducted the following week.

None of the participants reported any prior text-based programming experience, 64 indicated using a block-based programming environment like Scratch or Code.org, and 27 indicated taking a programming-related class in the past. English was the native language for 51 participants and five reported that explaining things in English was difficult for them (four of them were native English speakers). Three participants each indicated having one disability: ADHD, vision impairment, and hearing impairment. Although socioeconomic status was not asked about directly, all participants had access to a personal computer (51 indicated that they had their own computer) and 41 had at least two personal devices (e.g., a computer and a tablet or a phone).

4.2.1 Participant Condition Assignment. Following the Introduction Phase using Scratch, participants were divided into two groups using a matched-groups design [13, 60]. The ninety participants who completed the first session were divided into two groups to have similar means and variances based on their pre-test scores on Scratch. This process was used to balance the Codex and Baseline groups with regard to prior programming knowledge. One learner in each pair was randomly assigned to either the Codex or Baseline group.

Accounting for the 21 participants who did not complete the entire study, we ended up with 33 learners in the Codex group, and 36 learners in the Baseline group that finished all 10 sessions. These participants had similar means on prior Scratch programming knowledge as measured by the pre-test (Codex: M=62.7%; Baseline: M=60%; t(67)=0.54, p=.67).

4.3 Data Collection

To analyze learners coding performance, Coding Steps performed low-level instrumentation automatically, as described in Section 3.2. Additionally, demographic information was collected on the first session after the Scratch evaluation. Finally, a post-study survey was administered at the end of the study that included short answers and Likert questions about learners' perceptions about the Python

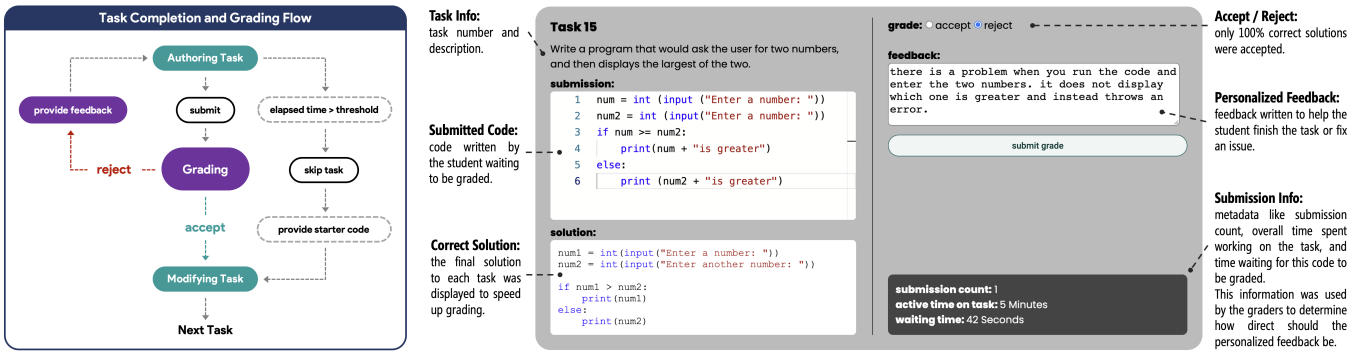

Figure 5: Left) Task completion and grading flow during the training phase with Coding Steps. Right) The grading UI that was used in the grading dashboard which includes anonymous information for each code submission.

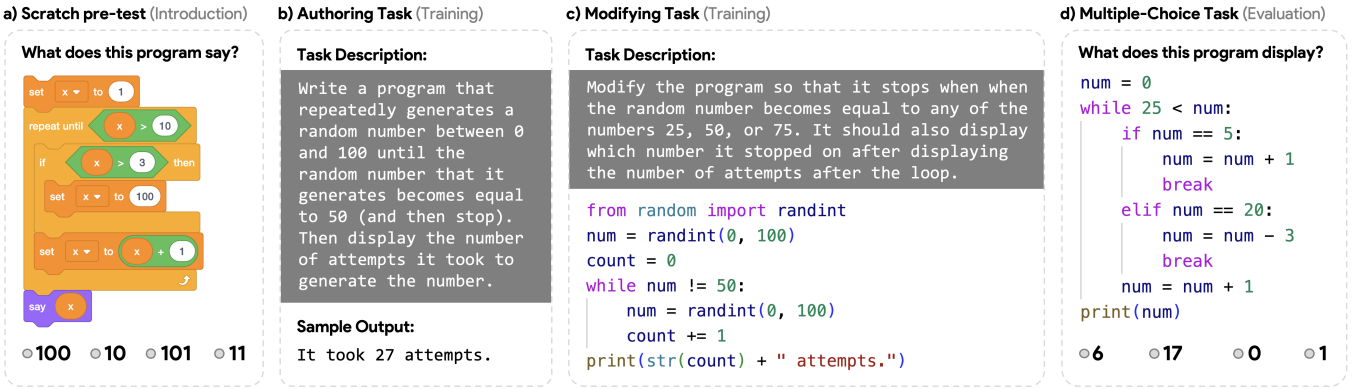

Figure 6: Sample tasks used in the pre-test, training, and evaluation phases.

documentation, learning gains, confidence, and several questions about the code generator exclusively in the Codex group.

4.4 Data Analysis

For calculating correctness scores on the coding tasks, two independent researchers graded each submitted solution using a simple and consistent grading rubric of deducting 25% for each major issue or missing part in their final submission (0%, 25%, 50%, 75%, and 100%). The two graders fully agreed on 79% of the submissions, with 16% of disagreements having a difference of 25% in which we averaged the two grades. For the 5% of the disagreements which were more than 25%, the two graders met again to resolve each of their disagreements. A similar approach was used for checking if any of the personalized feedback provided during the training phase could be counted as a direct hint or not.

Here we define the metrics that we used to analyze learners' performance and behavior while as they worked on the programming tasks. Our logging system enabled us to define various metrics that were all computed programmatically which can be seen in Table 2.

Overall Training Metrics: to measure overall task completion rate, we simply divided the number of tasks a learner completed or skipped, by the total number of tasks in the training phase as all learners went through a fixed order of tasks. To measure how much feedback learners received during the training phase, we quantitatively analyzed their length (in characters), count, and qualitatively analyzed whether they could be counted as a direct hint or not.

Task Performance Metrics: to measure task completion time, we calculated the active time a learner spent working on a task by removing inactivity gaps of longer than one minute. To check if a learner referenced the documentation during a task, we analyzed the open and close events on the subsections of the documentation. Furthermore, to categorize and count encountered errors during each task, the Python shell logs were scanned for three major types of errors: syntax, data-types, and semantic errors. Syntax errors consisted of issues related to indentation, mismatched identifiers, missing or mismatched quotes and parenthesis, incorrect or missing imports, missing or extra arguments in function calls, and general syntax errors. Data-type errors consisted of type mismatch errors between variables or literal values. And finally, semantic errors included infinite loops, array indexing errors, and division by zero errors.

AI Code Generator Usage Metrics: to measure what percentage of the code for each task was written by the learner or the AI code generator, we used the Jaccard text similarity coefficient [37] between each line of the final submission and the code generated by Codex, averaged over all lines. To measure what percentage of the prompt description was written by the learner themselves, or if

Table 2: Definition, unit, source, and calculation method for each of the metrics.

Overall Training Metrics	**Definition and Source**
Completion Rate (percentage)	*Definition*: How far a learner progressed through the training phase regardless of correctness of tasks or skips: number of seen tasks divided by total tasks count.
Personalized Feedback (count)	*Definition*: Total number of personalized feedbacks a learner received during the training phase. *Source*: code submission logs.
Feedback length (characters)	*Definition*: The length (number of characters) of the personalized feedback a learner received during the training phase. *Source*: code submission logs.
Direct hints (count)	*Definition*: Total number of personalized feedbacks a learner received that included direct hints towards solving the problem. *Source*: code submission logs.
Per-Task Performance	**Definition and Source**
Coding Correctness Score (percentage)	*Definition*: How correct was a learner's solution to a single task. *Source*: Final submission in the submissions log that was graded independently by two researchers.
MCQ Correctness Score (percentage)	*Definition*: Whether a learner responded correctly to a multiple-choice question. *Source*: submission logs.
Completion Time (seconds)	*Definition*: Active time a learner spent working on a task (by removing inactivity gaps of longer than one minute). *Source*: aggregated logs.
Documentation Referenced (count)	*Definition*: Whether a learner referenced the Python documentation for a task or not. *Source*: documentation logs.
Encountered Errors (count)	*Definition*: Number of errors a learner encountered after running their code categorized into syntax, data-type, and semantic errors. *Source*: console logs.
AI Code Generator Usage	**Definition and Source**
Code Generator Usage Per Task (count)	*Definition*: Number of unique prompts and codes generated using the AI code generator during a single task. *Source*: code generator logs.
AI-Generated Code Ratio (percentage)	*Definition*: The percentage of code in a task that was generated by an AI code generator, as opposed to being written manually by the learner calculated using the Jaccard text similarity coefficient [37]. *Source*: code submission logs and code generator logs.
Tasks Broke Down into Subgoals (count)	*Definition*: Whether different parts of the final submission for a task was generated from different codex usages. *Calculation Method*: averaging over the maximum hamming distance between each line of the final submission and each line in the codex generated codes. *Source*: code submission logs and code generator logs.
Prompt Similarity with Task Description (percentage)	*Definition*: Similarity between the prompt used for generating code and the task description. *Source*: code generator logs and task descriptions.

it was simply copied from the task description, we used the same Jaccard text similarity coefficient.

5 RESULTS

In this section we examine how each group performed in each of the three phases of the study. For each reported metric, we report means, standard deviations, and 95-percentile confidence intervals for each condition. An independent samples t-Test with an alpha level of 0.05 was used to determine whether there was statistical evidence that the associated population means between the two conditions were significantly different. A Bonferroni adjusted alpha level was used when performing multiple analyses at the topic level (five topics, $\alpha = 0.01$) or error type level (three error types, $\alpha = 0.016$). Finally, Cohen's d is reported for effect size [82]. We also report qualitative feedback from learners about the training phase and how learners in the Codex group felt about using the AI code generator. We report Mann-Whitney U Test to analyze statistical differences between the two groups on the Likert-scale questions, with an alpha level of 0.05 for statistical significance.

5.1 Training Phase

In this section we report how the two groups progressed and performed differently on three major types of tasks during the training phase: authoring, modifying and multiple-choice tasks. Figure 7 illustrates a summary of all results in this phase.

5.1.1 Overall Completion Rates and Direct Hints. In terms of overall progress in the training phase (Figure 7a), learners from the Codex group finished significantly more tasks compared to the Baseline group (Codex: M=90.9%, SD=16.7%, Baseline: M=79.0%, SD=18.6%, t(67)=2.8, p=.006, d=0.68). Particularly, learners from both groups fully completed the first two topics (basics and data-types) and one learner from each group did not complete conditionals. Additionally, the Codex group had more learners that fully completed the 18 tasks on loops (Codex: 27/33, Baseline: 23/36) and the 7 tasks on arrays (Codex: 23/33, Baseline: 9/36). In terms of personalized feedback during the training phase, learners in both groups received similar number of personalized feedback (Codex: M=0.33, SD=0.27; Baseline: M=0.31, SD=0.20; t(67)=0.25, p=.805, d=0.06) with

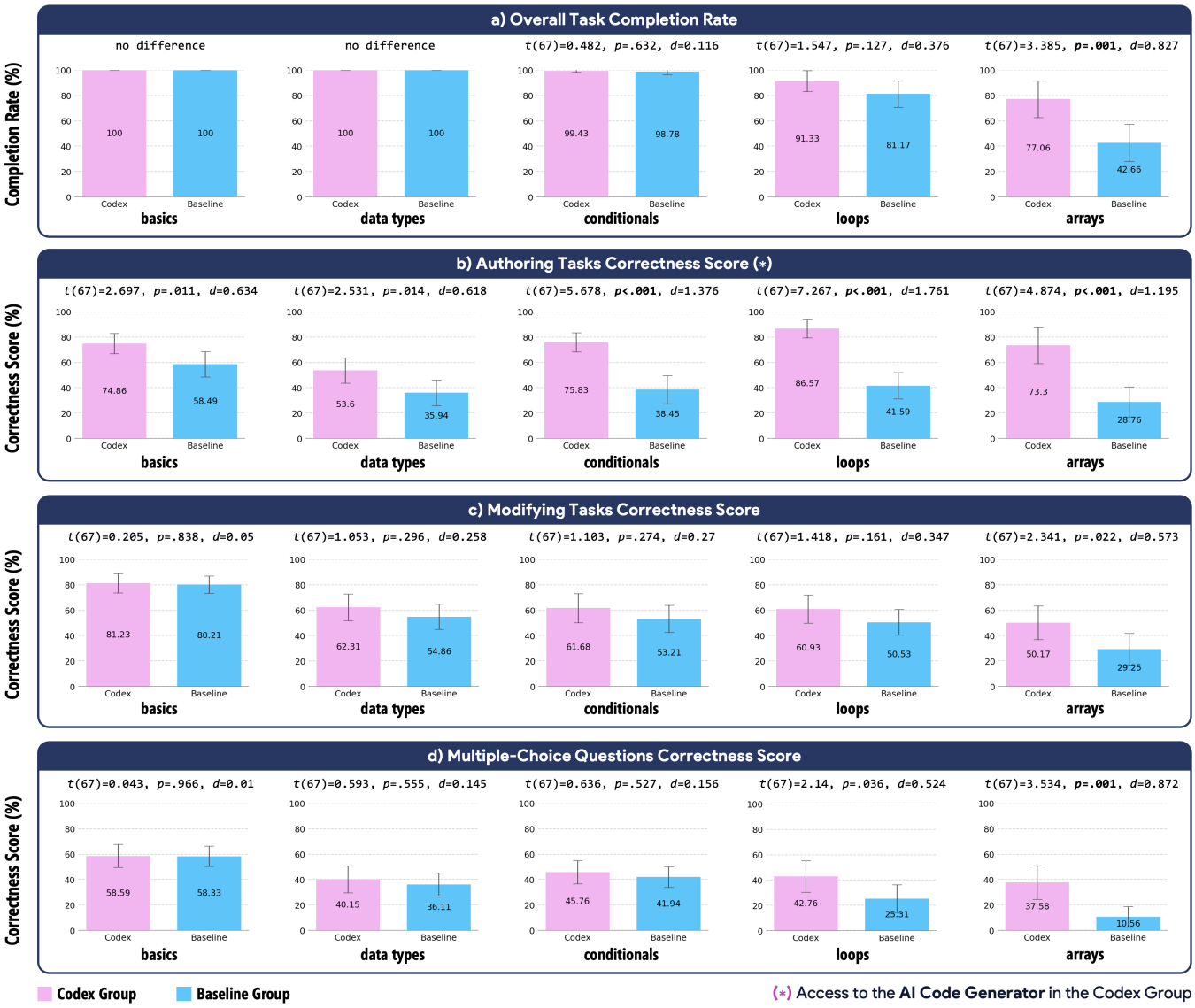

Figure 7: Correctness score of tasks during the training phase broken down by topic. Alpha levels for per-topic comparisons are adjusted using Bonferroni correction ($\alpha = 0.05/5 = 0.01$).

no difference in feedback length (Codex: M=65.8, SD=13.7; Baseline: M=67.5, SD=12.0; t(67)=0.53, p=.60, d=0.13) and no significant difference in direct hints (Codex: M=2.9, SD=2.6; Baseline: M=3.7, SD=2.8; t(67)=1.15, p=.25, d=0.28).

5.1.2 Authoring Tasks (Training Phase). On authoring tasks during the training phase, which was the only time that the Codex group had access to the AI code generator, correctness score was significantly higher for the Codex group (Codex: M=80.1%, SD=14.5%; Baseline: M=44.4%, SD=26.5%; t(67)=6.92, p<.001; d=1.67). Additionally, task completion time was significantly less in the Codex group (Codex: M=210s, SD=99s; Baseline: M=361s, SD=95s; t(67)=6.40, p<.001, d=1.56). See Figure 7b for more details per topic.

Furthermore, our analysis on documentation usage indicates that the Codex group accessed the documentation significantly less than the Baseline group for the authoring tasks (Codex: M=22.1%, SD=21.3%; Baseline: M=54.3%, SD=26.5%; t(67)=5.48, p<.001, d=1.33).

Learners in the Codex group also encountered significantly fewer errors per task (Codex: M=1.28, SD=1.1; Baseline: M=2.17, SD=1.31; t(67)=3.03, p=.004, d=0.74). Most of these errors were syntax errors that occurred significantly less for learners in the Codex group (Codex: M=0.87, SD=0.77; Baseline: M=1.67, SD=1.01; t(67)=3.65, p=.001, d=0.88) followed by data-type errors in which there were no meaningful differences between the two groups (Codex: M=0.30, SD=0.34; Baseline: M=0.39, SD=0.35; t(67)=0.98, p=.331, d=0.24). However, semantic errors occurred infrequently in both groups

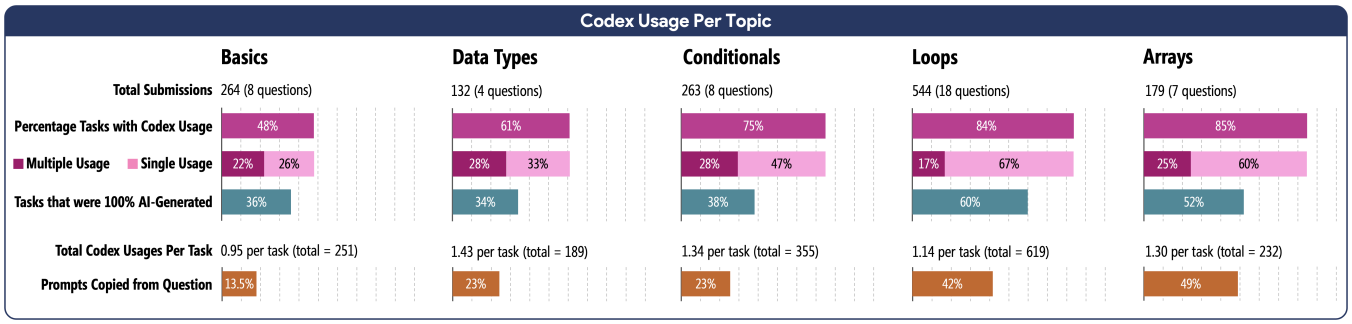

Figure 8: Statistics of tasks in which the AI Code Generator was used broken down by topic.

(Codex: M=0.01, SD=0.02; Baseline: M=0.03, SD=0.05; t(67)=1.84, p=.071, d=0.44).

5.1.3 AI Code Generator Usage. In this section we report AI code generator usage and metrics in the Codex group. This data sheds some initial light on how participants incorporated the AI code generator into their workflow; additional insights are provided from qualitative feedback presented in Section 5.4.

A summary of how learners used the AI code generator is illustrated in Figure 8. Learners used the AI code generator an average of 1.21 times per task (SD=0.66). They did not use it at all for 26.5% of the submitted authoring tasks (367/1382) and it was used more frequently on later topics (48% on basics compared to 84% on loops and arrays). Possible explanation for not using the AI code generator include their unfamiliarity with the interface or having confidence to write the code manually. Furthermore, learners used the code generator multiple times for 22.4% of the tasks (310/1382). This could indicate that either learners needed to refine their prompt message to generate better code or that learners tried to complete the task incrementally. Follow up analysis of the submitted code indicated that in 12.9% of the tasks (179/1382), learners broke down the task into multiple subgoals, and used the AI code generator to produce code for each requested subgoal. Future, more in-depth studies are required to reveal learners' intentions and thought processes while using AI code generators.

To further understand how learners interacted with the AI code generator, we also analyzed each of the AI code generator prompt messages (n=1646). In particular, we found that there was a 41% (SD=32%) similarity between the prompts and the task descriptions. We also found that 32% (n=530) of the code generator prompts were exactly the same as the task description. This indicates that learners sometimes copied the task description into the code generator's textbox.

Across all the submitted tasks in which the AI code generator was used, the final solution was on average 63% (SD=42%) similar to what was generated by the AI code generator. Learners submitted the AI-generated code without any modification for 49% of the tasks (503/1015). This particular pattern that occurred more frequently on loops (60%) and arrays (52%), usually happened when the learner copied the task description and asked the AI code generator to generate the solution to the whole task. However, these patterns were not consistently used across learners. In fact, 10/33 learners used this pattern for less than 4/45 of the tasks, while 14/33 learners used it for more than half of the tasks. Future qualitative analysis is required to explore ways in which learners break down tasks and write prompts for the AI code generator, or how different usage patterns impacts learning.

5.1.4 Modifying Tasks (Training Phase). In the Modifying tasks, both groups were provided a functioning program and then were asked to modify it, with neither group having access to the AI code generator. The starter code in modifying tasks was the solution to the previous authoring tasks. We were particularly interested to see if leveraging AI code generators would be detrimental when it came to modifying existing code.

Our results (Figure 7c) show that both groups had similar correctness scores on modifying tasks (Codex: M=66.2%, SD=25.6%; Baseline: M=58.4%, SD=23.9%; t(67)=1.28, p=.202, d=0.31). In fact, learners in the Codex group on average scored 20.8% higher on arrays in terms of correctness score (Codex: M=50%, SD=36.8%; Baseline: M=29.2%, SD=36.3%; t(39)=2.34, p=.022, d=0.57) and 10.5% higher on loops (Codex: M=61%, SD=30.5%; Baseline: M=50.5%, SD=29.3%; t(58)=1.41, p=.161, d=0.34) however, neither of these results reached statistical significance. The higher completion rate on arrays in the Codex group might potentially explain this difference. But overall, this is a promising result: although learners in the Codex group used the documentation less and relied heavily on AI-generated code for the authoring tasks, they still did just as well, and in some cases better, in manual code modification tasks.

Both groups had similar documentation access rates per task (Codex: M=13.4%, SD=10.5%; Baseline: M=17.7%, SD=12.3%; t(67)=1.53, p=.129, d=0.37) and similar overall errors encountered per task (Codex: M=1.06, SD=0.80; Baseline: M=1.24, SD=0.75; t(67)=0.91, p=.367, d=0.22).

5.1.5 Multiple-Choice Tasks (Training Phase). Both groups performed similarly on the multiple-choice questions (Figure 7d) during the training phase (Codex: M=48.8%, SD=23.5%, Baseline: M=45.2%, SD=20.2%, p=.510). Looking at the correctness score on multiple-choice questions broken down by topic, we see that both groups did comparably well on topics related to basics, data-types, conditionals, and loops, however, on arrays, the Codex group performed significantly better (Codex: M=37.6%, SD=36.8%; Baseline: M=10.5%, SD=23.7%; t(39)=3.53, p=.001, d=0.87).

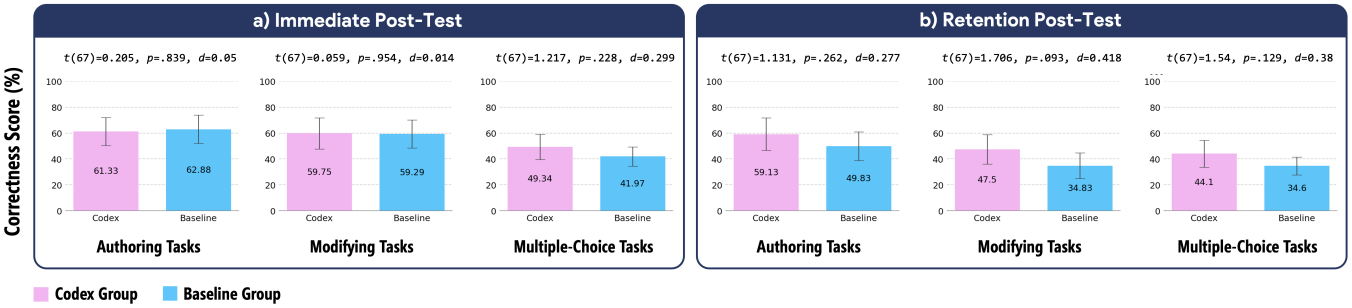

Figure 9: Correctness score of tasks during the evaluation phase.

5.2 Evaluation Phase

For our analysis of the evaluation phase, we excluded tasks related to topics for which the learner progressed less than 50% during the training phase.

5.2.1 Immediate Post-Test. Learners in both groups performed similarly on all three types of tasks in the immediate post-test (Figure 9a) that was conducted a day after the training phase. Both groups performed similarly in terms of correctness score on authoring tasks (Codex: M=61.3%, SD=30.1%; Baseline: M=62.9%, SD=32.0%; t(67)=0.20, p=.838, d=0.05) and modifying tasks (Codex: M=59.7%, SD=33.4%; Baseline: M=59.3%, SD=31.6%; t(67)=0.058, p=.953, d=0.01). There were no meaningful differences on task completion times and error-rates on both authoring and modifying tasks between the two groups. Furthermore, on multiple-choice questions, overall, both groups scored comparably (Codex: M=49.3%, SD=27.3%; Baseline: M=42.0%, SD=21.6%; t(67)=1.21, p=.228, d=0.30) while the Codex group performed significantly better on arrays (Codex: M=32.2%, SD=31.8%; Baseline: M=13.2%, SD=22.8%; t(39)=2.78, p=.007, d=0.68) and on average 16% higher on loops, but this did not reach statistical significance (Codex: M=50.2%, SD=35.7%; Baseline: M=33.8%, SD=27.7%; t(58)=2.08, p=.041, d=0.51).

5.2.2 Retention Post-Test. The retention post-test was administered a week later to understand if AI code generators would be detrimental to learners' ability to retain knowledge and skill. In terms of correctness score, our results (Figure 9b) show that learners from the Codex group on average scored 9.0% higher on authoring tasks (Codex: M=59.1%, SD=34.8%; Baseline: M=49.8%, SD=32.4%; t(67)=1.13, p=.262, d=0.28), and 12.7% higher on modifying tasks (Codex: M=47.5%, SD=31.6%; Baseline: M=34.8%, SD=29.0%; t(67)=1.70, p=.092, d=0.41), but neither of these reached statistical significance. Similar to the immediate post-test, the two groups had similar completion times on both authoring and modifying tasks. However, on error-rates, learners in the Codex group encountered significantly more errors on authoring tasks (Codex: M=1.58, SD=1.34; Baseline: M=0.99, SD=0.92; t(67)=2.08, p=.042, d=0.51) that no starter code was provided. Learners in the Codex group also encountered slightly higher errors on modifying tasks, but this did not reach statistical significance (Codex: M=0.81, SD=0.85; Baseline: M=0.57, SD=0.70; t(67)=1.21, p=.229, d=0.30). This increased error-rate in the Codex group could potentially be explained by the significantly higher percentage of tasks that were skipped by learners in the Baseline group compared to the Codex group (Codex: M=14%, SD=35%, Baseline: M=33%, SD=47%, t(67)=4.10, p<.001, d=0.44).

For multiple-choice questions, overall both groups performed comparably well (Codex: M=44.1%, SD=28.9%; Baseline: M=34.6%, SD=20.3%; t(67)=1.54, p=.129, d=0.38) while learners in the Codex group scored on average 16% higher on loops (Codex: M=40.6%, SD=34.5%; Baseline: M=24.7%, SD=26.6%; t(58)=2.09, p=.04, d=0.51) and 14% higher on arrays (Codex: M=30.9%, SD=32%, Baseline: M=16.7%, SD=24.7%, t(39)=2.02, p=.047, d=0.49), although neither of these differences reached statistical significance.

5.3 The Effect of Prior Programming Competency

To explore how prior programming competency affects learning performance with and without access to AI code generators, we divided learners into two groups based on their pre-test scores in the introduction phase. The Codex-High (CH: n=16, M=86.1%, SD=10%) and Baseline-High (BH: n=18, M=82.6%, SD=12%) were learners that scored higher on the pre-test, and the Codex-Low (CL: n=17, M=42.1%, SD=18%) and Baseline-Low (BL: n=18, M=37.3%, SD=18%) were learners that scored lower on the pre-test.

The results of comparing each measure within these groups are provided in Figure 10. The comparison is showing that most of the differences between conditions (in terms of effect size) are appearing on the left side of this table, for the high performers. That is, learners in the Codex-High group performed significantly better than the Baseline-High group on multiple measures during the retention post-tests, while the Codex-Low and Baseline-Low groups had nearly similar performance levels, except on authoring tasks during the training phase. One potential explanation is that learners in the Codex-High and Baseline-High groups knew more about the fundamental concepts (e.g., conditionals and loops) and therefore, using the AI code generator provided the scaffolding needed to transition that knowledge from Scratch to Python, which helped them to perform significantly better compared to the Baseline-High group. More research is warranted to further tease out this potential effect.

5.4 Qualitative Feedback

We inquired learners' perception on learning, stress, and discouragement during the training phase and their eagerness about future

	Codex-High	Baseline-High	Comparison	Codex-Low	Baseline-Low	Comparison
Scratch Pre-Test						
MCQs	*M*=86%, SD=10%	*M*=83%, SD=12%	p=.332, d=0.34	*M*=42%, SD=18%	*M*=37%, SD=18%	p=.446, d=0.27
Training						
Authoring Tasks	*M*=90%, SD=8%	*M*=59%, SD=24%	p<.001, d=1.71	*M*=71%, SD=13%	*M*=29%, SD=20%	p<.001, d=2.47
Modifying Tasks	*M*=81%, SD=16%	*M*=73%, SD=20%	p=.179, d=0.48	*M*=52%, SD=25%	*M*=44%, SD=19%	p=.311, d=0.36
MCQs	*M*=66%, SD=17%	*M*=55%, SD=20%	p=.110, d=0.58	*M*=33%, SD=17%	*M*=36%, SD=15%	p=.617, d=0.17
Immediate Post-Test						
Authoring Tasks	*M*=81%, SD=13%	*M*=76%, SD=30%	p=.566, d=0.20	*M*=43%, SD=30%	*M*=50%, SD=27%	p=.505, d=0.23
Modifying Tasks	*M*=82%, SD=14%	*M*=76%, SD=23%	p=.418, d=0.28	*M*=38%, SD=32%	*M*=42%, SD=29%	p=.758, d=0.10
MCQs	*M*=71%, SD=18%	*M*=53%, SD=20%	p=.011, d=0.95	*M*=28%, SD=17%	*M*=31%, SD=16%	p=.706, d=0.13
Retention Post-Test						
Authoring Tasks	*M*=86%, SD=15%	*M*=63%, SD=30%	p=.011, d=0.95	*M*=34%, SD=29%	*M*=36%, SD=28%	p=.805, d=0.09
Modifying Tasks	*M*=71%, SD=21%	*M*=49%, SD=27%	p=.015, d=0.90	*M*=26%, SD=23%	*M*=21%, SD=24%	p=.569, d=0.20
MCQs	*M*=65%, SD=24%	*M*=44%, SD=22%	p=.014, d=0.93	*M*=24%, SD=17%	*M*=25%, SD=13%	p=.856, d=0.06

Figure 10: Correctness scores and completion rate of each quartile (based on pre-test scores and access to AI code generator). Heat-map of statistics is colored based on effect size value (greener colors indicates higher effect sizes)

computing education using several Likert-scale questions. We also asked learners in the Codex group several Likert-scale questions specifically about the AI code generator and a few open-ended questions on their likes and dislikes about it. A summary of the responses to the Likert-scale questions is illustrated in Figure 11.

5.4.1 Perceptions on Learning, Stress, and Eagerness. As shown in Figure 11a, both groups felt they learned about Python programming and its concepts during the training phase. However, on stress and discouragement, learners in the Codex group felt slightly less stressed (U=390.5, p=.056). Some learners from the Codex explicitly attributed their reduced stress to using the AI Code Generator. For example, P26 reported "*using the code generator helped me save time and reduced pressure*". Additionally, learners from the Codex group felt more eager and excited to continue learning about programming after the study (U=692, p=.025).

5.4.2 Perceptions on Using AI Code Generator. On questions about using the AI code generator (Figure 11b), learners in the Codex group generally felt that using the code generator was easy to use. For example, P4 reported "*you could ask anything you want to the generator, and it would turn it into code*". They also felt that generating the code that they needed did not require a lot of practice. P12 reported "*I liked how you just had to use regular sentences like how I'm typing right now*". Furthermore, they also did not feel the need to change many things in the AI-generated code to make it work. For instance, P19 reported "*You could sometimes do an entire task without writing any code*" and P25 reported "*I liked how it presented code based on the words the user had entered in the input box, generating the necessary code lines and matching it with the given variables*". Additionally, several learners explicitly mentioned that they used the code generator whenever they were stuck. For instance, P29 reported "*I liked the fact that if you were stuck, you could ask it for some code*". A few learners also reported some difficulties. For example, P3 reported "*sometimes you had to be more descriptive, and you had to say things almost to point*" and P14 reported "*Sometimes I felt like it would have been easier writing the code instead of making the code generator do it*".

Furthermore, learners mostly felt that they learned about Python programming concepts after using the code generator. For instance, P13 reported "*it made it easier to learn certain applications of code such as asking it to determine if a number was even*", and P19 reported "*It would write the code for you, and you could study it if you didn't know how*". Several learners also mentioned that they did not like that the code generator doing the task for them without having them put any effort in the task. For example, P25 reported "*What I didn't like about it was that it was giving the direct answer to the user, instead of step-by-step hints to the user. It should give the user time to think about the problem / code before seeing the response*". Participants also mostly felt that using the code generator was similar to asking the instructor for help. P24 reported "*the code generator was a way for me to solve my own problems without having to turn to someone*". Some learners felt that the AI-generated code needed some explanations, for example, P18 reported "*I never got an explanation on why the generated code was what it was*". Finally, learners reported mixed responses when they were asked if using the AI Coding Assistant felt like cheating.

6 DISCUSSION

Our results show initial promise in using AI code generators in introductory programming settings. We reflect further on each of our research questions below.

RQ1: *Can novices use AI code generators?* Both our quantitative and qualitative results show evidence that overall, novice learners are able to use AI code generators to successfully solve tasks. Other than 32% of the tasks in which learners wrote the prompt description with almost no effort, they almost always actively tested the AI generated code before submission. Learners employed various methods to generate code such as breaking down the task into subgoals and using the code generator to generate code for each of the subgoals step-by-step.

RQ2: *How does learners' task performance differ with and without AI code generators?* Our results from code-authoring tasks during the training phase indicate that using AI code generators can significantly increase task completion, improve correctness score, reduce encountered errors, and reduce task completion time. Furthermore, our qualitative feedback shows that having access to the AI code

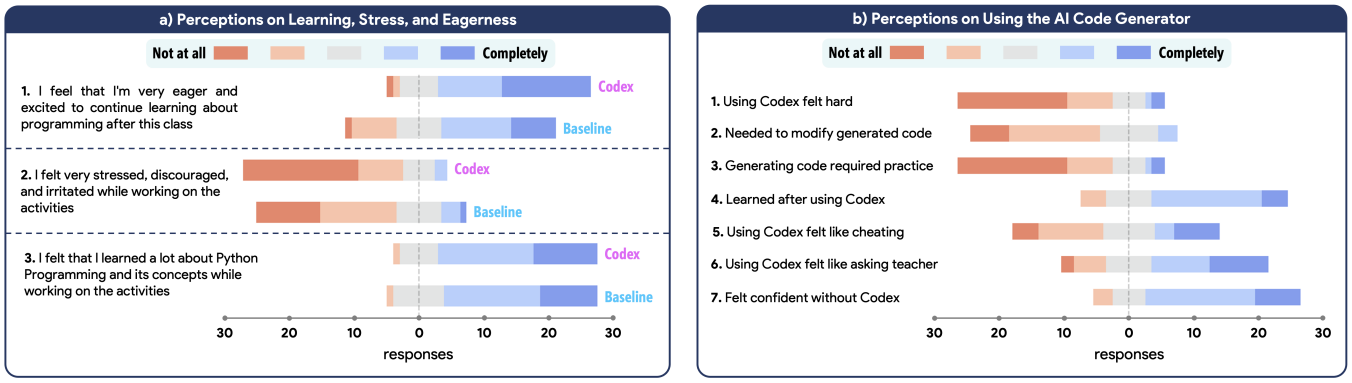

Figure 11: Aggregated responses to the Likert-scale questions.

generator reduced stress and discouragement in addition to improving their eagerness to continue programming later on.

RQ3: *How does learners' ability to manually modify code differ with and without AI code generators?* Our results from code-modification tasks during the training phase show that prior access to AI code generators did not reduce learners' ability to manually modify code afterwards. In fact, we noticed improvements in code modification skills on tasks about arrays.

RQ4: *What are the effects on learning performance and retention from using AI code generators versus not?* Our results from the immediate post-test and retention post-test show that having access to an AI code generator does not impede learning gains. In fact, we noticed that learners who performed better on the Scratch pre-test (who presumably had more prior programming knowledge) gained a significant improvement in the retention post-test after having access to the AI code generator during the training phase. Finally, it is important to note that the training experience used in our study limited the usage of AI code generators to code-authoring tasks, and learners might have potentially learned about Python programming concepts while working on the code-modification tasks.

RQ5: *What is the relationship between existing programming competency and the effectiveness of using AI code generators in introductory programming?* Our results show that learners with higher pre-test scores benefitted more from the AI code generators compared to those who scored lower. Particularly, learners with higher prior programming competency performed significantly better on the retention test if they had access to the AI code generator during their training phase. That being said, learners who scored lower on the pre-test still performed significantly better with the AI code generator when authoring code in the training phase. This shows that AI code generators can still be an important tool to minimize frustration for even the most inexperienced users.

6.1 How Novices Benefit from AI Coding Assistants

The benefit of AI code generators for novice learners could be explained by the effective employment of the use-modify-create pedagogical strategy often used in introductory programming [30, 52]. Although learners in the Baseline group used the documentation more frequently, they had to start each task by creating a new program from scratch before getting to the modify portion of the activity, and thus encountered more errors. However, learners from the Codex group had the advantage of using code that was generated specifically for an intended behavior that they wrote. This meant that the AI Coding Assistant turned the Create task into a Use-Modify task as they were provided with functional pieces of code in response to their written description which they could then modify. Therefore, they were able to spend more time to trace, test, and learn about the structure of the generated code before moving on to modifying it in the next task. Prior work in introductory programming has shown that code-reading and tracing skills are essential to problem-solving activities including code-writing [49, 55, 57]. This explanation also sheds light onto why learners with higher prior programming competency benefitted more from the AI code generator. These learners had the opportunity to transform their prior conceptual knowledge of variables, conditionals, loops, and lists from Scratch to Python by utilizing the use-modify-create method. Their higher prior programming knowledge meant that they probably had more code-reading and tracing skills which means that they were able to understand that code better, and then perform better during the next modification or code-creation tasks.

Furthermore, some learners in the Codex group broke down the task into multiple subgoals and then used the AI code generator to generate the Python code that would execute that particular subgoal. This way, learners had the chance to learn about the specifics of Python programming step-by-step instead of being overwhelmed with a large amount of code. Prior research has shown that subgoal-labeled instructional material reduces extraneous cognitive load imposed on novices learning programming by chunking problem-solving steps and promoting self-explanation [61]. This could potentially explain how step-by-step usage of the AI code generator could potentially reduce cognitive load and allow novices to learn faster due to more availability of their mental resources, specifically for germane cognitive load which is responsible for creating mental models and storing long-term memory [46].

6.2 Implications for Design

6.2.1 Support Complete Beginners. AI Coding Assistants could include a few features to support learners with less or no prior

programming knowledge and experience. From a tool-design perspective, learners could become overwhelmed when the AI code generator produces a huge chunk of code based on their generated description. Instead, AI code generators for novices could divide the generated code into multiple segments (based on their semantic block in the abstract syntax tree) and allow the learner to spend time on each segment separately before inserting the next segment. Furthermore, these tools could accompany each of the generated code segments with line-by-line code explanations using the code-to-language capabilities of AI Coding Assistants. Additionally, documentations and worked examples could accompany the generated code, based on the keywords and programming patterns used in each code segment.

6.2.2 Control for Over-Utilization. One of the main concerns with the code generation capabilities of AI Coding Assistants is that they might impede learning as a result of over-utilization. Although, our results show that this might not necessarily be problematic for learners with less prior programming knowledge as they did *not* perform worse compared to the Baseline group, it would not be advantageous to their learning. However, these tools could include constraints such as not allowing the learner to use the generated code in the editor before they actively engage in completing a mini-task based on the generated code. For example, this mini-task could be a Parsons problem that is automatically produced from the AI-generated code. Or a multiple-choice question that uses the same concepts in the generated code (e.g., two nested loops). Such interactions prior to using the AI-generated code could provide new possibilities for active learning and improve conceptual understandings in a personalized way.

6.2.3 Support Writing Prompts. When working with AI code generator, sometimes learners felt that they needed to provide a high amount of detail in the prompt description in order for the tool to generate their desired code. In these moments, they indicated that it would have been easier if they wrote the code for that task themselves. Learners could benefit from viewing a history, or a list of common phrases while they are typing the prompt to speed up the writing process (similar to automatic code completions that are used in code editors). Furthermore, novices could benefit from an interactive, dialogue-based code generation tool in which instead of writing the whole task, they would write the task through a series of smaller prompts. Similar to the implementation used in Coding Steps, such code generation bots could be context aware and allow the learner to execute commands like "*if the value of variable X is greater than Y then call function F*".

7 LIMITATIONS AND FUTURE WORK

The results presented in this study used correctness score as one of the main variables to compare the performance of the two conditions on both coding tasks and multiple-choice questions. Particularly on coding tasks, correctness is a quantifiable variable that we defined based on our specific rubric and grading scheme. For example, on code-modification tasks during the training phase, when there was a difference of 20% on correctness between the two groups on arrays, it means that on average learners in one group had about one fewer substantial problem per task on arrays. We could see that using more detailed rubrics such as grading at a subgoal level per submission could provide more tangible scores, however, for the sake of simplicity we decided to use the simple rubric presented in this study.

Our results show that the difference in correctness score increased at later topics such as loops and arrays. This could be due to learning effects, as learners became more familiar with using the AI code generator. However, it could also indicate that AI code generators are most useful for more complex topics. Using a fixed order of topics was needed to provide a scaffolded learning experience. Further research could be conducted to carefully study the impact that task complexity has on the benefits of AI code generators.

When discussing our results, there were several metrics in which the differences between the two groups did not reach statistical significance but there did seem to be some difference in means and confidence intervals. The lack of statistical findings for such metrics does not necessarily mean there is no effect present, it could potentially be due to the sample size of our study. Future studies with larger sample sizes, perhaps in real classroom deployments, could provide more definitive findings. As always, statistical findings should be considered with caution, and prior research has even argued for switching away from statistical significance testing in favor of reporting informative charts with effect sizes and confidence intervals and nuanced interpretations [20], which we have also tried to provide. While out of scope for our current work, we hope to perform a more thorough analysis of our qualitative data, which could also bring to light learner behaviors and explanations of our findings, to compliment are quantitative findings. This would be particularly interesting for effects related to how learners reframe and break down questions and prompts for the AI code generator. Currently, we still know very little of when novice learners used the AI code generator on each task, what their usage patterns were, and the details about how they interacted with the AI code generator. A qualitative analysis could also answer questions such as: how did learner express their intended behavior to the AI code generator?; what common themes existed in expressing their behavior?; and, what did they do when the code generator did not generate their desired code?

In terms of the target audience, our study focused on novice learners in the context of introductory programming. But there still exist many questions about the implications of AI Coding Assistants with other populations such as conversational programmers [15] who are motivated to improve their efficacy in technical conversations, or experienced programmers when they want to learn a new programming language or programming library. Similarly, a future study could be conducted where AI code generators are introduced in more formal learning environments such as at the high school or university level. Such a study could examine the malicious usages of AI code generators and its impact on introductory computer science education before widely integrating into curriculum. Furthermore, although our study included many non-native English speakers, but they were mostly comfortable explaining things in English. Future work could explore natural language programming with non-English speakers.

To measure learning performance, our study included evaluation post-tests that did not leave the boundaries of what learners were trained on. An interesting question would be to see how AI

code generators affect transfer of learning to new topics or more complicated tasks. Furthermore, we designed the tasks used in the training phase to help novices learn about the primary syntactical and logical constructs of a text-based programming. However, another unanswered question is whether AI code generators can support algorithmic thinking and algorithm design.

Finally, our study focused only on AI code generation which is one of the main capabilities of AI Coding Assistants. However, these tools could be utilized to generate explanations of code, or they could be used when novices are stuck on a task, by automatically fixing syntax and semantic issues in code (e.g., using the **code-davinci-edit-001** model trained by OpenAI Codex) with commands like "*fix the index error in my code*".

8 CONCLUSION

The prevalence of Large Language Model-based AI Coding Assistants like OpenAI Codex has the potential to change how educators teach and students learn about programming. To date, the learning effects and support provided by AI code generators have not been studied in the context of introductory programming. Our results indicate that AI Coding assistants can be an asset for computer science educators and students alike: AI code generators allowed novice programmers to perform better and faster with less frustration when writing code and did not reduce their performance on manual code modification or in the subsequent absence of AI code generators. While future studies are certainly warranted, we now have encouraging evidence that integrating AI Coding Assistants into programming support tools can scaffold the learning process for novices, which could help more students to engage in programming and broaden participation in computing.

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to the after-school coding and STEM camps that helped us run our study, especially **Coder Sports** in Ottawa, and **CodeZilla**, **Hive5 Innovative Center**, and **Junior Innovators** in the Greater Toronto Area. We thank you for your invaluable assistance in recruiting participants for our research study.

REFERENCES

[1] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM technical symposium on computer science education*. 522–527.
[2] Amazon Web Services. 2022. CodeWhisperer: ML-powered coding companion. https://aws.amazon.com/codewhisperer/. [Online; accessed 9-September-2022].
[3] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michał Moskal, and Jacqueline Russell. 2019. Microsoft MakeCode: embedded programming for education, in blocks and TypeScript. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. 7–12.
[4] Bruce W Ballard and Alan W Biermann. 1979. Programming in natural language: "NLC" as a prototype. In *Proceedings of the 1979 annual conference*. 228–237.
[5] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).
[6] David Bau, D Anthony Bau, Mathew Dawson, and C Sydney Pickens. 2015. Pencil code: block code for a text world. In *Proceedings of the 14th international conference on interaction design and children*. 445–448.
[7] Brett A Becker and Keith Quille. 2019. 50 years of cs1 at sigcse: A review of the evolution of introductory programming education research. In *Proceedings of the 50th acm technical symposium on computer science education*. 338–344.
[8] Andrew Begel. 1996. LogoBlocks: A graphical programming language for interacting with the world. *Electrical Engineering and Computer Science Department, MIT, Boston, MA* 2 (1996).
[9] Andrew Begel and Susan L Graham. 2005. Spoken programs. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 99–106.
[10] Klara Benda, Amy Bruckman, and Mark Guzdial. 2012. When life and learning do not fit: Challenges of workload and communication in introductory computer science online. *ACM Transactions on Computing Education (TOCE)* 12, 4 (2012), 1–38.
[11] Alan W Biermann, Bruce W Ballard, and Anne H Sigmon. 1983. An experimental study of natural language programming. *International journal of man-machine studies* 18, 1 (1983), 71–87.
[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[13] Miriam Bruhn and David McKenzie. 2009. In pursuit of balance: Randomization in practice in development field experiments. *American economic journal: applied economics* 1, 4 (2009), 200–232.
[14] 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).
[15] Parmit K Chilana, Rishabh Singh, and Philip J Guo. 2016. Understanding conversational programmers: A perspective from the software industry. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 1462–1472.
[16] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).
[17] Stephen Cooper, Wanda Dann, and Randy Pausch. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of computing sciences in colleges* 15, 5 (2000), 107–116.
[18] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*. 345–356.
[19] Edsger W Dijkstra. 1979. On the foolishness of" natural language programming". *Program Construction, International Summer School* (1979), 51–53.
[20] Pierre Dragicevic. 2015. *HCI Statistics without p-values*. Ph. D. Dissertation. Inria.
[21] Benedict Du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.
[22] Rodrigo Duran, Juha Sorva, and Sofia Leite. 2018. Towards an analysis of program complexity from a cognitive perspective. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 21–30.
[23] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. 2022. Cognitive Load Theory in Computing Education Research: A Review. *ACM Transactions on Computing Education (TOCE)* 22, 4 (2022), 1–27.
[24] Barbara J Ericson, James D Foley, and Jochen Rick. 2018. Evaluating the efficiency and effectiveness of adaptive parsons problems. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 60–68.
[25] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th koli calling international conference on computing education research*. 20–29.
[26] Garry Falloon. 2016. An analysis of young students' thinking when completing basic coding tasks using Scratch Jnr. On the iPad. *Journal of Computer Assisted Learning* 32, 6 (2016), 576–593.
[27] 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).
[28] Georgios Fessakis, Evangelia Gouli, and Elisavet Mavroudi. 2013. Problem solving by 5–6 years old kindergarten children in a computer programming environment: A case study. *Computers & Education* 63 (2013), 87–97.
[29] 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.
[30] Diana Franklin, Merijke Coenraad, Jennifer Palmer, Donna Eatinger, Anna Zipp, Marco Anaya, Max White, Hoang Pham, Ozan Gökdemir, and David Weintrop. 2020. An Analysis of Use-Modify-Create Pedagogical Approach's Success in Balancing Structure and Student Agency. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 14–24.
[31] Github. 2022. Copilot: Your AI pair programmer. https://github.com/features/copilot. [Online; accessed 9-September-2022].
[32] Shuchi Grover and Roy Pea. 2013. Computational thinking in K–12: A review of the state of the field. *Educational researcher* 42, 1 (2013), 38–43.
[33] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.
[34] Sumit Gulwani and Mark Marron. 2014. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings*

of the 2014 ACM SIGMOD international conference on Management of data. 803–814.

[35] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. *ACM Sigcse Bulletin* 35, 1 (2003), 153–156.

[36] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct manipulation interfaces. *Human–computer interaction* 1, 4 (1985), 311–338.

[37] Paul Jaccard. 1901. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bull Soc Vaudoise Sci Nat* 37 (1901), 547–579.

[38] Dhanya Jayagopal, Justin Lubin, and Sarah E Chasins. 2022. Exploring the Learnability of Program Synthesizers by Novice Programmers. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–15.

[39] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–19.

[40] Filiz Kalelioğlu. 2015. A new way of teaching programming skills to K-12 students: Code. org. *Computers in Human Behavior* 52 (2015), 200–210.

[41] Filiz Kalelioglu and Yasemin Gülbahar. 2014. The Effects of Teaching Programming via Scratch on Problem Solving Skills: A Discussion from Learners' Perspective. *Informatics in education* 13, 1 (2014), 33–50.

[42] Majeed Kazemitabaar, Viktar Chyhir, David Weintrop, and Tovi Grossman. 2022. CodeStruct: Design and Evaluation of an Intermediary Programming Environment for Novices to Transition from Scratch to Python. In *Interaction Design and Children*. 261–273.

[43] Majeed Kazemitabaar, Viktar Chyhir, David Weintrop, and Tovi Grossman. 2023. Scaffolding Progress: How Structured Editors Shape Novice Errors When Transitioning from Blocks to Text. In *Proceedings of the 54th acm technical symposium on computer science education*.

[44] Paivi Kinnunen and Beth Simon. 2010. Experiencing programming assignments in CS1: the emotional toll. In *Proceedings of the Sixth international workshop on Computing education research*. 77–86.

[45] Päivi Kinnunen and Beth Simon. 2011. CS majors' self-efficacy perceptions in CS1: results in light of social cognitive theory. In *Proceedings of the seventh international workshop on Computing education research*. 19–26.

[46] Paul Kirschner, John Sweller, and Richard E Clark. 2006. Why unguided learning does not work: An analysis of the failure of discovery learning, problem-based learning, experiential learning and inquiry-based learning. *Educational Psychologist* 41, 2 (2006), 75–86.

[47] Roman Knöll and Mira Mezini. 2006. Pegasus: first steps toward a naturalistic programming language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 542–559.

[48] Michael Kölling, Neil CC Brown, and Amjad Altadmri. 2015. Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education*. 29–38.

[49] Amruth N Kumar. 2013. A study of the influence of code-tracing problems on code-writing skills. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 183–188.

[50] Mathias Landhäußer, Sebastian Weigelt, and Walter F Tichy. 2017. NLCI: a natural language command interpreter. *Automated Software Engineering* 24 (2017), 839–861.

[51] Vu Le, Sumit Gulwani, and Zhendong Su. 2013. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. 193–206.

[52] Irene Lee, Fred Martin, Jill Denner, Bob Coulter, Walter Allan, Jeri Erickson, Joyce Malyn-Smith, and Linda Werner. 2011. Computational thinking for youth in practice. *Acm Inroads* 2, 1 (2011), 32–37.

[53] 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. *Science* 378, 6624 (2022), 1092–1097.

[54] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).

[55] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Acm sigcse bulletin* 41, 3 (2009), 161–165.

[56] Greg Little and Robert C Miller. 2006. Translating keyword commands into executable code. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*. 135–144.

[57] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*. 101–112.

[58] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[59] Andrew Luxton-Reilly, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 55–106.

[60] I Scott MacKenzie. 2012. Human-computer interaction: An empirical research perspective. (2012).

[61] Lauren E Margulieux, Mark Guzdial, and Richard Catrambone. 2012. Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proceedings of the ninth annual international conference on International computing education research*. 71–78.

[62] Raymond B Miller, Gwendolyn N Kelly, and Joseph T Kelly. 1988. Effects of Logo computer programming experience on problem solving and spatial relations ability. *Contemporary Educational Psychology* 13, 4 (1988), 348–357.

[63] Brad A Myers, John F Pane, and Amy J Ko. 2004. Natural programming languages and environments. *Commun. ACM* 47, 9 (2004), 47–52.

[64] John Francis Pane. 2002. *A programming system for children that is designed for usability*. Carnegie Mellon University.

[65] John F Pane and Brad A Myers. 2006. More natural programming languages and environments. *End user development* (2006), 31–50.

[66] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. 157–163.

[67] Shahira Popat and Louise Starkey. 2019. Learning to code or coding to learn? A systematic review. *Computers & Education* 128 (2019), 365–376.

[68] David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. 2000. NaturalJava: A natural language interface for programming in Java. In *Proceedings of the 5th international conference on Intelligent user interfaces*. 207–211.

[69] Sarantos Psycharis and Maria Kallia. 2017. The effects of computer programming on high school students' reasoning skills and mathematical self-efficacy and problem solving. *Instructional science* 45, 5 (2017), 583–602.

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

[71] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 878–888.

[72] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: synthesizing what I mean: code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*. 357–367.

[73] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional program synthesis from natural language and examples. In *IJCAI 2015*.

[74] Alexander Renkl. 2005. The worked-out-example principle in multimedia learning. *The Cambridge handbook of multimedia learning* (2005), 229–245.

[75] Mitchel Resnick. 2014. Give P'sa chance: Projects, peers, passion, play. In *Constructionism and creativity: Proceedings of the third international constructionism conference. Austrian computer society, Vienna*. 13–20.

[76] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[77] Mitchel Resnick and David Siegel. 2015. A different approach to coding. *International Journal of People-Oriented Programming* 4, 1 (2015), 1–4.

[78] Ma Mercedes T Rodrigo and Ryan SJd Baker. 2009. Coarse-grained detection of student frustration in an introductory programming course. In *Proceedings of the fifth international workshop on Computing education research workshop*. 75–80.

[79] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 27–43.

[80] Viktor Schlegel, Benedikt Lang, Siegfried Handschuh, and André Freitas. 2019. Vajra: step-by-step programming with natural language. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*. 30–39.

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

[82] Gail M Sullivan and Richard Feinn. 2012. Using effect size—or why the P value is not enough. *Journal of graduate medical education* 4, 3 (2012), 279–282.

[83] Jiao Sun, Q Vera Liao, Michael Muller, Mayank Agarwal, Stephanie Houde, Kartik Talamadupula, and Justin D Weisz. 2022. Investigating explainability of generative AI for code through scenario-based design. In *27th International Conference on Intelligent User Interfaces*. 212–228.

[84] John Sweller, Jeroen JG van Merriënboer, and Fred Paas. 2019. Cognitive architecture and instructional design: 20 years later. *Educational Psychology Review* 31 (2019), 261–292.

[85] Tabnine. 2022. Tabnine: AI assistant for software developers. https://www.tabnine.com/. [Online; accessed 9-September-2022].

[86] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[87] Jeroen JG Van Merriënboer, Paul A Kirschner, and Liesbeth Kester. 2003. Taking the load off a learner's mind: Instructional design for complex learning. *Educational psychologist* 38, 1 (2003), 5–13.

[88] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

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

[90] Mary Webb, Niki Davis, Tim Bell, Yaacov J Katz, Nicholas Reynolds, Dianne P Chambers, and Maciej M Sysło. 2017. Computer science in K-12 school curricula of the 2lst century: Why, what and when? *Education and Information Technologies* 22 (2017), 445–468.

[91] Jeannette M Wing. 2006. Computational thinking. *Commun. ACM* 49, 3 (2006), 33–35.

[92] David Wolber. 2011. App inventor and real-world motivation. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. 601–606.

[93] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).

[94] Rui Zhi, Min Chi, Tiffany Barnes, and Thomas W Price. 2019. Evaluating the effectiveness of parsons problems for block-based programming. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. 51–59.

[95] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).

A APPENDICES

A.1 Coding Steps Programming Tasks

This appendix outlines the tasks involved in the code authoring and modification tasks used in the training phase of our study (using Coding Steps). The tasks are organized into five tables based on their topics: basics, data types, conditionals, loops, and arrays.

Table A: Basics: output, variables, concatenating strings, input, arithmetic operations, and random numbers

Task #	Task Description
1A	Write a program that will display the following message: I'm Wall-E!
1B	Modify the given program so it displays another message after the first one: Beep Boop
2A	Write a program that first, creates a variable called name and sets its value to Wall-E. Then, display the value of the variable
2B	Modify the given program's variable name from name to robot_name
3A	Write a program that creates a variable called name and sets its value to Wall-E. Then, display the message My name is name
3B	Modify the given program so that it displays the following message: Hi, name! Nice to meet you!
4A	Write a program that creates a variable called name and sets its value to ro. Then, update the name variable by adding the value bot to its previous value. Finally, display the message Created: name
4B	Modify the given program so that instead of adding bot to the name variable at once, it adds the characters b, o, and t one at a time. Print the value of the variable name after adding each of the characters. Finally, display the message Created: name
5A	Write a program that asks the user for their name and then stores their name into a variable called name. Finally, display the message Hello, name!
5B	Modify the following program so that it also asks the user for their family name and stores it into family_name. Then, display the message Hello, name family_name!
6A	Write a program that first, creates a variable called food1 and set its value to nuts. Then, creates another variable called food2 sets it to bolts. Afterwards, create a third variable called robot_food and sets it to the value of food1 and food2. Finally, display the message I like robot_food. **Note:** pay attention to the space before and after the and
6B	Modify the following program so that it includes a third food (called food3) set to screws. Then modify robot_food to be the value of food1, food2 and food3. Finally display the message I like robot_food.
7A	Write a program that sets num1 to 20, and num2 to 5. Then set another variable called add to the addition of num1 and num2, sub to their subtraction, mult to their multiplication, and div to their division. Finally, display each of the add, sub, mult and div variables.
7B	Modify the following program so that it sets a new variable called some_num to the addition of all add, sub, mult and div. Then, in another line, update some_num by multiplying it by 2. Finally, display the value of some_num
8A	Write a program that generates a random number between 1 and 10 and sets it to a variable called num. Then, display the value of num
8B	Modify the following program so it generates a second random number between 50 and 100 and sets it to another variable named num2. Then, display the value of num2 below the value of num

Table B: Data-Types: cast integer to string, and string to integer

Task #	Task Description
9A	Write a program that first, sets the variable num to a random number between 1 and 10. Then create another variable called message and set it to the message num is: num. Then, display the value of message
9B	Modify the following program by creating a second variable called num2 and setting it to the number 5. Then change message to display the following message: num is: num and num2 is: num2
10A	Write a program that first, sets num1 to 12, and num2 to 21. Then sets a variable named message to the value num1 times num2 = (the value of num1 multiplied by num2). Finally, print message

Continued on next page

Table B – *Continued from previous page*

Task #	Task Description
10B	Modify the value of message so that it displays the value of num1 times num2 like the following example. **Note:** the values of num1 and num2 can be anything and your code should work regardless of their values
11A	Write a program that asks the user for two numbers and then displays the sum of them
11B	Modify the following program so that after displaying the sum of num1 and num2, it would ask for another number from the user and then display the sum of all three numbers
12A	Write a program that asks the user for four numbers and then displays the sum of them. Note that your program should only use one variable called total. The display message when asking the user to enter a new number should also include the value of total so far. At the end, it should display the value of total like this: Total: total
12B	Modify the following program by including a variable called count that would be incremented whenever a new number is entered. The display message when asking the user to enter a new number should also include the count of numbers entered so far. Finally, it should display the total and the count like this: The sum is: total from count entries.

Table C: Conditionals: if, elif, else, comparators, and logical expressions

Task #	Task Description
13A	Write a program that first, generates a random number between 1 and 6 and assigns it to a variable called roll and then display roll. Finally, display the message rolled six only if roll is equal to six
13B	Modify the following program so that it generates a second random number between 1 and 6 and sets it to another variable named roll2. Display both variables and finally, display the message rolled the same only if both rolls were equal
14A	Write a program that first, generates two random numbers between 1 and 6 and check if both of the variables are greater than 3 (either 4, 5, or 6). If both are greater than 3, then first display their values and then in another line, display the message: both rolled greater than 3
14B	Modify the following program so that it would generate a third random number called grade between 25 and 100. Then check if roll1 and roll2 are greater than 3 in addition to grade being greater than 50. If yes, display the values of each of the three variables and display the message All three above half
15A	Write a program that asks the user to enter a number between 10 and 100. Then, check if the number is greater than 75. If it is, display the message Greater than 75; otherwise, display the message Less than 75. Note that only one of these messages should be displayed
15B	Modify the following program so that it asks for a second number as well. Check if the first number is greater than the second. If it is, display the message First number is greater; otherwise, display the message Second number is greater. Note that only one of these messages should be displayed
16A	Write a program that asks the user to enter a number between 0 and 100 and set it to a variable called score. Additionally, create a variable called grade and set it to an empty text. Then check if the score is less than 50, if it is, then set grade to the letter C, if it's between 50 and 75, set grade to B, otherwise, set grade to A. Then display the message Grade: grade
16B	Modify the following program so that if the score is less than 20 set grade to F, if it's between 20 and 40 set grade to E, if it's between 40 and 60 set grade to D, if it's between 60 and 80 set grade to C, if it's between 80 and 90 set grade to B, otherwise, sets grade to A
17A	Write a program that creates a variable called coin. Then use a random number generator to generate a number between 1 and 2. If the number is 1, set coin to heads, otherwise, set it to tails. Then display the message Coin: followed by the value of coin
17B	Modify the program so that it would generate a random number between 1 and 7, and then display one of the days in the week based on the number generated
18A	Write a program that gets two numbers from the user and then asks for an operator (from one of the following choices: +, -, *, and /). Then it should check which operator the user has entered, and then perform the appropriate operation. For example, if the user enters + then it should add the two numbers and display the result

Continued on next page

Table C – *Continued from previous page*

Task #	Task Description
18B	Modify the program so that it would ask a third number. And then perform the appropriate operation between the three numbers for + and *. If the user enters a different operator, then display an error message: Error: Invalid operator
19A	Ask the user to enter a number and store it in a variable called num. Check if it is even or odd. If it is odd, display the message The number num is odd otherwise display the message The number num is even. **Hint:** a number is even if the remainder of the division of the number by 2 is 0 (or in other words, it's divisible by two)
19B	Modify the program so that it asks for another number called divisor then, checks if the entered number is divisible by the divisor. If it is, display the message The number num is divisible by divisor otherwise display the message The number num is not divisible by divisor
20A	Set two variables called num1 and num2 to a random number between 1 and 1000 and a third variable called result to 0. Ask the user to enter one of the two options: greater, or smaller and then check which one the user has entered. (Display an error message: Invalid Option if the user didn't enter any of the two). If the user enters greater, then check if the num1 is greater than num2. If it is, set result to num1 and otherwise, set result to num2. However, if the user enters smaller, then check if the num1 is smaller than num2. If it is, set result to num1 and otherwise, set result to num2. Finally, if the user did not enter an invalid input, display the message: You entered option, and the result is result
20B	Modify the code by adding a third option called equal that would check if the two numbers are equal or not. If they are, then display the message The numbers are equal otherwise, display three messages (each in a line): the value of num1, the value of num2, and the message The numbers are not equal

Table D: Loops: for loops, range, and while loops

Task #	Task Description
21A	Display Hello 10 times using a loop
21B	Modify the code so that it would instead repeat the following program for 5 times: display Hello then display World!. Then finally, display Bye Bye only once afterwards
22A	Set a variable called num to 0 and then create a loop that would add the number 5 to num for 25 times and display the value of the variable as it increases inside the loop
22B	Modify the program so that it includes another variable that is initially set to 125. Then the loop would reduce its value by 5 for 25 times. Display the value of both variables every time their value changes in the loop
23A	Set a variable called text to the text w. Then create a loop that would repeatedly add the letter e to the text for 5 times and displaying the text every time. After the loop, add an exclamation mark ! to the text variable and then display its value
23B	Add another loop to the program (after the first loop) that would add the text * for 3 times to the text variable and display the text every time. Finally, after the loop, add a dot . to the text variable and display its value
24A	Set a variable called fruits to the text I like these fruits: . Then create a loop that would repeatedly do the following things for 5 times: first, ask the user to enter a fruit name and then adding what the user entered to the fruits (separated with a space). After the loop, display the value of the fruits variable
24B	Change the program so that when it is done with the first loop, it would then create another variable called movies and set it to I like these movies: and then use another loop that would repeat for 5 times to ask the user for their favorite movies and then add them to the variable movies one by one. After the second loop, display the value of the movies variable
25A	Display all the numbers from 1 to 100 line by line using a loop
25B	Modify the following program so that it would display all the numbers from 250 to 300 line by line and after each line (inside the loop), it would also display the number multiplied by 2
26A	Ask the user to enter a number, then display all the numbers from 1 to the number entered line by line
26B	Modify the following program so that it would ask two numbers, then display all the numbers from the first number to the second number line by line

Continued on next page

Table D – *Continued from previous page*

Task #	Task Description
27A	Write a program that would ask the user to enter a number, then use a loop to calculate the total sum of all numbers from 1 to the given number (including the given number). Finally, display the total
27B	Modify the program so that it would ask the user for two numbers and then use a loop to calculate the total sum of all numbers between the first and the second one (including both first and second numbers). (Note that the second number should always be greater than the first number). So for example, if the user enters 3 and 8, it should calculate the sum of 3, 4, 5, 6, 7, and 8. Within the loop, also display the value of the total as it increases every time. Then finally, display the total
28A	Write a program that would calculate the sum of all even numbers between 1 to a number asked from the user (including that number). Finally, display the sum. **Hint:** a number is even when the remainder of its division by 2 is 0
28B	Modify the loop so that it would also calculate the sum of all odd numbers between 1 to the given number at the same time. Then display two messages, first the sum of all even numbers, then the sum of all odd numbers. **Hint:** a number is even when the remainder of its division by 2 is 0, and odd when the remainder of its division by 2 is 1
29A	Write a program that uses a while loop to repeatedly ask the user to enter a password (as a number) and check if the password is equal to 123. If it is, display the message Password is correct. If it is not, display the message Password is incorrect and ask the user to re-enter the password. The program should stop when the user enters the number 123. Finally, after the user gets the correct password, display the message Password is correct
29B	Modify the program by changing the password from 123 to 7512 and also count the number of incorrect attempts (each time the user enters the password incorrectly). Finally display the incorrect attempts at the end of the loop
30A	Write a program that repeatedly does the following until the user enters the number 0: ask the user for a number, and then add it up to a variable called total. If the user enters 0, display the total at the end (only once)
30B	Modify the program so that it calculates the average of all numbers entered by the user. **Note:** the average is the sum of all numbers entered, divided by the count of numbers entered. Hint: use another variable to count the number of numbers entered by the user
31A	Write a program that asks the user to enter a number between 1 and 100 and then display the difference of that number with the value 50. The difference between two numbers is always a positive number. **Note:** you can use the abs( ) function to calculate the positive value of any number
31B	Modify the program so that asks the user for two numbers and then display the difference between each of the numbers. Again, note that the difference between two numbers is always a positive number
32A	Write a guess a number game: the program will first set the variable picked_number to a random number between 1 and 1000. Then it should repeatedly do the following until the user guesses the number (if it's equal to picked_number): If the user guesses a number that is too high, the program should display the message The number is too high. If the user guesses a number that is too low, the program should display the message The number is too low. Finally, if the user guesses the number, the while loop should stop repeating and the program should display the message You guessed the number!
32B	Modify the program so that it would count the number of incorrect attempts the user has made and display it at the end. Additionally, on every guess it should check if the difference between the guessed number and the picked number is less than 50. If it is, the program should display another message You are close!. **Note:** you can use the abs( ) function to calculate the positive value of any number
33A	Write a program that would use a while loop to repeatedly ask the user to guess a number until the user enters the number 0. Inside the loop, check if the number is divisible by both 2 and 3, if it is, then display the message The number is divisible by 2 and 3 and then break out of the loop. At the end of the loop, it should simply display the message Finished loop
33B	Modify the program by adding another if statement inside the while loop to check if the number is divisible by 5, if it is then display the message The number is divisible by 5 and then break out of the loop. The program should also include another variable called did_break that is set to False and then set to True if one of the breaks are triggered. At the end, display the message broke out of the loop only if the variable did_break is equal to True
34A	Write a program that asks the user to enter a number between 1 and 100. The program should then repeatedly decrease the number by 1 until it reaches 0 and display the number each time

Continued on next page

Table D – *Continued from previous page*

Task #	Task Description
34B	Modify the program so that it would first ask the user to enter a number greater than 100 and then use the loop to continuously decreases the number by 10 every time while the number is greater than 100. At this point the number should be equal or less than 100, use another loop to continuously decrease the number by 3 every time while the number is greater than zero
35A	Write a program that generates a number between 1 and 999999. Then, it displays the number of digits in the number by repeatedly dividing the number by 10 until it reaches 0 counting the number of times it was divided. For example, 1874 10 = 187 (first) - 187 10 = 18 (second) - 18 10 = 1 (third) - 1 10 = 0 (fourth). Therefore, 1874 has four digits
35B	Modify the program so that it displays each digit of the number from the right to the left as the number is being divided by 10. To obtain the digit, the program should use the modulus operator (%) to obtain the remainder of the division
36A	Write a program that includes a for loop that uses the variable i to go from 0 to 1 (including 1). Then inside the loop, have another loop that uses the variable j to go from 0 to 1 (including 1). The program should display the value of i and j every time like the provided sample
36B	Modify the program so that the first loop would go from 0 to 10 (instead of 0 to 1) and the second loop to go from 0 to 2 instead (including 2). It should also display the message i changed to i whenever the value of i (in the outer loop) changes
37A	Write a program that repeatedly generates a random number between 0 and 100 until the random number that it generates becomes equal to 50 (and then stop). Then display the number of attempts it took to generate the number
37B	Modify the program so that it stops when the random number becomes equal to any of the numbers 25, 50, or 75. It should also display which number it stopped on after displaying the number of attempts after the loop
38A	Repeatedly roll a dice for 1000 times. At the end, display the total times it rolled six
38B	Modify the program so that it counts the number of times it rolled each of the six faces (using six variables) and then finally display the value of all six variables

Table E: Arrays: initializing lists, obtaining a list's length, appending to lists, iterating over lists

Task #	Task Description
39A	Create a list with these values: 1, 5, 9, 13, 17, 21. Then, display the first item in the list by accessing the list using the appropriate indices. Then, display the length of the list. Hint: you should use a special function that returns the length of a list
39B	Modify the program so that it displays the last item in the list by accessing the list using the appropriate index. You have to calculate the index of the last item of the list using the length of the list (ask yourself what the relationship between the index of the last item of a list and the length of a list is). **Note:** You must use the len function to determine the length of the list
40A	Write a program that creates a list with the following textual values: "math", "history", "programming", and "art". Then use a while loop and an index variable to display all of the items in the list one by one
40B	Modify the program so that it displays the items in the list in reverse order
41A	Write a program that creates an empty list and then, inside a for loop that repeats for 10 times, ask the user to enter a number and then add it to the end of the list. At the end, display the length of the list
41B	Rename the list to grades and create another empty list called students before the loop. Then, inside the loop, first ask the user to enter a student name (as a text/string) and then add the student name into the students list. Then, ask the user to enter a grade (as a number/integer) and then add the grade into the grades list. Finally, after the loop, display the length of both lists
42A	Create an empty list called grades. Then, repeatedly add a random number between 50 to 100 to the list, for a random number of times (between 15 and 25). Finally, use another loop to display all the items in the grades list. **Note:** your program should use the for loop with the range function, not a while loop
42B	Modify the code so that it defines a second list called grades2 and uses another loop to repeatedly add a random number between 1 to 10 to the grades2 list for a random number of times (between 100 to 500). In summary, your code should define two lists with random values and random lengths, then display the contents of both lists

Continued on next page

Table E – *Continued from previous page*

Task #	Task Description
43A	Create a list called numbers, and then use a for loop that repeats for 5 times to repeatedly ask the user to enter a number (as an integer) and add it to the list. Then use another loop to go through the items of the numbers list and find the largest number. Finally, display the value of the largest number. (**Note:** you can NOT use the max function.)
43B	Modify the program so that it also finds the smallest number in the list and displays it at the end. (**Note:** you can NOT use the min or max function.)
44A	Repeatedly ask the user to enter a movie name and add it to a list called movies until the user enters stop. At the end just display how many movies the user has entered. **Note:** The list should not contain the word stop
44B	Create another list called ratings and for each movie that is entered (that is not equal to stop), ask the user to enter a rating from 0 to 10 (as an integer) and add the number to the ratings list. At the end, display the number of movies and the number of ratings **Note:** they should have the same number of elements and stop should not be included in the movies
45A	Create an empty list called numbers and then use a for loop that repeats for a random number of times between 50 to 75 to update the list by adding a random number between 0 to 100. Then use another loop to find the largest number in the list. Finally, display the largest number after the second loop has finished
45B	Create another variable called smallest and use the second for loop to find the smallest number in the list in addition to the largest. At the end, display both the largest and the smallest numbers