Learning to Code
with
Natural Language Programming
powered by LLMs

Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming (CHI'23)
Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, Tovi Grossman

How Novices Use LLM-based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment (Koli Calling'23)
Majeed Kazemitabaar, Xinying Hou, Austin Z. Henley, Barbara J. Ericson, David Weintrop, Tovi Grossman
Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming

Majed Kazemtabaah, University of Technology, Toronto, Ontario, Canada
Barbaj J. Ericson, School of Information, University of Michigan
Ann Arbor, Michigan, USA
barbara@umich.edu

Justin Chow, Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
justinchow@utoronto.ca

David Weintrop, School of Computer Science, University of Toronto
Toronto, Ontario, Canada
davidw@cs.toronto.edu

Tovi Grossman, Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
tovig@cs.toronto.edu

1. Prompt Description
2. Generated Code
3. Manual Modification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Permissions may be requested from PublicationsPermission@acm.org.

© Copyright 2023 by the authors. Publication rights licensed to ACM.

https://doi.org/10.1145/3566781.3573850

ABSTRACT

AI code generation like OpenAI Code has 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 68 novices (ages 19–27). Learners worked on 45 Python rule-authoring tasks, for which half of the learners had access to Codex, followed by a code-modification task. Our results show that using Codex increased code-authoring performance (11.6% increased completion rate and 1.8% fewer errors) 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-test, conducted one week later, although this difference did not reach statistical significance. Of interest, learners with higher reach performances performed significantly better on retention post-tests, if they had prior access to Codex.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Permissions may be requested from PublicationsPermission@acm.org.

© Copyright 2023 by the authors. Publication rights licensed to ACM.

https://doi.org/10.1145/3566781.3573850

How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment

Majed Kazemtabaah, University of Technology, Toronto, Ontario, Canada
majeedsl@alumni.utoronto.ca

Xinying Hou, School of Information, University of Michigan
Ann Arbor, Michigan, USA
xinying@umich.edu

Barbaj J. Ericson, School of Information, University of Michigan
Ann Arbor, Michigan, USA
barbara@umich.edu

David Weintrop, School of Computer Science, University of Toronto
Toronto, Ontario, Canada
davidw@cs.toronto.edu

Tovi Grossman, Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
tovig@cs.toronto.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Permissions may be requested from PublicationsPermission@acm.org.

© Copyright 2023 by the authors. Publication rights licensed to ACM.

https://doi.org/10.1145/3566781.3573850

ABSTRACT

As Language Models (LMs) gain in popularity, it is important to understand how novice programmers use them and the effect they have on learning to code. We present the results of a thematic analysis on data from 33 learners, aged 16–19, who independently learned Python by working on 46 rule-authoring tasks with access to an AI Code Generator based on OpenAI Codex. We explore several important questions related to how learners used LLM-based AI code generators, and provide an analysis of the properties of the written prompts and the resulting AI-generated code. Specifically, we explore (A) the context in which learners test Codex, (B) what learners are asking from Codex in terms of syntax and logic, (C) properties of prompt written by learners in terms of relation to task description, language, logic, and prompt crafting strategy, (D) properties of the AI-generated code in terms of correctness, complexity, and (E) how learners utilize AI-generated code in terms of placement, verification, and manual modifications. Furthermore, our analysis reveals four distinct coding approaches when writing code with an AI code generator: AI Single Prompt, when learners prompted Codex once to generate the entire solution to a task, AI Split-by-Stage, when learners divided the problem into parts and used Codex to generate such part, Hybrid, when learners wrote some of the code themselves and used Codex to generate others; and Manual coding, when learners wrote the code themselves. The AI Single-Prompt approach resulted in the highest correctness scores on self-coding checkout tasks, but the lowest correctness scores on subsequent code-modification tasks during training. The Hybrid approach also resulted in similar correctness outcomes. In contrast, AI Single-Prompt and Manual coding approaches significantly outperformed Access to Code generation and the challenges and opportunities associated with integrating them into self-paced learning environments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Permissions may be requested from PublicationsPermission@acm.org.

© Copyright 2023 by the authors. Publication rights licensed to ACM.

https://doi.org/10.1145/3566781.3573850

1 INTRODUCTION

Novices in this study are upper-level college students (aged 16–19) who were exposed to AI code generation tools during their introductory programming course. They used an LLM-based AI code generator called OpenAI Codex during an introductory programming course. The AI code generators are available as plugins in modern IDEs and can be customized to suit different coding environments. The main objective of our study was to understand how novices used AI code generators to solve coding tasks in a self-paced, self-directed learning environment. This approach allowed us to observe learners’ coding processes and behaviors with minimal intervention. We also examined the challenges and opportunities associated with integrating AI code generators into self-paced learning environments.

Despite their potential benefits, LM-based code generation tools can hinder learning by providing incorrect or incomplete code, which can lead to incorrect conclusions. However, this study aimed to address these challenges by providing insights into the use of AI code generators in self-paced learning environments.
Abstract

AI code generation 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 18-19). Learners worked on 4 Python coding assignments, for which half of the learners had access to Codex, followed by a code-modification task. Our results show that using Codex significantly increased code-rewriting performance (F(1,45) = 14.9; p < .001) and the number of lines written (F(1,45) = 14.9; p < .001) whilst not increasing performance on manual code-modification tasks. Additionally, learners with access to Codex during the training phase performed slightly better on the evaluation post-test, conducted one week later, although this difference did not reach statistical significance. Of interest, learners with higher interest in programming performed significantly better on retention post-tests, if they had prior access to Codex.

CSC Concepts

- Human-centered computing — Interactive systems and tools
- Social and professional topics — Computing education

Keywords

Large Language Models, AI Coding Assistants, ASIST, PAT-Programming, OpenAI Codex, GPT, ChatGPT, Codex, Introductory Programming, K-12 Computer Science Education

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2023 Copyright held by the owner/authors. Publication rights licensed to ACM.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2023 Copyright held by the owner/authors. Publication rights licensed to ACM.

1 Introduction

Powered by the recent advancements in Deep Learning [16], Large Language Models that are trained on millions of lines of code, such as OpenAI Codex [14], can automatically generate code from natural language descriptions (Figure 1.1). 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 [14, 17]. By generating code from simple sentences instead of formal and syntactically-based specifications, these AI-Coding Assistants may lower the barriers to entry into programming.

Figure 1.1: Generate-modify-assign with AI code generators. Right: Summary of our controlled study over 30 sessions.

How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment

Majed Kazemitaarab
Department of Computer Science, University of Toronto, Toronto, Canada
majeed@uwaterloo.ca

Xiyu Hou
School of Information, University of Maryland
ann.arbor@umich.edu

Barbara J. Ericson
School of Information, University of Michigan
barbara@umich.edu

David Winetrop
College of Education, University of Maryland
College Park, Maryland, USA
westrop@umd.edu

Tovi Grossman
Department of Computer Science, University of Toronto, Toronto, Canada
cmg@cs.toronto.edu

Koli Calling Conference in Computing Education Research

Chi Conference in Human Factors in Computing

Koli, Finland, May 2023
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.

```python
import random

attempts = 0
while True:
    num = random.randint(0, 100)
    attempts += 1
    if num == 50:
        print("It took", attempts, "attempts to generate the number 50.")
        break
```
Intro:  LLMs Trained on Code

```javascript
#!/usr/bin/env ts-node

import { fetch } from "fetch-h2";
```

Github Copilot
Released: June 2021
Intro: Generative Operations

**Language to code**

- **Natural Language** → **Code**
  - Code Generation

- **Code** → **Natural Language**
  - Explanation
  - Evaluation

**Code to code**

- **Code** → **Code**
  - Completion
  - Summarization
  - Repair
  - Translation
Enables Natural Language Programming
Enables Natural Language Programming

> ask the user to enter a number

```python
num = int(input("enter a number: "))
```
Intro: Natural Language Programming

Potential Benefits

Focus on problem-solving aspects of computing
Intro: Natural Language Programming

Potential Benefits

Focus on problem-solving aspects of computing

Help with debugging and fixing syntax errors
Natural Language Programming

Potential Benefits

Focus on **problem-solving** aspects of computing

Help with **debugging** and **fixing** syntax errors

Generating a **variety** of correct solutions


**Intro:** Natural Language Programming

**Potential Benefits**

- Focus on **problem-solving** aspects of computing
- Help with **debugging** and **fixing** syntax errors
- Generating a **variety** of correct solutions

**Potential Drawbacks**
Potential Benefits

- Focus on **problem-solving** aspects of computing
- Help with **debugging** and **fixing** syntax errors
- Generating a **variety** of correct solutions

Potential Drawbacks

*Usage Challenges:*

- Properly express their intentions
Potential Benefits:

- Focus on **problem-solving** aspects of computing
- Help with **debugging** and **fixing** syntax errors
- Generating a **variety** of correct solutions

Potential Drawbacks:

Usage Challenges:

- Properly express their intentions
- Understand, verify and use AI-generated code
Potential Benefits

- Focus on **problem-solving** aspects of computing
- Help with **debugging** and **fixing** syntax errors
- Generating a **variety** of correct solutions

Potential Drawbacks

**Usage Challenges:**
- Properly express their intentions
- Understand, verify and use AI-generated code

**Behavioral Challenges:**
- Learners might become overly-dependent
Intro: Natural Language Programming

Potential Benefits

- Focus on problem-solving aspects of computing
- Help with debugging and fixing syntax errors
- Generating a variety of correct solutions

Potential Drawbacks

Usage Challenges:

- Properly express their intentions
- Understand, verify and use AI-generated code

Behavioral Challenges:

- Learners might become overly-dependent

Ethical Issues:

- Academic integrity, plagiarism, and attribution
Intro: Impact on Learning

New York City Public School:

“ChatGPT doesn’t help build critical-thinking and problem-solving skills”

January 2023

Seattle public school district:

“The district does not allow cheating and requires original thought and work from students”

January 2023

source: www.artimatic.io
RESEARCH GOAL

Scale Programming Education?
Explore the **Impact** of using AI Code Generators on Young Students When **Learning to Write Code for the First Time.**
RESEARCH QUESTIONS
RESEARCH QUESTIONS

RQ1: Code Composition:
How do learners’ task performance differ with and without AI code generators?
**RESEARCH QUESTIONS**

**RQ1**
**Code Composition:**
How do learners’ task performance differ with and without AI code generators?

**RQ2**
**Manual Code Modification:**
How does prior access to the AI code generator affect learners’ ability to manually modify code?
RESEARCH QUESTIONS

RQ1
Code Composition:
How do learners’ task performance differ with and without AI code generators?

RQ2
Manual Code Modification:
How does prior access to the AI code generator affect learners’ ability to manually modify code?

RQ3
Learning Retention:
What are the effects on learning performance and retention from using an AI code generator versus not using?
CONTROLLED STUDY

Codex Group
- 33 Participants
- Had access to AI Code Generator

Baseline Group
- 36 Participants

10 Sessions
Self-Paced Python Training
7 Sessions

Evaluation
2 Sessions

Session 1

STUDY PROCEDURE

Scratch Intro + Pre-Test
1 Session

Self-Paced Python Training
7 Sessions

Evaluation
2 Sessions

Immediate Post-Test
one week later
Retention Post-Test

Code Authoring

Code Modifying

Scratch Lecture

Scratch Pre-Test

x45
STUDY PROCEDURE

**Scratch Intro + Pre-Test**
1 Session
- Scratch Lecture → Scratch Pre-Test

**Self-Paced Python Training**
7 Sessions
- Code Authoring → Code Modifying ×45

**Evaluation**
2 Sessions
- Immediate Post-Test → one week later → Retention Post-Test

Session 9 to 10

1 2 3 4 5 6 7 8 9 10
AI ASSISTED PROGRAMMING

Coding Steps

Optional AI Code Generator
Self-Paced Python Learning
Logs all Activities

Available Open Source:
https://github.com/MajeedKazemi/coding-steps
Task Description:
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

Sample:
output: both rolled greater than 3
User Study

PARTICIPANTS

Total Participants: 69 (21 female, 48 male)

Ages: 10 – 17 ($M=12.53$, $SD=1.83$)

Recruitment: from multiple coding camps

Prior Programming Experience: 64 indicated using Scratch
Intro + Scratch Pre-Test

1 Session
1. Scratch Lecture (75 mins)
   Topics: variables, operators, conditionals, loops, and arrays

2. Scratch Pre-Test (45 mins)
   25 Multiple-Choice Questions
   Same Topics
STUDY PROCEDURE

Scratch Pre-Test Samples

What does this program say?

```
set var1 to 3
if var1 > 4 then
  set var1 to 5
else
  set var1 to 6
say var1
```

What does this program say?

```
set var1 to 10
repeat 5
  set var1 to var1 + 1
say var1
```

What does this program say?

```
set x to 1
repeat until x > 10
set x to x + 1
say x
```

Intro + Pre-Test

Scratch Lecture → Scratch Pre-Test

Self-Paced Python Training

Code Authoring → Code Modifying x45

Evaluation

Immediate Post-Test → one week later → Retention Post-Test
Self-Paced Python Training

7 Sessions
**STUDY PROCEDURE**

**Two Normalized Groups**

**Codex Group**
- **Count:** 33 Participants
- **Gender:** 11 Female

**Scratch Pre-Test:** 63%

**Baseline Group**
- **Count:** 36 Participants
- **Gender:** 10 Female

**Scratch Pre-Test:** 60%

**Self-Paced Python Training**

- Code Authoring
- Code Modifying

- *x45*

**Evaluation**

- Immediate Post-Test
- One week later
- Retention Post-Test
**STUDY PROCEDURE**

**Python Topics**

- **basics**
  - 8 coding + 6 MCQ

- **data-types**
  - 4 coding + 4 MCQ

- **conditionals**
  - 8 coding + 10 MCQ

- **loops**
  - 18 coding + 9 MCQ

- **arrays**
  - 7 coding + 10 MCQ

---

**Intro + Pre-Test**

**Self-Paced Python Training**

- Code Authoring
- Code Modifying

**Evaluation**

- Immediate Post-Test
- one week later
- Retention Post-Test
1. Authoring Task

**Task Description:**
Repeatedly generate a random number from 0 to 100 until it generates 50. Then display the number of times it took to generate the number.

**Sample Output:**
It took 27 attempts.

2. Modifying Task

**Task Description:**
Modify the program so it stops on any of the numbers 25, 50, or 75.

```python
from random import randint
num = randint(0, 100)
count = 0
while num != 50:
    num = randint(0, 100)
    count += 1
print(str(count) + " attempts.")
```

**Codex Group:**
Access to AI Code Generation

**Without AI Code Generation**
(Regardless of Condition)
Evaluation Post-Tests

2 Sessions
1. Immediate Post-Test
- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

2. Retention Post-Test
- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

Evaluation Post-Test

STUDY PROCEDURE

Intro + Pre-Test
- Scratch Lecture
- Scratch Pre-Test

Self-Paced Python Training
- Code Authoring
- Code Modifying
- x45

Evaluation
- Immediate Post-Test
- one week later
- Retention Post-Test

No Python Documentation • No Instructor Hints • No AI Code Generators
Results
Differences in task performance measures

Overall Completion rate (progress)
Task Completion time
Task Correctness score
RESULTS

Training Phase

Authoring + Modifying Tasks

Overall Task Completion Rate (%)

- Codex: 91%
- Baseline: 79%

Significant Difference: $p < .006$
RESULTS

Training Phase

Authoring Tasks

<table>
<thead>
<tr>
<th>Task Correctness Score (%)</th>
<th>Codex</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>80%</td>
<td>43%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Completion Time (s)</th>
<th>Codex</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>210s</td>
<td>357</td>
</tr>
</tbody>
</table>
AI Code Generator Usage

- Students used the code generator n=1646 times (1.21 times per task)
- 32% (n=530) of prompts were an exact copy of the task description
- Final code of 49% tasks was 100% AI generated (unmodified)
AI Code Generator Usage

- Students used the code generator n=1646 times (1.21 times per task)
- 32% (n=530) of prompts were an exact copy of the task description
- Final code of 49% tasks was 100% AI generated (unmodified)
**RESULTS**

**Training Phase**

**AI Code Generator Usage**

- Students used the code generator $n=1646$ times (1.21 times per task)
- 32% ($n=530$) of prompts were an exact copy of the task description
- Final code of 49% tasks was 100% AI generated (unmodified)
Differences in **manual code modification**

*Without* the AI Code Generator
Modifying Tasks
without AI code generators

RESULTS

Training Phase

Task Correctness Score (%)

- Codex: 66%
- Baseline: 57%

Task Completion Time (s)

- Codex: 225s
- Baseline: 244s
Differences in **Learning Performance and Retention**
1. Immediate Post-Test

- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

2. Retention Post-Test

- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

No Python Documentation * No Instructor Hints * No AI Code Generators

RESULTS

Evaluation

Intro + Pre-Test  Self-Paced Python Training  Evaluation

Scratch Lecture → Scratch Pre-Test  Code Authoring → Code Modifying  Immediate Post-Test → one week later → Retention Post-Test

RESULTS
Immediate Post-Test

**Correctness Score (%)**

- **Authoring**
  - Codex: 61%
  - Baseline: 63%

- **Modifying**
  - Codex: 60%
  - Baseline: 59%

- **Multiple-Choice**
  - Codex: 49%
  - Baseline: 42%
1. Immediate Post-Test
- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

2. Retention Post-Test
- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

No Python Documentation • No Instructor Hints • No AI Code Generators

RESULTS
RESULTS Retention Post-Test

**Authoring**

- **Correctness Score (%):**
  - Codex: 59%
  - Baseline: 50%

**Modifying**

- **Correctness Score (%):**
  - Codex: 47%
  - Baseline: 35%

**Multiple-Choice**

- **Correctness Score (%):**
  - Codex: 44%
  - Baseline: 35%
Differences in Perceptions about Learning and Frustration
Student Perceptions

RESULTS

Eager to continue learning about programming

Felt stressed, discouraged, and irritated

Felt that I learned a lot about Python Programming
Student Perceptions

RESULTS

Eager to continue learning about programming
- Codex: Completely
- Baseline:

Felt stressed, discouraged, and irritated
- Codex: Not at all
- Baseline: Completely

Felt that I learned a lot about Python Programming
- Codex: Completely
- Baseline: Completely
Student Perceptions

Eager to continue learning about programming
- Codex: Completely
- Baseline: Not at all

Felt stressed, discouraged, and irritated
- Codex: Completely
- Baseline: Not at all

Felt that I learned a lot about Python Programming
- Codex: Completely
- Baseline: Not at all
Overall, having access to AI Code Generators:

• Significantly increased completion rate of tasks
• Significantly increased code-authoring performance (correctness)
• Did not decrease manual code modification performance
• Felt more motivated, and less stressed during the training phase
• Slightly increased performance on retention tests
But how…?

Let’s dig deeper…

How prior programming skills affects learning performance with and without Codex?
Divided learners into four groups based on Scratch pre-test scores and access to Codex.

**RESULTS**

- **Scratch Pre-Test Score (%):**
  - **Codex High:** 86%
  - **Baseline High:** 82%
  - **Codex Low:** 42%
  - **Baseline Low:** 37%

**Top 50%**

**Lower 50%**
RESULTS: Effect of Prior Programming

Evaluation Phase: Retention Test

Authoring
- Top 50%
- Lower 50%

Modifying
- Top 50%
- Lower 50%

Multiple-Choice
- Top 50%
- Lower 50%
RESULTS

Effect of Prior Programming

Evaluation Phase: Retention Test

Authoring

Modifying

Multiple-Choice

Top 50%

Lower 50%

Top 50%

Lower 50%

Top 50%

Lower 50%
Part Two:

To understand the benefits and drawbacks of LLM-powered Coding tools, it’s crucial to know how students use them.

We analyzed usage patterns of students using Codex.
RESEARCH QUESTIONS

RQ1 How Novices Use AI Code Generators?
RESEARCH QUESTIONS

RQ1 How Novices Use AI Code Generators?

RQ2 Effect of Coding Approaches on Learning?
Collected Data:

- 1379 submitted tasks (356 manually, without Codex)
- 1666 Codex usages (1.62 usage per task)
- Code edit logs + Console run logs + Codex usages
Sequence of Actions:

Codex
Manual Edit
Run Code
Manual Edit
Codex
Run Code
Submit
Codex Usage

- Prompt Message
- Similarity with Task Description
- Generated Code

```
# Instructions: check if a variable is an even number
if number % 2 == 0:
    print("The number is even")
```
## Manual Code Edit

- **Code Before Edit vs. After Edit**
- **Diff:** Before vs. After
- **Key-Strokes Count**

### Code Before Edit
```
number = int(input("enter a number : "))
sum = 0
if number % 2 == 0:
    print("The number is even")
else:
    print("The number is odd")
```

### Code After Edit
```
number = int(input("enter a number : "))
sum = 0
if number % 2 == 0:
    print("The number is even")
else:
    print("The number is odd")
```

### Key-Strokes Count
- **34 key-strokes**

<table>
<thead>
<tr>
<th>Expand 1 lines ...</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2 sum = 0</td>
<td>2 sum = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 if number % 2 == 0:</td>
<td>3 if number % 2 == 0:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 print(&quot;The number is even&quot;)</td>
<td>4 print(&quot;The number is even&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 +</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 + else:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 + print(&quot;The number is odd&quot;)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Code Execution

- Code that was Executed
- Console Input and Output

```python
number = int(input("enter a number: "))
num = 0
if number % 2 == 0:
    print("The number is even")
```

**console output:**

```plaintext
-->> enter a number:
<-- 7
-->> enter a number:
<-- 8
-->> The number is even
```
Analysis Interface

Code Submission

- Code that was Submitted
- Any feedback provided by TAs

```python
start = int(input("Enter a start number: "))
end = int(input("Enter an end number: "))
sum = 0
for i in range(start, end + 1):
    if i % 2 == 0:
        sum += i
print(sum)
```
Sequence of Actions:

Codex
Manual Edit
Run Code
Manual Edit
Codex
Run Code
Submit
When did Learners Use Codex?

Focus of Thematic Analysis:
- Prior manual edits
- Prior codex usage
- Existing state of code
RQ1 A  When did Learners Use Codex?

Five Primary Situations:

- **At the Beginning**: 46%
- **After Clearing the Editor**: 5%
- **After Manual Coding**: 17%
- **After Using Codex**: 34%
- **While Having the Solution**: 1%
RQ1: When did Learners Use Codex?

Five Primary Situations:

- **At the Beginning**: 46%
- **After Clearing the Editor**: 5%
- **After Manual Coding**: 17%
- **After Using Codex**: 34%
- **While Having the Solution**: 1%
When did Learners Use Codex?

Five Primary Situations:

- **At the Beginning**: 46%
- **After Clearing the Editor**: 5%
- **After Manual Coding**: 17%
- **After Using Codex**: 34%
- **While Having the Solution**: 1%
RQ1 A When did Learners Use Codex?

Five Primary Situations:

- At the Beginning: 46%
- After Clearing the Editor: 5%
- After Manual Coding: 17%
- After Using Codex: 34%
- While Having the Solution: 1%
Five Primary Situations:

- At the Beginning: 46%
- After Clearing the Editor: 5%
- After Manual Coding: 17%
- After Using Codex: 34%
- While Having the Solution: 1%
RQ1A. When did Learners Use Codex?

Five Primary Situations:

- **At the Beginning:** 46%
- **After Clearing the Editor:** 5%
- **After Manual Coding:** 17%
- **After Using Codex:** 34%
- **While Having the Solution:** 1%
When did Learners Use Codex?

Situation: Starting with Codex ($n=760$, 46%)

Common Behaviors *When Using Codex at The Beginning:*

- Copy Full Task Description Generate Entire Solution: 66%
- Rephrase Task Description Generate Entire Solution: 7%
- Break Down Task into Subgoals and Generate First Subgoal: 26%
RQ1 A  When did Learners Use Codex?

Situation: Starting with Codex (n=760, 46%)

Common Behaviors When Using Codex at The Beginning:

- Copy Full Task Description Generate Entire Solution: 66%
- Rephrase Task Description Generate Entire Solution: 7%
- Break Down Task into Subgoals and Generate First Subgoal: 26%
RQ1A When did Learners Use Codex?

Situation: Starting with Codex (n=760, 46%)

Common Behaviors When Using Codex at The Beginning:

- Copy Full Task Description Generate Entire Solution: 66%
- Rephrase Task Description Generate Entire Solution: 7%
- Break Down Task into Subgoals and Generate First Subgoal: 26%
RQ1 A

When did Learners Use Codex?

Situation: Starting with Codex (n=760, 46%)

Common Behaviors When Using Codex at The Beginning:

- Copy Full Task Description Generate Entire Solution: 66%
- Rephrase Task Description Generate Entire Solution: 7%
- Break Down Task into Subgoals and Generate First Subgoal: 26%
Situation: After Manual Coding (n=282, 17%)

State of Code When Using Codex After Manual Coding:

- After Mostly Incorrect Code: 27%
- After Mostly Correct Code with Minor Issues: 41%
- After Writing Mostly Correct Code: 36%
RQ1 A  When did Learners Use Codex?

Situation: After Manual Coding (n=282, 17%)

State of Code When Using Codex After Manual Coding:

- After Mostly Incorrect Code: 27%
- After Mostly Correct Code with Minor Issues: 41%
- After Writing Mostly Correct Code: 36%
RQ1 A When did Learners Use Codex?

Situation: After Manual Coding (n=282, 17%)

State of Code When Using Codex After Manual Coding:

- After Mostly Incorrect Code: 27%
- After Mostly Correct Code with Minor Issues: 41%
- After Writing Mostly Correct Code: 36%

Self-Regulated
RQ1 A When did Learners Use Codex?

Situation: After Using Codex ($n=572, 34\%$)

- Decomposing Tasks into Multiple Subgoals: Write Next Subgoal with Codex

243 Codex Usages (15\%)

```python
import random
pivot = random.randint(1, 100)

# PROMPT: ask the user to enter a number
num = int(input("guess a number"))
```

84 Codex Usages (5\%)

```python
import random
num1 = random.randint(1, 6)

# PROMPT: generate another random number
num2 = random.randint(1, 6)
```
Situation: After Using Codex (n=572, 34%)

- Decomposing Tasks into Multiple Subgoals: Write Next Subgoal with Codex

<table>
<thead>
<tr>
<th>243 Codex Usages (15%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 import random</td>
</tr>
<tr>
<td>pivot = random.randint(1, 100)</td>
</tr>
<tr>
<td>2 # PROMPT: ask the user to enter a number</td>
</tr>
<tr>
<td>num = int(input(&quot;guess a number&quot;))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>84 Codex Usages (5%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 import random</td>
</tr>
<tr>
<td>num1 = random.randint(1, 6)</td>
</tr>
<tr>
<td>2 # PROMPT: generate another random number</td>
</tr>
<tr>
<td>num2 = random.randint(1, 6)</td>
</tr>
</tbody>
</table>

Over-Reliance
Situation: Already Having the Solution (n=16, 1%)
What did Learners Ask from Codex?

Focus of Thematic Analysis:
• What parts of the task?
• Requesting Syntax or Logic?
What are they asking for using Codex?

- Generate the Entire Solution: 43%
- Generate New Subgoals: 37%
- Fix Existing Code: 7%
RQ1 B

What did Learners Ask from Codex?

What are they asking for using Codex?

- Generate the Entire Solution: 43%
- Generate New Subgoals: 37%
- Fix Existing Code: 7%
What did Learners Ask from Codex?

What are they asking for using Codex?

- Generate the Entire Solution: 43%
- Generate New Subgoals: 37%
- Fix Existing Code: 7%
What did Learners Ask from Codex?

What are they asking for using Codex?

- Generate the Entire Solution: 43%
- Generate New Subgoals: 37%
- Fix Existing Code: 7%
RQ1 B

**What did Learners Ask from Codex?**

When Decomposing Task into Subgoals

![Bar chart showing the percentage of learners asking for completely new subgoals vs similar to existing code.

- Completely New Subgoal: 21%
- Similar to Existing Code: 79%

![Bar chart showing the percentage of learners asking for pure syntax vs syntax and logic.

- Pure Syntax: 15%
- Syntax and Logic: 85%]
RQ1 B

What did Learners Ask from Codex?

When Decomposing Task into Subgoals

- 21% Completely New Subgoal
- 79% Similar to Existing Code

- 15% Pure Syntax
- 85% Syntax and Logic
Novice Learners’ Prompt Properties

Focus of Thematic Analysis:
• Prompt Content
• Vagueness
• Relationship to Task Description
Five Primary Properties:

- Copy Entire Task Description: 52%
- Partial Copy from Task Description: 14%
- Reworded Versions of Task Description: 21%
- Included Less Detail or Vague Interpretations of Task: 13%
- Incorrectly Interpreted Task: 2%
Five Primary Properties:

- Copy Entire Task Description: 52%
- Partial Copy from Task Description: 14%
- Reworded Versions of Task Description: 21%
- Included Less Detail or Vague Interpretations of Task: 13%
- Incorrectly Interpreted Task: 2%
Five Primary Properties:

- Copy Entire Task Description: 52%
- Partial Copy from Task Description: 14%
- Reworded Versions of Task Description: 21%
- Included Less Detail or Vague Interpretations of Task: 13%
- Incorrectly Interpreted Task: 2%
Five Primary Properties:

- Copy Entire Task Description: 52%
- Partial Copy from Task Description: 14%
- Reworded Versions of Task Description: 21%
- Included Less Detail or Vague Interpretations of Task: 13%
- Incorrectly Interpreted Task: 2%
Five Primary Properties:

- Copy Entire Task Description: 52%
- Partial Copy from Task Description: 14%
- Reworded Versions of Task Description: 21%
- Included Less Detail or Vague Interpretations of Task: 13%
- Incorrectly Interpreted Task: 2%
Five Primary Properties:

- Copy Entire Task Description: 52%
- Partial Copy from Task Description: 14%
- Reworded Versions of Task Description: 21%
- Included Less Detail or Vague Interpretations of Task: 13%
- Incorrectly Interpreted Task: 2%
Novice Learners’ Prompt Properties

Prompts Similar to Pseudo-Code (n=89, 5%)

Prompt 1: “for n in numbers, if n > l, set l to n”

Prompt 2: “print Largest number: l”

Prompt: “find the largest number”
Utilizing AI-Generated Code

Focus of Thematic Analysis:
• Placement of AI-Generated Code
• Modifying Existing or Generated Code
• Testing and Verifying Code
Verifying: Tinkering with AI-Generated Code

```python
# PROMPT: print 1st message in list
print(numbers[0])

print(numbers[1])

print(numbers[0])
```
Verifying: Running and Testing AI-Generated Code

Common Behaviors When Using Codex at The Beginning:

- Properly Tested AI-Generated Code
- Deleted AI Code
- Moved Immediately to Use Codex for Next Subgoal
- Submitted AI-Generated Code Without Running
Verifying: Running and Testing AI-Generated Code

Common Behaviors When Using Codex at The Beginning:

- Properly Tested AI-Generated Code
- Deleted AI Code
- Moved Immediately to Use Codex for Next Subgoal
- Submitted AI-Generated Code Without Running
Common Behaviors When Using Codex at The Beginning:

- Properly Tested Al-Generated Code
- Deleted Al Code
- Moved Immediately to Use Codex for Next Subgoal
- Submitted Al-Generated Code Without Running
Common Behaviors When Using Codex at The Beginning:

- Properly Tested AI-Generated Code
- Deleted AI Code
- Moved Immediately to Use Codex for Next Subgoal
- Submitted AI-Generated Code Without Running
Common Behaviors *When Using Codex at The Beginning:*

- Properly Tested AI-Generated Code
- Deleted AI Code
- Moved Immediately to Use Codex for Next Subgoal
- Submitted AI-Generated Code Without Running

Graph showing the distribution of these behaviors.
Verifying: Manually Adding Code to Verify

1. Prompt Codex
   > generate two random numbers between 1 and 6 and check both if they are greater than 3

2. Generated Code + Placed
   ```python
   import random
   roll1 = random.randint(1, 6)
   roll2 = random.randint(1, 6)
   if roll1 > 3 and roll2 > 3:
       print("Both greater than 3")
   ```

3. Added Verification Code
   ```python
   import random
   roll1 = random.randint(1, 6)
   roll2 = random.randint(1, 6)
   if roll1 > 3 and roll2 > 3:
       print("Both greater than 3")
   ```

Student added a new line
```python
print(roll1, ",", roll2)
```
Verifying: Manually Adding Code to Verify

1. Prompt Codex

> generate two random numbers between 1 and 6 and check both if they are greater than 3

2. Generated Code + Placed

```
import random
roll1 = random.randint(1, 6)
roll2 = random.randint(1, 6)
if roll1 > 3 and roll2 > 3:
    print("Both greater than 3")
```

3. Added Verification Code

```
import random
roll1 = random.randint(1, 6)
roll2 = random.randint(1, 6)
if roll1 > 3 and roll2 > 3:
    print("Both greater than 3")
    print(roll1, ",", roll2)
```

Student added a new line `print(roll1, ",", roll2)` to verify the AI-generated code.

😊 Self-Regulation
AI Code Generator Coding Approaches
Manual (without Codex)

The final submitted code was 100% manually written.

29% tasks
Manual (without Codex)

The final submitted code was 100% manually written.

29% tasks

AI Step-by-Step

Decomposed task into multiple, consecutive Codex usages, with no manual coding

6% tasks
Hybrid
A few subgoals were AI-generated, while other subgoals were written manually
19% tasks

Manual (without Codex)
The final submitted code was 100% manually written.
29% tasks

AI Step-by-Step
Decomposed task into multiple, consecutive Codex usages, with no manual coding
6% tasks
AI Single Prompt
Use a single prompt (either by copying the task, or rewording) to solve the entire task
46% tasks

Hybrid
A few subgoals were AI-generated, while other subgoals were written manually
19% tasks

Manual (without Codex)
The final submitted code was 100% manually written.
29% tasks

AI Step-by-Step
Decomposed task into multiple, consecutive Codex usages, with no manual coding
6% tasks

Manual (without Codex)
The final submitted code was 100% manually written.
29% tasks

AI Step-by-Step
Decomposed task into multiple, consecutive Codex usages, with no manual coding
6% tasks

Hybrid
A few subgoals were AI-generated, while other subgoals were written manually
19% tasks

AI Single Prompt
Use a single prompt (either by copying the task, or rewording) to solve the entire task
46% tasks
What is the Relationship between Authoring and Modifying Tasks for Each Coding Approach?

Self-Paced Python Training
7 Sessions
Relationship between Authoring and Modifying Tasks

- **AI Single Prompt**
  - Authoring: 96%
  - Modifying: 62%

- **AI Step-by-Step**
  - Authoring: 76%
  - Modifying: 55%

- **Hybrid**
  - Authoring: 77%
  - Modifying: 73%

- **Manual**
  - Authoring: 63%
  - Modifying: 73%
Relationship between Authoring and Modifying Tasks

- **AI Single Prompt**: 96% Authoring, 62% Modifying
- **AI Step-by-Step**: 76% Authoring, 55% Modifying
- **Hybrid**: 77% Authoring, 73% Modifying
- **Manual**: 63% Authoring, 73% Modifying
Relationship between Authoring and Modifying Tasks

- **AI Single Prompt**
  - Authoring: 96%
  - Modifying: 62%

- **AI Step-by-Step**
  - Authoring: 76%
  - Modifying: 55%

- **Hybrid**
  - Authoring: 77%
  - Modifying: 73%

- **Manual**
  - Authoring: 63%
  - Modifying: 73%
Relationship between Authoring and Modifying Tasks

- **AI Single Prompt**
  - Authoring: 96%
  - Modifying: 62%
- **AI Step-by-Step**
  - Authoring: 76%
  - Modifying: 55%
- **Hybrid**
  - Authoring: 77%
  - Modifying: 73%
- **Manual**
  - Authoring: 63%
  - Modifying: 73%

Legend:
- Blue: Authoring
- Orange: Modifying
Relationship between Authoring and Modifying Tasks

- **AI Single Prompt**: 96% Authoring, 62% Modifying
- **AI Step-by-Step**: 77% Authoring, 73% Modifying
- **Hybrid**: 76% Authoring, 55% Modifying
- **Manual**: 63% Authoring, 73% Modifying
What is the Relationship between utilizing each of the Coding Approaches and Post-Test Evaluation Tests?

1. Immediate Post-Test
- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

2. Retention Post-Test
- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

one week later

No Python Documentation  *  No Instructor Hints  *  No AI Code Generators
Utilization of Each Coding Approach

Each Student during Training:

45 Authoring Tasks (7 Training Sessions)  Evaluation Post-Tests

Coding Approaches:
- Manual (without Codex)
- AI Step-by-Step
- Hybrid
- AI Single Prompt
Correlation Analysis:
Utilization of Coding Approach with Post-Test Evaluation Score

33 dots, each representing a student

Evaluation Score

Utilization of Coding Approach During Training

p-value = 0.159
AI Single Prompt

Hybrid

Manual

AI Step-by-Step

😊 Negative Correlation

😘 Positive Correlation
## KEY TAKEAWAYS

### Signs of Self-Regulation
- Attempting manual coding before using Codex and using the **Hybrid AI Coding Approach**
- Prompting Codex mainly for syntax
- Actively adding code to verify AI-generated code
- Tinkering with AI-generated code to understand it

### Signs of Over-Reliance
- Frequent usage of the AI Single Prompt approach
- Copying the task description and submitting generated code without any editing
- Prompting Codex for code similar to existing code
- Over-trust: submitting code without running
Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming (*CHI*’23)

Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, Tovi Grossman

How Novices Use LLM-based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment (*Koli Calling*’23)

Majeed Kazemitabaar, Xinying Hou, Austin Z. Henley, Barbara J. Ericson, David Weintrop, Tovi Grossman
We developed an LLM-powered pedagogical Assistant named CodeAid with five main features that responds to various programming-related questions and help requests.

Unlike unmoderated LLMs, CodeAid produces responses without revealing direct code solutions. Instead, it helps students by producing pseudo-code, suggested fixes and natural language responses.

We deployed CodeAid at a large class of 700 students, interviewed 22 students about their usage, and then interviewed 8 computing educators.

Our results help guide the design of future AI-powered assistants for educational settings.
## Design Goals of Educational AI Assistants

<table>
<thead>
<tr>
<th>D1: Exploiting Unique Advantages of AI</th>
<th>D2: Designing the AI Querying Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Help-Seeking Choice:</strong> Al assistant vs. Traditional resources</td>
<td><strong>Problem Identification:</strong> Student (reactive) vs. Tool (proactive)</td>
</tr>
<tr>
<td><strong>AI Tool’s Goal:</strong> Productivity-focused vs. Learner-centric</td>
<td><strong>Input Format:</strong> Structured vs. Free-form</td>
</tr>
<tr>
<td><strong>Educational Versatility:</strong> General-purpose vs. Course-specific</td>
<td><strong>Context Provision:</strong> Manual vs. Automatic</td>
</tr>
<tr>
<td><strong>Key Trade-Off:</strong> Broad Scope vs. Unique Advantages</td>
<td><strong>Key Trade-Off:</strong> Meta-Cognitive Engagement vs. Ease of Use</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D3: Balancing the Directness of AI Responses</th>
<th>D4: Supporting Trust, Transparency and Control</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Directness Level:</strong> Direct vs. Indirect</td>
<td><strong>Transparency of AI:</strong> Minimal vs. Detailed</td>
</tr>
<tr>
<td><strong>Scaffolding Type:</strong> Hints vs. Leading Questions</td>
<td><strong>Validation Point:</strong> After Generation vs. Before Generation</td>
</tr>
<tr>
<td><strong>Agency of Control:</strong> Student vs. Instructor vs. AI</td>
<td><strong>AI Steering &amp; Control:</strong> Autonomous vs. User Guided</td>
</tr>
<tr>
<td><strong>Key Trade-Off:</strong> Learning Opportunities vs. Ensuring Progress</td>
<td><strong>Key Trade-Off:</strong> Ease of Use vs. Transparency and Engagement</td>
</tr>
</tbody>
</table>