On the Performance of Large Language Models on Introductory Programming Assignments

Nishat Raihan[1*], Dhiman Goswami[1], Sadiya Sayara Chowdhury Puspo[1], Mohammed Latif Siddiq[2], Christian Newman[3], Tharindu Ranasinghe[4], Joanna C. S. Santos[2], Marcos Zampieri[1]

[1]George Mason University, Fairfax, VA, USA.
[2]University of Notre Dame, Notre Dame, IN, USA.
[3]Rochester Institute of Technology, Rochester, NY, USA.
[4]Lancaster University, Lancaster, UK.

*Corresponding author(s). E-mail(s): mraihan2@gmu.edu;

Abstract

Recent advances in artificial intelligence (AI), machine learning (ML), and natural language processing (NLP) have led to the development of a new generation of Large Language Models (LLMs) trained on massive amounts of data. Commercial applications (e.g., ChatGPT) have made this available to the general public, enabling the use of LLMs to produce high-quality texts for academic and professional purposes. Educational institutions are increasingly aware of students' use of AI-generated content and are researching its impact and potential misuse. Computer Science (CS) and related fields are particularly affected, as LLMs can also generate programming code in various languages. To understand the potential impact of publicly available LLMs in CS education, we extend our previously introduced `CSEPrompts` [31], a framework comprising hundreds of programming exercise prompts and multiple-choice questions from introductory CS and programming courses. We provide experimental results on `CSEPrompts`, evaluating the performance of several LLMs in generating Python code and answering basic computer science and programming questions, offering insights into the implications of this technology for CS education.

Keywords: Benchmark Dataset, Code LLM, Prompting

1 Introduction

The past decade has witnessed a remarkable evolution in natural language processing (NLP) models. We have progressed from n-gram and word embedding models, such as word2vec [25] and GloVe [28], to sophisticated context-aware models like ELMo [29], and BERT [7]. These advancements have significantly enhanced performance across various NLP tasks [35]. More recently, Large Language Models (LLMs) such as GPT [1, 21] have further revolutionized the field.

Recent studies explored the use of LLMs for assessment in various domains, such as law [17], mathematics and computer science [48], medicine [14], and computer science education [34]. These studies demonstrate the high quality of the output of these models, with some even suggesting that these models could "pass the bar exam"[17]. Such findings underscore the potential impact of LLMs on educational assessment and the need for further research.

The impact of GPT models on education remains a key subject of several recent studies, which include studies conducted by Halaweh [13], Lo [21], Raihan et al. [32], Sok and Heng [39], among others. While these models offer numerous opportunities in educational technology, such as enhanced writing assistants, intelligent tutoring systems, and automatic assessment tools, they also raise concerns about potential misuse, particularly in coding tasks. Savelka et al. [37] find that while GPT scores may not meet course completion criteria, the model exhibits notable capabilities, including the ability to correct solutions based on auto-grader feedback. This capability raises concerns about students potentially exploiting this technology to generate complete essays and programming assignments, thereby artificially inflating their grades. Furthermore, Surameery and Shakor [40] demonstrate that ChatGPT excels in debugging, bug prediction, and explanation, although it has limitations in reasoning and integration.

In this paper, we build upon our original `CSEPrompts` benchmark [31] to introduce `CSEPrompts 2.0`. This extension encompasses four key areas:

1. An increased number of multiple-choice question (MCQ) prompts,
2. Evaluations of newer and more recent models,
3. A thorough statistical analysis on model performances, and
4. A detailed error analysis.

We present a comprehensive evaluation that goes beyond GPT, examining the performance of eight models capable of generating both English text and Python code on introductory CS and programming course assignments. To facilitate reproducibility and ensure further research in this area, we develop `CSEPrompts 2.0` as a robust framework containing 219 programming prompts and 100 MCQs that were carefully collected from coding websites and massive open online courses (MOOCs). This diverse set of prompts allows for a thorough assessment of LLM capabilities in the context of CS education.

In this study, we do not observe a significant difference in performance after adding the new prompts introduced in this extension. Although the expanded MCQ pool does

not substantially change the relative ranking of models, the consistently narrow accuracy range indicates a potential ceiling effect for current introductory-level MCQs. This observation has two implications: (i) when the evaluation target is limited to similar entry-level questions, practitioners may choose among top models based on factors such as cost or latency rather than marginal accuracy gains, and (ii) future dataset construction should incorporate MCQs that demand deeper conceptual reasoning and multi-step distractor analysis to meaningfully differentiate model understanding.

Our investigation addresses the following research questions:

RQ1: How well do state-of-the-art LLMs perform on introductory CS assignments compared to existing benchmarks?

RQ2: Is there a significant difference in the performance of LLMs when completing assignments from coding websites compared to academic MOOCs?

RQ3: Are state-of-the-art LLMs better at generating code or answering MCQs?

RQ4: Are Code LLMs better at generating code and/or answering MCQs than raw LLMs?

We aim to provide key insights into the capabilities and limitations of LLMs in CS education, informing both educators and researchers about the potential implications of these powerful tools in academic settings.

2 Related Work

2.1 Code Generation Models

Early automated code generation methods concentrated on inferring user intent from high-level specifications or input-output examples [10, 11, 24]. These methods convert task specifications into constraints, and a program is generated once it demonstrates compliance with those constraints [11]. With the advent of attention-based transformer models [42], code generation has evolved into a sequence-to-sequence task, where user intent is expressed through natural language. Most coding tasks involved code completion, code infilling, comment generation, and similar tasks that were often handled using encoder-only models like BERT [7]. Models such as CodeBERT [9], GraphCodeBERT [12], and SynCoBERT [44] are pre-trained on text-code pairs, including Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs) to capture syntactic and semantic code structures. However, encoder-only models are not primarily designed for generative tasks and exhibit subpar performance in code generation [44].

The emergence of generative models based on encoder-decoder architectures, such as CodeT5 [45], and decoder-only architectures, like CodeGen [26], CodeLLaMA [36], StarCoder [19], and domain specific models like Mojo-Coder [33] has significantly improved code generation capabilities. Zan et al. [47] conduct a comprehensive survey, highlighting the superior performance of these models in code generation tasks. With these advancements, the need for unified benchmarks to evaluate and compare code generation models has become more pronounced.

2.2 Code Generation Benchmarks

Several benchmarks have been introduced to assess the performance of code generation models. HumanEval, introduced alongside OpenAI's Codex model [5], and MBPP (Mostly Basic Python Problems) [2] are among the most widely used. These datasets contain coding prompts paired with human-generated solutions and three test cases for each task. Other benchmarks include CONCODE [15], and extensions like HumanEval+ [20], and mHumanEval [30].

Recently, Large Language Models (LLMs) like GPT-3 [4], GPT-4 [1], and fine-tuned code models like CodeLLaMA [36] and StarCoder [19] have demonstrated remarkable code generation abilities. These models are evaluated on benchmarks like HumanEval and MBPP, consistently outperforming previous models. For instance, Roziere et al. Roziere et al. [36] show that CodeLLaMA achieves state-of-the-art results on HumanEval, demonstrating the effectiveness of decoder-only architectures for code generation. Siddiq et al. [38] analyzed these benchmark datasets and found several quality issues, such as insufficient contextual information.

In addition to code generation, other related tasks, such as code completion, which involves predicting the next token or sequence of tokens in code, have been extensively explored. Models like GPT-J [43], GPT-NeoX [3], and PaLM-Coder [6] are applied to code completion tasks using prompts longer than one sentence. Svyatkovskiy et al. [41] introduce IntelliCode Compose, a transformer-based model for real-time code completion, emphasizing the importance of handling multi-line code completions.

Despite these advancements, existing datasets and benchmarks primarily focus on general-purpose coding tasks relevant to software development but do not adequately address educational coding tasks. Educational coding tasks often require a deep understanding of specific programming language syntax and semantics, evaluating the learner's comprehension of fundamental concepts. These tasks differ significantly from the prompts included in existing benchmarks.

To bridge this gap, we introduce `CSEPrompts 2.0`, an extension of our previous work [31]. Our framework provides diverse programming exercise prompts and multiple-choice questions retrieved from introductory CS and programming courses. Each programming prompt is paired with five test cases, compared to three in most benchmarks, offering a more rigorous evaluation of code correctness.

While significant progress has been made in code generation and related tasks using LLMs, benchmarks focusing on educational coding tasks remain needed. `CSEPrompts 2.0` addresses this need by providing a comprehensive framework for evaluating LLMs on introductory CS assignments, thereby contributing to the understanding of LLMs' potential in educational contexts.

3 `CSEPrompts 2.0`

We introduce `CSEPrompts 2.0`[1], an enhanced evaluation framework that extends the original CSEPrompts dataset. The key features are as follows:

- **Retained from the original CSEPrompts:**
  - 219 programming prompts (118 from coding websites and 101 from MOOCs)

[1] https://github.com/mraihan-gmu/CSEPrompts

  - Comprehensive test cases for each programming prompt
  - Coverage of topics ranging from basic syntax and control structures to complex algorithmic problems
- **New in CSEPrompts 2.0:**
  - Doubling the multiple-choice questions from 50 to 100, resulting in a total of 319 exercise prompts (see Table 1)
  - A broader range of challenges and assessment formats designed to better evaluate theoretical understanding and practical knowledge

3.1 Data Sources

The prompts are collected from a diverse set of online resources. All twelve of them are listed in Appendix A with their respective website links.

Coding Websites

We curated our dataset of introductory Python exercises from five leading coding platforms—CodingBat, LearnPython, Edabit, Python Principles, and HackerRank—using a detailed set of inclusion and exclusion criteria designed to ensure methodological transparency and replicability. Inclusion required that each exercise present a well-defined problem statement with clear input/output specifications and a focus on core programming constructs, while being self-contained. In contrast, exercises were excluded if they involved additional complexities, such as File I/O operations or Command-Prompt/Terminal interactions, given the limitations of Language Learning Models (LLMs). As shown in Table 1, this rigorous selection process yielded a final dataset of 118 carefully curated prompts. This methodological framework is similarly tailored for other source categories, ensuring that future extensions or applications to different contexts remain replicable.

Table 1: Summary of Coding Prompts [MCQs: v2.0 (v1.0)].

Coding Websites		**MOOCs - Coding Prompts**			**MOOCs - MCQs**		
Platform	Prompts	University	Course	Prompts	University	Course	Prompts
CodingBat	24	Harvard	CS50	29	GT	CS1301xI	20 (20)
LearnPython	16	UMich	PforE	7	GT	CS1301xII	20 (8)
Edabit	29	GT	CS1301xI	11	GT	CS1301xIII	16 (6)
Python Principles	26	GT	CS1301xII	20	GT	CS1301xIV	16 (16)
HackerRank	23	GT	CS1301xIII	17	Meta	Programming in Python	28 (—)
Total	**118**		**Total**	**101**		**Total**	**100 (50)**

MOOCs

The framework also preserves the 101 programming prompts drawn from MOOCs offered by Harvard University, the University of Michigan, and the Georgia Institute of Technology. In contrast, the multiple-choice questions have been significantly expanded. In the original CSEPrompts, 50 MCQs were included (all from GT courses). In `CSEPrompts 2.0`, the number of MCQs has been increased to 100. This update is achieved by revising the counts for existing courses (for example, GT's CS1301xII

increased from 8 to 20 and CS1301xIII from 6 to 16) and by adding a new source, Meta's *Programming in Python* (see Table 1). These changes provide a more balanced assessment of programming theory and practice.

3.2 Dataset Statistics

The prompts from coding sites are generally shorter than those from MOOCs, as shown in Table 2. For each prompt, we collect a minimum of 5 test cases, primarily from the source platforms. When necessary, we supplement with additional test cases generated using Pynguin [22], an open-source unit test generator for Python. To ensure the quality and relevance of Pynguin-generated tests, we manually review them, focusing on edge cases and comprehensive code coverage. For MCQs, we obtain correct answers from the original platforms. We collect LLM-generated responses for each prompt, manually clean them to isolate code snippets, and label them based on the number of passed test cases. We present a few prompts for each subset of `CSEPrompts 2.0` in Appendix B.

Table 2: Statistics for Prompts [MCQs: v2.0 (v1.0)]

Metric	CodingSites	Academic	MCQ
Total Prompts	118	101	100 (50)
Max. No. of Tokens	101	372	221 (199)
Min. No. of Tokens	5	17	15 (7)
Mean No. of Tokens	28	158	106 (95)
Standard Deviation	16	72	51 (47)

3.3 Data Collection Strategy

Our dataset is built from real programming assignments sourced from academic courses and coding websites. Unlike benchmarks such as HumanEval [**?**] or MBPP [2], which are designed solely to test code generation, our approach reflects the actual challenges encountered in classroom settings.

In the original version, we manually collected coding prompts, ensuring that no duplicates were included and maintained strict inclusion-and-exclusion criteria for the coding prompts only (see Appendix C, Table C2). For the second version (2.0), we compile a new set of criteria applicable for MCQ prompts as we have expanded the dataset by adding multiple-choice question (MCQ) prompts.

This extension was carried out with the following considerations:

- **Expanded Scope:** Although the coding prompts are unchanged, the addition of MCQ prompts broadens the dataset to better capture the range of academic questions students face.
- **Selective Completeness:** The current 2.0 extraction represents all the qualified MCQ prompts (see Appendix C, Table C3) from the selected sources at the time of this report.
- **Manual Verification:** As with the original data collection, all prompts were manually reviewed to ensure no duplication and to maintain quality.

This extension focuses on increasing the breadth of the dataset, and by including a more diverse set of academic prompts, we provide a dataset that more accurately reflects the challenges encountered in computer science education, especially with MCQ prompts that are underexplored in the domain.

4 Experiments

4.1 Large Langauge Models (LLMs)

We experiment with eight different LLMs that represent diverse architectural designs and implementations. Our selection is based on their exceptional performance across established leaderboards maintained by respected research communities: the EvalPlus LeaderBoard [20], AllenAI's WildBench[2] and HuggingFace's BigCode LLM[3].

Table 3: Models evaluated on CSEPrompts 2.0

Model	Open/Close	Parameters	Version	Ref
GPT-4o	Closed	–	4.0	[27]
GPT3.5	Closed	–	3.5	[1]
Llama-3	Open	8B	3.1	[8]
Mistral	Open	7B	0.1	[16]
Code-Llama	Open	13B	1.0	[36]
StarCoder	Open	7B	2.0	[19]
MagiCoder	Open	7B	S-DS-6	[46]
WizardCoder	Open	15B	1.0	[23]

In evaluating CSEPrompts 2.0, we examine a diverse range of language models, including proprietary systems from OpenAI, open-source base models, and specialized systems fine-tuned for code generation. This selection, which highlights different architectures and optimization strategies, enables us to assess both general NLP capabilities and targeted programming performance. Table 3 provides a summary of each model's key attributes, such as openness, parameter details, version, and relevant references.

4.2 Code Generation

We prepare prompts and test each model on tasks from `CSEPrompts 2.0`. Figure 1 and 2 show the simple prompt format used for the models. The generated responses, including code, pseudo-code, and explanations, are manually cleaned to isolate the code. We then evaluate these codes using pytest[4], which enables efficient creation of readable Python unit tests.

4.3 Evaluation Metric

In our work, we employ the `pass@1` metric, a variant of *pass@k* [5]. *Pass@k* metric evaluates the probability that *at least one* out of k generated samples are *functionally*

[2]https://huggingface.co/spaces/allenai/WildBench
[3]https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard
[4]docs.pytest.org/en/7.4.x/

'You are a helpful AI assistant. You are given the following problem: '

Write a function named capital_indexes. The function takes a single parameter, which is a string. Your function should return a list of all the indexes in the string that have capital letters.

'Please write a Python code snippet to solve the problem. Thanks.'

Fig. 1: Sample Prompt for Coding Tasks; includes 3 parts - a system prompt, the task description, and the instruction.

'You are a helpful AI assistant. You are given a Multiple Choice Question. '

(False and True) or (False or True)
Is this statement resolved to True or False?

- **True**
- **False**
- **Statement will not compile**

'You need to pick one or multiple correct answers from the given ones. Thanks.'

Fig. 2: Sample Prompt for MCQs; includes 3 parts - a system prompt, the MCQ with multiple choices, and the instruction.

correct (i.e., passed all *functional* test cases). To evaluate the `pass@k`, we generate n samples per prompt ($n \geq k$), count the number of samples c that are functionally correct ($c \leq n$), and calculate the unbiased estimator $\mathbb{E}$ by Kulal et al. [18]:

$$pass@k = \mathbb{E}_{prompts}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{1}$$

We use `pass@1` in this work, which measures how often a model passes all test cases on its first attempt.

5 Results and Analysis

5.1 Coding Tasks

For our analysis, we employ the `pass@1` metric. Figure 3 illustrates the performance across different models and prompt sources. Proprietary models (GPT-4o and GPT3.5) demonstrate superior performance on both MOOC and CodingSite prompts, followed by code-finetuned models. Notably, all models consistently perform better on CodingSite prompts compared to MOOC prompts, suggesting a difference in prompt complexity or structure between these sources. This performance gap may be attributed to the more structured nature of coding website problems compared to the potentially broader, more conceptual requirements of academic assignments.

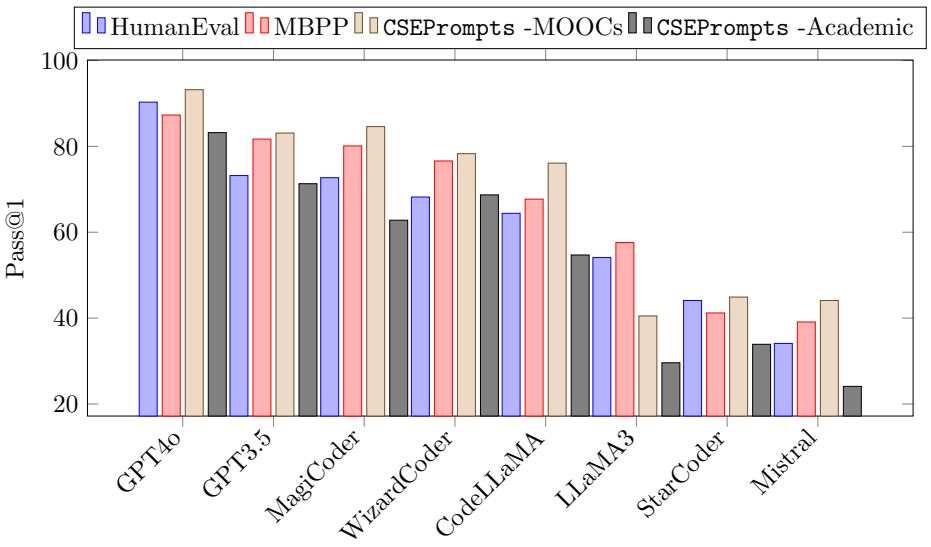

Fig. 3: Comparing `CSEPrompts` with HumanEval and MBPP based on **Pass@1**.

To contextualize our results, we compare `CSEPrompts 2.0` with other widely used benchmarks. Our analysis reveals that the difficulty level of `CSEPrompts 2.0` falls between that of HumanEval and MBPP, providing a balanced challenge for evaluating code generation capabilities. This intermediate positioning makes our benchmark particularly suitable for assessing both the basic and advanced capabilities of code generation models in educational contexts.

5.1.1 Summary Statistics

For each model, the Pass@1 scores were measured on four benchmarks: HumanEval, MBPP, `CSEPrompts` – MOOCs, and `CSEPrompts` – Academic. Table 4 summarizes the mean and standard deviation for each model.

Table 4: Summary Statistics of Pass@1 Scores

Model	Mean Pass@1 (%)	Sample Std. Dev.
GPT4o	88.5	4.28
GPT3.5	77.33	5.94
MagiCoder	75.05	9.53
WizardCoder	72.95	5.25
CodeLLaMA	65.73	5.32
LLaMA3	45.45	12.88
StarCoder	41.03	5.01
Mistral	35.35	8.54

The summary statistics reveal clear differences in performance among the models. GPT4o shows the highest mean score with low variability, indicating robust performance across benchmarks. In contrast, models such as Mistral and StarCoder exhibit lower means and, in some cases, higher variability. Based on these observations, we decide that the initial descriptive analysis is sufficient to suggest that there is a systematic difference among the models. This justifies proceeding to a formal significance test.

5.1.2 Comparison Across Models: Friedman Test

To evaluate whether the differences in Pass@1 scores among the eight evaluated models—namely, GPT4o, GPT3.5, MagiCoder, WizardCoder, CodeLLaMA, LLaMA3, StarCoder, and Mistral—are statistically significant, we applied the Friedman test. Since all models are evaluated on the same four benchmarks, this non-parametric test is appropriate for repeated measures.

For each benchmark, the models were ranked from 1 (lowest score) to 8 (highest score). For instance, for HumanEval the ranking might be: GPT4o (8), GPT3.5 (7), MagiCoder (6), WizardCoder (5), CodeLLaMA (4), LLaMA3 (3), StarCoder (2), and Mistral (1). The sum of ranks for each model is computed across all benchmarks (see Table 5).

Table 5: Sum of Ranks for Each Model

Model	Sum of Ranks
GPT4o	32
GPT3.5	27
MagiCoder	24
WizardCoder	21
CodeLLaMA	16
LLaMA3	9
StarCoder	10
Mistral	5

The Friedman test statistic is given by:

$$\chi_F^2 = \frac{12}{nk(k+1)} \sum_{j=1}^{k} R_j^2 - 3n(k+1), \tag{2}$$

where $n = 4$ is the number of benchmarks and $k = 8$ is the number of models. With $\sum R_j^2 = 3232$, we obtain:

$$\chi_F^2 = \frac{12}{4 \cdot 8 \cdot 9} \times 3232 - 3 \times 4 \times 9 \approx 26.67. \tag{3}$$

With $k - 1 = 7$ degrees of freedom, this test statistic is significant at $p < 0.01$.

The Friedman test confirms that the differences in performance among the models are statistically significant. The test statistic of approximately 26.67, which exceeds

the critical value for 7 degrees of freedom at $p < 0.01$, provides strong evidence that the variation in Pass@1 scores across the four benchmarks is not due to random chance. Notably, GPT4o consistently outperforms the others, while models such as StarCoder and Mistral tend to have lower scores.

5.1.3 Comparison Across Benchmarks: Friedman Test

In addition to comparing models, we examine the performance differences across the four benchmarks: HumanEval, MBPP, `CSEPrompts` – MOOCs, and `CSEPrompts` – Academic. In this analysis, each model is treated as a block and the benchmarks as treatments. For each model, the benchmarks are ranked from 1 (lowest score) to 4 (highest score). For example, for GPT4o the scores are: 90.3 (HumanEval), 87.3 (MBPP), 93.2 (`CSEPrompts` – MOOCs), and 83.2 (`CSEPrompts` – Academic), yielding ranks of 3, 2, 4, and 1, respectively.

After computing the rankings for all models, we obtain the following sums of ranks for each benchmark:

1. **HumanEval:** Sum of ranks = 18
2. **MBPP:** Sum of ranks = 23
3. `CSEPrompts` – **MOOCs:** Sum of ranks = 30
4. `CSEPrompts` – **Academic:** Sum of ranks = 9

Here now $n = 8$ (the number of models) and $k = 4$ (the number of benchmarks). With the sum of squares of the benchmark ranks computed as

$$\sum R_j^2 = 18^2 + 23^2 + 30^2 + 9^2 = 324 + 529 + 900 + 81 = 1834, \tag{4}$$

the test statistic becomes:

$$\chi_F^2 = \frac{12}{8 \cdot 4 \cdot 5} \times 1834 - 3 \times 8 \times 5 \approx \frac{12}{160} \times 1834 - 120 \approx 17.55. \tag{5}$$

With $k-1 = 3$ degrees of freedom, the Friedman test yields a statistic of approximately 17.55, which is significant at $p < 0.01$.

The Friedman test for the benchmarks indicates that the differences in model performance across the four datasets are statistically significant. A test statistic of approximately 17.55, exceeding the critical value for 3 degrees of freedom at $p < 0.01$, confirms that the benchmarks vary in difficulty. Importantly, the `CSEPrompts` – MOOCs benchmark shows the highest average rank, suggesting that it is the easiest among the four. In contrast, the `CSEPrompts` – Academic benchmark has the lowest average rank, indicating that it is the most challenging.

5.2 MCQ Tasks

We compare our Multiple Choice Question (MCQ) task results with the MathQA-Python benchmark [2], which contains coding-related question-answer pairs. The MCQ subset of `CSEPrompts 2.0` introduces the first Code-MCQ benchmark in this domain,

addressing a significant gap in the evaluation of LLMs' comprehension of programming concepts. Figure 4 illustrates LLM performance on both datasets. Our analysis reveals that models generally find MCQ tasks easier than open-ended QA tasks, likely due to the additional context and limited answer set guiding responses. Notably, while proprietary models excel in this task, code-finetuned models underperform, possibly due to their specialization in structured code generation rather than MCQ-style prompts. This performance disparity suggests that the ability to generate code does not necessarily translate to strong performance in understanding and answering questions about programming concepts, highlighting an important distinction in model capabilities.

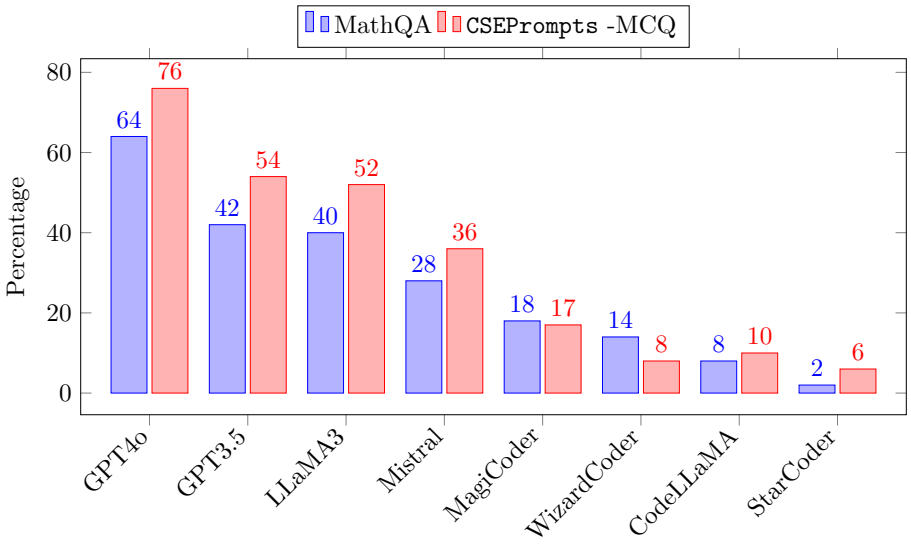

Fig. 4: Comparing `CSEPrompts` -MCQ with MathQA based on Zero Shot Prompting (in percentage).

5.2.1 Summary Statistics

The performance of eight models was evaluated on two benchmarks, MathQA and `CSEPrompts` – MCQ, under zero-shot prompting. The reported scores (in percentage) are shown in Figure 4. For each model, we compute the mean performance and the sample standard deviation across the two benchmarks. Table 6 summarizes these statistics for each model.

The summary statistics reveal substantial variation in performance across models. GPT4o shows the highest mean performance, while StarCoder registers the lowest. Models such as MagiCoder and CodeLLaMA exhibit very low variability between benchmarks, suggesting consistent behavior, whereas others (e.g., GPT4o) show larger deviations. These descriptive results motivate further inferential testing to assess whether the observed differences are statistically significant.

Table 6: Summary Statistics of Zero-Shot Performance (%)

Model	Mean (%)	Std. Dev.
GPT4o	70.0	8.49
GPT3.5	48.0	8.49
LLaMA3	46.0	8.49
Mistral	32.0	5.66
MagiCoder	17.5	0.71
WizardCoder	11.0	4.24
CodeLLaMA	9.0	1.41
StarCoder	4.0	2.83

5.2.2 Comparison Across Models: Friedman Test

To determine if the differences among the eight models are statistically significant, we apply the Friedman test, treating the two benchmarks as blocks. For each benchmark, the models are ranked from 1 (lowest score) to 8 (highest score). For example, the rankings for the two benchmarks are:

- **MathQA:** GPT4o (8), GPT3.5 (7), LLaMA3 (6), Mistral (5), MagiCoder (4), WizardCoder (3), CodeLLaMA (2), StarCoder (1).
- `CSEPrompts` – **MCQ:** GPT4o (8), GPT3.5 (7), LLaMA3 (6), Mistral (5), MagiCoder (4), CodeLLaMA (3), WizardCoder (2), StarCoder (1).

Thus, the sum of ranks for each model (summing the two benchmarks) is:

$$R_{\text{GPT4o}} = 16, \quad R_{\text{GPT3.5}} = 14, \quad R_{\text{LLaMA3}} = 12, \quad R_{\text{Mistral}} = 10,$$

$$R_{\text{MagiCoder}} = 8, \quad R_{\text{WizardCoder}} = 6, \quad R_{\text{CodeLLaMA}} = 4, \quad R_{\text{StarCoder}} = 2.$$

Now, $n = 2$ (benchmarks) and $k = 8$ (models). With

$$\sum R_j^2 = 16^2 + 14^2 + 12^2 + 10^2 + 8^2 + 6^2 + 4^2 + 2^2 = 816, \tag{6}$$

we have:

$$\chi_F^2 = \frac{12}{2 \times 8 \times 9} \times 816 - 3 \times 2 \times 9 = \frac{12}{144} \times 816 - 54 = 68 - 54 = 14. \tag{7}$$

With 7 degrees of freedom, a test statistic of 14 is marginally below the critical value at the 0.05 significance level (approximately 14.07), indicating borderline significance.

The Friedman test for models yields a statistic of 14 with 7 degrees of freedom, which is at the threshold of significance. Although the overall ranking indicates that GPT4o outperforms the other models, the evidence is marginal.

5.2.3 Comparison Across Benchmarks: Friedman Test

To assess whether the performance differences between the two benchmarks (MathQA and `CSEPrompts` – MCQ) are statistically significant, we treat each model as a block and the benchmarks as treatments. For each model, the two benchmark scores are

ranked from 1 (lower performance) to 2 (higher performance). Table 7 summarizes these rankings.

Table 7: Benchmark Rankings for Each Model

Model	MathQA Rank	CSEPrompts – MCQ Rank
GPT4o	1	2
GPT3.5	1	2
LLaMA3	1	2
Mistral	1	2
MagiCoder	2	1
WizardCoder	2	1
CodeLLaMA	1	2
StarCoder	1	2

From Table 7, the sum of ranks for each benchmark is computed as follows:

$$R_{\text{MathQA}} = 1 + 1 + 1 + 1 + 2 + 2 + 1 + 1 = 10, \tag{8}$$

$$R_{\texttt{CSEPrompts} - \text{MCQ}} = 2 + 2 + 2 + 2 + 1 + 1 + 2 + 2 = 14. \tag{9}$$

With $n = 8$ (models) and $k = 2$ (benchmarks), the Friedman test statistic is calculated using:

$$\chi_F^2 = \frac{12}{n\,k\,(k+1)} \sum_{j=1}^{k} R_j^2 - 3n(k+1). \tag{10}$$

Substituting the values:

$$\sum R_j^2 = 10^2 + 14^2 = 100 + 196 = 296, \tag{11}$$

$$\chi_F^2 = \frac{12}{8 \times 2 \times 3} \times 296 - 3 \times 8 \times 3 = \frac{12}{48} \times 296 - 72 = 0.25 \times 296 - 72 = 74 - 72 = 2. \tag{12}$$

With 1 degree of freedom, the critical value at $p = 0.05$ is approximately 3.84, so a test statistic of 2 is not significant.

The Friedman test comparing benchmarks yields a test statistic of 2 (with 1 degree of freedom), which is not significant at the 0.05 level. This indicates that there is no statistically significant difference in performance between the MathQA and `CSEPrompts` – MCQ benchmarks across the eight models. Although individual models such as MagiCoder and WizardCoder show a reversal in ranking between benchmarks, the overall evidence suggests that the benchmarks are comparable in difficulty.

6 Error Analysis

In this section, we present an error analysis of different code generation models in two distinct environments: coding sites and academic settings. Our analysis reveals four major categories of errors consistently encountered by these models: (1) syntax-related issues (*Indentation* and general *Syntax* errors), (2) naming and referencing problems

(*Name*, *Attribute*, and *Key* errors), (3) data handling errors (*Value* and *Type* errors), and (4) runtime issues (*Recursion*, *Import*, and *SystemExit* errors). Tables 8 and 9 show the percentage distribution of these error types for each model in coding sites and academic tasks, respectively. This comprehensive categorization enables a detailed assessment of model performance across different programming contexts.

Table 8: Error Analysis (in percentage): Coding Sites

Error Type	GPT4o	GPT 3.5	Magi Coder	Wizard Coder	Code LLaMA	Star Coder	LLaMA3	Mistral
Name	2.00	2.50	4.00	4.50	5.00	4.50	–	–
Indentation	1.00	3.00	5.00	–	7.00	7.00	8.00	9.00
Value	1.50	4.00	–	–	8.00	8.00	9.00	10.00
Type	2.00	4.50	5.00	6.00	7.00	8.50	–	–
Syntax	–	–	0.50	0.50	8.50	9.50	10.00	11.00
UnboundLocal	–	2.00	2.50	4.00	4.50	5.50	7.00	8.00
Attribute	–	–	–	3.50	4.50	5.00	6.00	7.00
Recursion	–	1.00	1.50	3.50	–	6.00	7.00	8.00
ModuleNotFound	–	1.00	1.50	2.50	3.50	4.50	6.00	7.00
Index	–	1.00	2.00	3.50	4.50	5.50	–	8.00
Key	–	–	2.00	–	–	4.50	6.00	7.00
Import	–	–	2.50	2.00	–	4.00	6.00	7.00
Tab	–	0.50	–	–	2.50	–	6.00	–
System Exit	–	0.50	–	2.00	2.50	3.00	4.00	5.00
Infinite Loop	–	0.50	–	–	2.00	–	4.00	5.00

In the coding sites environment (Table 8), we observe a relatively even distribution of errors across categories, with *Name*, *Indentation*, and *Type* errors emerging as the most prevalent across models. Notably, Magi Coder shows a significant proportion of *Name* errors (4.00%), suggesting challenges with variable naming and scope management. Similarly, GPT 3.5 exhibits notable *Name* errors (2.50%). The more recent models - Star Coder, LLaMA3, and Mistral - demonstrate higher rates of *Syntax* errors, indicating fundamental challenges with code structure and formatting requirements.

Table 9: Error Analysis (in percentage): Academic

Error Type	GPT4o	GPT 3.5	Magi Coder	Wizard Coder	Code LLaMA	Star Coder	LLaMA3	Mistral
Name	1.00	5.25	7.25	3.00	–	3.50	2.00	4.00
Indentation	1.00	2.50	3.75	2.25	2.00	5.50	3.00	2.50
Value	0.50	3.00	2.00	1.00	1.00	3.50	2.50	1.00
Type	2.50	3.50	3.00	3.50	1.50	5.00	4.50	2.00
Syntax	–	–	–	–	–	5.50	6.00	5.75
Attribute	0.50	3.00	2.50	2.25	2.00	3.75	3.25	3.00
Recursion	0.25	1.50	1.75	2.50	2.50	2.00	2.75	2.25
ModuleNotFound	0.25	2.00	1.25	1.75	2.00	3.00	2.25	2.50
Index	0.50	1.50	2.50	1.50	1.00	2.25	3.00	2.75
Key	–	0.50	1.00	1.00	1.00	1.75	1.25	1.50
Import	–	0.75	0.50	1.25	1.50	1.50	2.00	1.75
ZeroDivision	1.00	1.00	1.00	1.00	1.00	–	–	–
FileNotFound	0.50	1.50	0.75	1.00	0.75	–	–	–
Exception	–	0.25	0.50	0.75	0.75	0.25	0.50	0.75

The academic environment (Table 9) presents a markedly different error distribution pattern. Models like Code LLaMA, Star Coder, LLaMA3, and Mistral show elevated rates of *Syntax*, *Value*, and *Type* errors. For instance, Mistral's high *Syntax* error rate (5.75%) and consistent *Value* error occurrences point to difficulties in handling academic code complexity. These patterns suggest that these models struggle

with both advanced programming concepts and the more rigorous syntax requirements typical in academic assignments, particularly when dealing with complex data structures and algorithmic implementations.

Cross-environmental comparison reveals GPT4o's consistent superior performance, maintaining low error rates across all major categories in both settings. This consistency suggests a robust capability in generating correct code regardless of the context. The generally higher error rates in academic settings likely stem from several factors: the presence of more complex algorithms, requirements for specialized libraries, and stricter formatting standards. These elements may be underrepresented in the training data of models like Code LLaMA and Mistral.

Furthermore, academic tasks often demand a deeper grasp of theoretical concepts, potentially exceeding these models' current capabilities. These findings underscore the importance of careful model selection based on the intended application context, particularly in educational settings where code quality and conceptual understanding are equally important.

Sample Errors

To exemplify some of the errors mentioned previously, we present below examples of errors generated by different models:

Prompt: Write a function `mid(s)` that returns the middle character of a string if its length is odd, or the two middle characters if its length is even.

Code generated by CodeLLaMA:

```
def mid(s):
    center = len(s) // 2
    return s[center] + s[center + 1]
```

Error type: `IndexError: string index out of range`

Prompt: Convert the two lists `height_list` and `weight_list` into NumPy arrays named `height_arr` and `weight_arr`.

Code generated by Mistral:

```
import numpyy as np
height_list = [170, 165, 180]
weight_list = [70, 60, 85]
height_arr = np.array(height_list)
weight_arr = np.array(weight_list)
```

Error type: `ModuleNotFoundError: No module named 'numpyy'`

Prompt: Given a NumPy array `BMI_array`, return a list of indices where the BMI value is strictly greater than 23.

Code generated by LLaMA3:

```
import numpy as np
BMI_array = np.array([24.22, 22.04, 26.23])
above = []
for val in BMI_array:
    if val > 23:
        above.append(BMI_array[val])
print(above)
```

Error type: `TypeError`

7 Conclusion and Future Work

In this work, we evaluated the performance of various Large Language Models (LLMs) on introductory computer science tasks, focusing on Multiple Choice Questions (MCQs) and Python programming assignments. We compiled `CSEPrompts 2.0`, a diverse evaluation framework comprising prompts from online coding platforms, academic resources, and programming courses. By analyzing eight state-of-the-art LLMs, we provided detailed performance metrics and error analyses to address four key research questions.

RQ1: How do LLMs perform on introductory CS assignments?

Both our prior and current studies consistently show that state-of-the-art LLMs generate high-quality outputs on introductory computer science assignments. In our prior study [31], all models were shown to perform well on `CSEPrompts` with GPT outperforming the other seven models. The expanded evaluation on `CSEPrompts 2.0` confirms this finding, with GPT-based models (specifically GPT4o and GPT 3.5) demonstrating superior performance. While both studies reveal that LLMs perform better on MOOCs-based prompts compared to traditional benchmarks, the current study adds nuance by emphasizing that academic-style prompts—characterized by their integration of theoretical content and practical coding challenges—continue to pose significant difficulties. This suggests that even as overall performance improves, the complexity of academic content remains a persistent challenge.

RQ2: Is there a performance difference between coding websites and academic MOOCs?

In both analyses, a distinct performance gap is evident between prompts originating from coding websites and those from academic MOOCs. Our earlier study [31] reported that LLMs found coding website prompts easier, while academic MOOC prompts proved more challenging. The current study not only replicates this observation on `CSEPrompts 2.0` but also provides further insight by quantifying higher accuracy and

lower error rates on coding website prompts. In contrast, academic prompts—which often involve abstract concepts and deeper theoretical integration—consistently result in increased error rates. This reinforces the idea that the nature of the content (practical versus academic) significantly influences LLM performance.

RQ3: How do LLMs perform in code generation tasks compared to MCQs?

Contrary to the initial assumption that LLMs would excel in natural language tasks like answering MCQs, both studies reveal a consistent trend: LLMs generate code more reliably than they answer multiple-choice questions. Our earlier study [31] highlighted that, despite LLMs being primarily designed for text generation, they performed better in code generation. The current study confirms and deepens this finding by demonstrating that the structured syntax and clear semantics of programming languages allow LLMs to produce syntactically correct and logically coherent code, whereas the nuanced comprehension required for MCQs leads to comparatively lower performance.

RQ4: How do Code LLMs compare to general-purpose LLMs in CS tasks?

Both studies agree that model specialization plays a significant role in task performance. Our earlier study [31] noted that GPT3.5—a larger, general-purpose model—outperformed other models overall, yet observed that general-purpose LLMs tend to do better on MCQs while Code LLMs excel in coding tasks. The current study expands on this by confirming that while larger general-purpose models (GPT4o and GPT 3.5) achieve the highest overall performance, the specialization of Code LLMs, driven by training on extensive programming repositories, provides them with an edge on coding-specific tasks. This clear division of strengths reinforces the importance of selecting models based on the specific requirements of the task—whether it be theoretical understanding assessed through MCQs or practical coding proficiency.

7.1 Future Directions

The refined insights from `CSEPrompts 2.0` build upon our initial findings by incorporating a wider range of coding prompts and more challenging academic MCQs. Our current work not only corroborates the previous trends—such as superior performance on MOOCs-based prompts and a consistent gap between coding website and academic prompt performance—but also opens new avenues for exploration. Future research will focus on:

- Expanding the dataset to include a diverse set of prompts spanning multiple programming languages and advanced computer science topics.
- Conducting in-depth analyses of code characteristics such as comprehensibility, security, and algorithmic complexity.
- Investigating the underlying factors that contribute to performance disparities between academic and non-academic prompts.

These efforts aim to refine the application of LLMs in educational contexts, enhancing both automated assessment systems and tailored instructional strategies. Ultimately, our work underscores that while current LLMs show remarkable capabilities in coding tasks, careful consideration of prompt type and model specialization is essential for optimizing their deployment in computer science education.

Funding None

References

[1] Achiam, J., S. Adler, S. Agarwal, et al. 2023. Gpt-4 technical report. *arXiv:2303.08774*. https://doi.org/10.48550/arXiv.2303.08774 .

[2] Austin, J., A. Odena, M. Nye, and et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*. https://doi.org/10.48550/arXiv.2108.07732 .

[3] Black, S., L. Gao, P. Wang, et al. 2022. Gpt-neox-20b: An open-source autoregressive language model. *arXiv:2204.06745*. https://doi.org/10.48550/arXiv.2204.06745 .

[4] Brown, T.B., B. Mann, N. Ryder, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*. https://doi.org/10.48550/arXiv.2005.14165 .

[5] Chen, M., J. Tworek, H. Jun, et al. 2021. Evaluating large language models trained on code. *arXiv:2107.03374*. https://doi.org/10.48550/arXiv.2107.03374 .

[6] Chowdhery, A., S. Narang, J. Devlin, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv:2204.02311*. https://doi.org/10.48550/arXiv.2204.02311 .

[7] Devlin, J., M.W. Chang, K. Lee, et al. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL, 10.18653/V1/N19-1423*.

[8] Dubey, A., A. Jauhri, A. Pandey, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*. https://doi.org/10.48550/arXiv.2407.21783 .

[9] Feng, Z., D. Guo, D. Tang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, 10.18653/v1/2020.findings-emnlp.139*.

[10] Green, C. 1969. Application of theorem proving to problem solving. In *Proc. of the 1st Intl. Joint Conf. on Artificial Intelligence, DOI: https://www.ijcai.org/Proceedings/69/Papers/023.pdf*, IJCAI'69, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[11] Gulwani, S., O. Polozov, R. Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages 4*(1-2): 1–119. https://doi.org/10.1561/2500000010 .

[12] Guo, D., S. Ren, S. Lu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*. https://doi.org/10.48550/arXiv.2009.08366 .

[13] Halaweh, M. 2023. Chatgpt in education: Strategies for responsible implementation. *Contemporary Educational Technology 15*(2). https://doi.org/10.30935/cedtech/13036 .

[14] Haruna-Cooper, L. and M.A. Rashid. 2023. Gpt-4: the future of artificial intelligence in medical school assessments. *Journal of the Royal Society of Medicine*: 01410768231181251. https://doi.org/10.1177/01410768231181251 .

[15] Iyer, S., I. Konstas, A. Cheung, et al. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, 10.18653/v1/D18-1192*. PMLR.

[16] Jiang, A.Q., A. Sablayrolles, A. Mensch, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*. https://doi.org/10.48550/arXiv.2310.06825 .

[17] Katz, D.M., M.J. Bommarito, S. Gao, et al. 2023. Gpt-4 passes the bar exam. *SSRN*. https://doi.org/10.2139/ssrn.4389233 .

[18] Kulal, S., P. Pasupat, K. Chandra, et al. 2019. Spoc: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems, 10.48550/arXiv.1906.04908*.

[19] Li, R., L.B. Allal, Y. Zi, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*. https://doi.org/10.48550/arXiv.2305.06161 .

[20] Liu, J., C.S. Xia, Y. Wang, et al. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems, 10.48550/arXiv.2305.01337* 36 .

[21] Lo, C.K. 2023. What is the impact of chatgpt on education? a rapid review of the literature. *Education Sciences 13*(4): 410. https://doi.org/10.3390/educsci13040410 .

[22] Lukasczyk, S. and G. Fraser 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, 10.1145/3510454.3516829*, pp. 168–172.

[23] Luo, Z., C. Xu, P. Zhao, et al. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*. https://doi.org/10.48550/arXiv.2306.08568 .

[24] Manna, Z. and R.J. Waldinger. 1971. Toward automatic program synthesis. https://doi.org/10.1145/362566.362568 .

[25] Mikolov, T., I. Sutskever, K. Chen, et al. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of NIPS,*

10.48550/arXiv.1310.4546.

[26] Nijkamp, E., J. Lee, H. Touvron, et al. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv:2203.13474*. https://doi.org/10.48550/arXiv.2203.13474 .

[27] OpenAI. 2024. Gpt-4 omni: A comprehensive multimodal model for language, vision, and beyond. *arXiv preprint arXiv:2408.01234*. https://doi.org/10.48550/arXiv.2408.01234 .

[28] Pennington, J., R. Socher, and C.D. Manning 2014. Glove: Global vectors for word representation. In *Proceedings of EMNLP, 10.3115/v1/D14-1162*.

[29] Peters, M.E., M. Neumann, M. Iyyer, et al. 2018. Deep contextualized word representations. In *Proceedings of ACL, 10.18653/v1/N18-1202*.

[30] Raihan, N., A. Anastasopoulos, and M. Zampieri 2025, April. mHumanEval - a multilingual benchmark to evaluate large language models for code generation. In L. Chiruzzo, A. Ritter, and L. Wang (Eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. Association for Computational Linguistics.

[31] Raihan, N., D. Goswami, S.S.C. Puspo, et al. 2024. Cseprompts: A benchmark of introductory computer science prompts. In *International Symposium on Methodologies for Intelligent Systems*.

[32] Raihan, N., C. Newman, and M. Zampieri 2024. Code llms: A taxonomy-based survey. In *2024 IEEE International Conference on Big Data (BigData), 10.1109/BigData62323.2024.10826108*.

[33] Raihan, N., J. Santos, and M. Zampieri. 2024. Mojobench: Language modeling and benchmarks for mojo. *arXiv preprint arXiv:2410.17736*. https://doi.org/10.48550/arXiv.2410.17736 .

[34] Raihan, N., M.L. Siddiq, J.C. Santos, et al. 2025. Large language models in computer science education: A systematic literature review. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1, 10.1145/3699459.3703350*.

[35] Rogers, A., O. Kovaleva, and A. Rumshisky. 2020. A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics* 8: 842–866. https://doi.org/10.1162/tacl_a_00349 .

[36] Roziere, B., J. Gehring, F. Gloeckle, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*. https://doi.org/10.48550/arXiv.2308.12950 .

[37] Savelka, J., A. Agarwal, C. Bogart, et al. 2023. Can generative pre-trained transformers (gpt) pass assessments in higher education programming courses? *arXiv preprint arXiv:2303.09325*. https://doi.org/10.48550/arXiv.2303.09325 .

[38] Siddiq, M.L., S.B. Dristi, J. Saha, et al. 2024. The fault in our stars: Quality assessment of code generation benchmarksn. In *24th IEEE International Conference on Source Code Analysis and Manipulation (SCAM), 10.1109/SCAM63248.2024.00018.*

[39] Sok, S. and K. Heng. 2023. Chatgpt for education and research: A review of benefits and risks. *Available at SSRN 4378735*. https://doi.org/10.2139/ssrn.4378735 .

[40] Surameery, N.M.S. and M.Y. Shakor. 2023. Use chat gpt to solve programming bugs. *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290 3*(01): 17–22. https://doi.org/10.55529/ijitc.31.17.22 .

[41] Svyatkovskiy, A., S.K. Zhao, S. Fu, et al. 2021. Fast and memory-efficient neural code completion. In *ICML, 10.1109/MSR52588.2021.00045.*

[42] Vaswani, A., N. Shazeer, N. Parmar, et al. 2017. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, and et al. (Eds.), *Advances in Neural Information Processing Systems, 10.48550/arXiv.1706.03762*, Volume 30. Curran Associates, Inc.

[43] Wang, B. and A. Komatsuzaki. 2021. Gpt-j-6b: A 6 billion parameter autoregressive language model. *arXiv, 10.5281/zenodo.5297110.*. https://doi.org/10.5281/zenodo.5297110. .

[44] Wang, X., Y. Wang, F. Mi, et al. 2021. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*. https://doi.org/10.48550/arXiv.2108.04556 .

[45] Wang, Y., W. Wang, S. Joty, et al. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 10.18653/v1/2021.emnlp-main.685.*

[46] Wei, Y., Z. Wang, J. Liu, et al. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*. https://doi.org/10.48550/arXiv.2312.02120 .

[47] Zan, X.V., M. Deng, D. Yang, et al. 2022. A survey of benchmarks for natural language to code generation. In *ACL, 10.18653/v1/2022.acl-long.412.*

[48] Zhang, S.J., S. Florin, Lee, and et al. 2023. Exploring the mit mathematics and eecs curriculum using large language models. *arXiv preprint arXiv:2306.08997*. https://doi.org/10.48550/arXiv.2306.08997 .

Appendix A Data Sources

Table A1: List of Coding Websites & MOOCs

Name	**Link**
CodingBat	https://codingbat.com/python
Learn Python	https://www.learnpython.org
Edabit	https://edabit.com/challenges/python3
Python Principles	https://pythonprinciples.com/challenges/
Hacker Rank	https://www.hackerrank.com/domains/python
Edx	https://www.edx.org
Coursera	https://www.coursera.org
CS50 (Harvard)	https://learning.edx.org/course/course-v1:HarvardX+CS50S+Scratch/home
PforE (UMich)	https://www.coursera.org/learn/python/home
CS1301xI (GT)	https://learning.edx.org/course/course-v1:GTx+CS1301xI+1T2023/home
CS1301xII (GT)	https://learning.edx.org/course/course-v1:GTx+CS1301xII+1T2023/home
CS1301xIII (GT)	https://learning.edx.org/course/course-v1:GTx+CS1301xIII+1T2023/home
CS1301xIV (GT)	https://learning.edx.org/course/course-v1:GTx+CS1301xIV+1T2023/home
Programming in Python (Meta)	https://www.coursera.org/learn/programming-in-python

Appendix B Sample Prompts

```
Prompt1.
    You are given the coefficients of a polynomial P.
    Your task is to find the value of P at point x.
Prompt2.
    You are given a square matrix A with dimensions N x N.
    Your task is to find the determinant.
Prompt3.
    ...
```

(a) Sample Prompts from the Coding Sites.

```
Prompt1.
    implement a program that prompts the user for the answer to the Great
    Question of Life, the Universe and Everything, outputting Yes if the
    user inputs 42 or (case-insensitively) forty-two or forty two.
    Otherwise output No.
Prompt2.
    implement a program that prompts the user for a greeting. If the
    greeting starts with "hello", output $0. If the greeting starts with
    an "h" (but not "hello"), output $20. Otherwise, output $100. Ignore
    any leading whitespace in the user's greeting, and treat the user's
    greeting case-insensitively.
Prompt3.
    ...
```

(b) Sample Prompts from the MOOCs.

```
Prompt1.
    def func(x): return x * 2
    print(func(3))
    What is the output?
    6 3 9 None
Prompt2.
    print(float(5))
    What will be the output?
    5 5.0 None Error
Prompt3.
    ...
```

(c) Sample MCQ Prompts.

Fig. B1: Sample prompts from `CSEPrompts 2.0`

Appendix C Inclusion and Exclusion Criteria

This appendix outlines the comprehensive criteria applied in selecting coding exercises and MCQ prompts for our dataset. These criteria were designed to ensure that the methodology remains transparent, replicable, and aligned with the objectives of introductory Python education.

C.1 Applied on Coding Prompts

Table C2: Inclusion and Exclusion Criteria for Coding Prompts

Criterion Type	Description and Technical Details	Example
Inclusion: Self-Containment	The exercise must be fully self-contained. All necessary context, parameters, and variable definitions must be provided within the prompt, ensuring no external references are needed.	A Python function prompt that defines all inputs in the description and provides sample input/output directly (e.g., "Implement a function that returns the square of a number; given input 3, output 9").
Inclusion: Clear Specifications	The problem must present an unambiguous statement with explicit definitions for expected inputs and outputs. This includes technical details such as data types, valid input ranges, and any operational constraints (e.g., expected time complexity).	A challenge stating: "Given a list of integers, return a new list with each element doubled. For input [1, 2, 3], output should be [2, 4, 6]."
Inclusion: Core Programming Constructs	The exercise should focus on basic programming elements such as loops, conditionals, functions, and elementary data structures (e.g., lists, dictionaries). It must be directly applicable to the curriculum of an introductory Python course.	A task that requires iterating over a list using a `for`-loop to compute the sum of its elements, illustrating basic control flow.
Exclusion: File I/O and Terminal Operations	Any exercise that involves file input/output or requires interaction with the command line/terminal is excluded. This is due to the inherent limitations in Language Learning Models (LLMs) for reliably handling such operations.	A problem that instructs: "Read data from `input.txt` and write the processed results to `output.txt`."
Exclusion: Ambiguous Evaluation Criteria	Exercises with vague or subjective instructions that could lead to inconsistent automated grading are omitted. Clear, technical grading metrics must be specified to ensure reproducibility in evaluation.	A prompt that simply states: "Optimize the code" without defining whether the focus is on reducing time complexity, memory usage, or another specific metric.
Exclusion: External Library Dependencies	Problems requiring libraries beyond the Python Standard Library are excluded. This ensures that exercises remain focused on core language features and avoids potential issues with library installation or versioning.	A challenge that requires the use of a third-party package, such as `numpy`, for performing operations that could otherwise be implemented with standard Python lists.

C.2 Applied on MCQ Prompts

Table C3: Inclusion and Exclusion Criteria for MCQ Prompts

Criterion Type	Description and Technical Details	Example
Inclusion: Clear Question Stem	The MCQ prompt must present a well-defined question stem that encapsulates a single, focused inquiry. It should provide all necessary context and use precise technical language to ensure that the question is unambiguous and self-contained.	A question such as: "Which of the following data types is immutable in Python?" where the stem clearly defines the concept under assessment.
Inclusion: Unambiguous Answer Choices	All answer options must be distinct and adhere to a consistent format. Each option should be clearly identifiable as either correct or incorrect, without overlapping concepts or technical inaccuracies.	Answer choices: "List", "Tuple", "Dictionary", "Set" with "Tuple" being the correct option.
Inclusion: Adequate Distractor Quality	Distractors should be plausible yet technically incorrect, designed to challenge students by testing their understanding of subtle differences between similar concepts. They must be crafted to avoid common misconceptions while remaining sufficiently challenging.	Distractors that include data structures with similar characteristics, e.g., comparing mutable versus immutable types.
Exclusion: Ambiguous Distractors	Any MCQ prompt where distractors are overly vague or too similar to the correct answer—thereby risking subjective interpretation—is excluded. This ensures a clear distinction between correct and incorrect responses.	A question with options such as "Tuple", "Immutable list", "Frozen list", where the phrasing may lead to confusion about Python's data types.
Exclusion: Non-Standard Terminology	MCQs that utilize informal language, colloquialisms, or non-standard technical terms are omitted. This exclusion maintains academic rigor and ensures consistency in terminology across prompts.	A prompt using "cool variable type" instead of the accepted term "immutable type".
Exclusion: Compound Questions	Questions that conflate multiple concepts or assessment objectives in a single prompt are excluded. Each MCQ should target a single, clearly defined concept to facilitate precise evaluation.	A compound question such as: "Which data type is immutable and supports indexing?" where combining two properties can obscure the primary learning objective.