CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution

Reviewed on OpenReview: `https://openreview.net/forum?id=XXXX`

Abstract

We present CRUXEval (**C**ode **R**easoning, **U**nderstanding, and e**X**ecution **Eval**uation), a benchmark consisting of 800 Python functions (3-13 lines). Each function comes with an input-output pair, leading to two natural tasks: input prediction and output prediction. First, we propose a general recipe for generating our execution benchmark by sampling from a model, which can be used for more challenging versions of the benchmark if needed. Second, we evaluate twenty code models on our benchmark and discover that many recent high-scoring models on HumanEval show no improvements on our benchmark. The best setup, GPT-4 with chain of thought (CoT), achieves a pass@1 of 75% and 81% on input and output prediction, respectively. In contrast, Code Llama 34B achieves a pass@1 of 50% and 46% on input and output prediction. When it comes to reasoning about code, GPT-4 has a huge edge over other models but still fails consistently on some surprisingly simple Python programs.

1 Introduction

In recent months, software engineering and programming have become increasingly mainstream domains for language models (LMs) as they attempt to conquer a potpourri of tasks including code completion, program repair, debugging, test case generation, and code optimization (see Zan et al. (2023) and Fan et al. (2023) for surveys). Recent models including Code Llama (Roziere et al., 2023), GPT-3.5 (Brown et al., 2020; Ouyang et al., 2022), and GPT-4 (OpenAI, 2023) have shown promise in code-related tasks and are being used to develop tools to help programmers.

The primary evaluation for code LMs is via benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), which test the ability to generate short code snippets from natural language specifications. While HumanEval and MBPP capture code generation abilities on simple and fundamental tasks, there is an absence of benchmarks capturing other fundamental dimensions of code LMs such as code understanding and execution.

Motivated by this, we contribute a new benchmark, CRUXEval (**C**ode **R**easoning, **U**nderstanding, and e**X**ecution **Eval**uation) with two tasks: 1) output prediction, CRUXEval-O to measure code execution following and 2) input prediction, CRUXEval-I to measure code reasoning and understanding. Two samples in CRUXEval are shown in Listing 1 (modified for readability). CRUXEval examines the abilities of code LMs to reason about the execution behaviour of *simple* Python programs. While LMs shouldn't be expected to replace an interpreter on arbitrarily complex problems, we ensure the samples in our benchmark are simple (maximum 13 lines, no complex arithmetic) and solvable by a university-level CS graduate without needing more memory than the output (in our opinion). CRUXEval

©2023 Author One and Author Two.

```python
def f(string):
    string_x = string.rstrip("a")
    string = string_x.rstrip("e")
    return string

# output prediction, CRUXEval-O
assert f("xxxxaaee") == ??
## GPT4: "xxxx", incorrect

# input prediction, CRUXEval-I
assert f(??) == "xxxxaa"
## GPT4: "xxxxaae", correct
```

```python
def f(nums):
    count = len(nums)
    for i in range(-count+1, 0):
        nums.append(nums[i])
    return nums
# output prediction, CRUXEval-O
assert f([2, 6, 1, 3, 1]) == ??
# GPT4: [2, 6, 1, 3, 1, 6, 1, 3, 1], incorrect

# input prediction, CRUXEval-I
assert f(??) == [2, 6, 1, 3, 1, 6, 3, 6, 6]
# GPT4: [2, 6, 1], incorrect
```

Listing 1: Sample problems

provides a useful and important probe for better understanding the capabilities of code LMs, as following a few simple steps of code execution should be a basic requirement for these models. The ability to reason about the execution behavior of code also paves the way to tackling more difficult tasks such as code repair with execution feedback and code summarization.

At a high level, our benchmark is constructed as follows. First, we use Code Llama 34B to generate a large set of functions and inputs. The outputs are generated by executing the functions on the inputs. Second, we filter the set so that our benchmark only consists of short problems with low computation and memory requirements, problems which a good human programmer should be able to do without extra memory in a minute or so. Third, we randomly select 800 samples passing the filter, ensuring the benchmark is both small enough to easily run but large enough to reliably see performance differences among various models. While it is difficult to come up with example where the strongest models like GPT-4 fail consistently, we observe that models actually fail often on random yet reasonable generated programs. We also highlight that as models improve, this generate-and-filter approach can be used to create future benchmarks that are more difficult and test different aspects of program execution.

The best model, GPT-4, achieves a pass@1 of 67% on CRUXEval-I and 63% on CRUXEval-O. In contrast, the best open-source models only achieve 47% on CRUXEval-I and 44% on CRUXEval-O, failing over half the time at simple execution prediction and code reasoning despite being trained on 100G of Python code and 1T of code data. We also observe that for base models, stronger HumanEval performance correlates with stronger CRUXEval performance. However, this trend breaks down for models distilled on GPT-4 like data such as WizardCoder, Phind, and Phi, which have impressively high HumanEval scores but no better than CodeLlama on CRUXEval.

We also observe that CoT is an effective technique for improving performance on CRUXEval, but is far from enough to ace it. Overall, our benchmark reveals that the gap between GPT-4 and open source models reflects GPT-4's stronger ability to reason about the behavior of code. As existing benchmarks like HumanEval and MBPP are insufficient for measuring code understanding and execution ability, capturing it through our benchmark is critical to make progress towards closing the gap between open models and GPT-4. Finally, we discover that despite its impressive abilities, GPT-4 consistently fails to understand the execution behavior of some surprisingly simple Python programs.

2 Related Work

Benchmarks for Evaluating Code LMs: There are various benchmarks serving to evaluate different aspects of these code LMs. We survey a handful here and refer readers to the survey (Zhang et al., 2023d) for more. HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) evaluate Python code generation on relatively simple functions. HumanEval+ (Liu et al., 2023b) augments HumanEval with better test cases after discovering many passing solutions are incorrect.

Leveraging Test Cases and Code Execution: Another line of work uses test cases and code execution information to improve code generation. Some examples include Speculyzer (Key et al., 2022), CodeT (Chen et al., 2022), CodeGen-Test (Zhong et al., 2022), Coder-Reviewer reranking (Zhang et al., 2023c), MBR-EXEC (Shi et al., 2022) TCoT (Tian and Chen, 2023), Algo (Zhang et al., 2023b), Pangu-Coder2 (Shen et al., 2023), LEVER Ni et al. (2023), and Self-Play (Haluptzok et al., 2022). The idea of these works is to generate many programs and many test cases and select which programs and test cases seem correct based on the execution results. using execution info.

Most relevant to our work, a handful of works examine and improve the execution ability of code LMs. Austin et al. (2021), Scratchpad (Nye et al., 2021), and CodeExecutor (Liu et al., 2023a) train code LMs on execution information. Moving forward, we believe our CRUXEval could serve as a useful reference point as more techniques are designed to improve code execution abilities.

3 Benchmark Construction

CRUXEval consists of 800 distinct functions, each with an input-output pair such that executing the function on the input deterministically produces the output. Using these functions and input-output pairs, we derive two benchmark tasks. In the *output prediction* task, the goal is to predict the output of executing the function on its associated input. In the *input prediction* task, the goal is to find any input such that executing the function on that input produces the output. For both tasks, we use an execution-based correctness metric. For input prediction, a generated input passes if `assert f(generated_input) == output` passes, and for output prediction, a generated output passes if `assert f(input) == generated_output` passes. A few statistics about the samples of CRUXEval can be found in Appendix B.3.

3.1 Generating Candidates

We use Code Llama 34B to generate all the candidate functions and inputs of CRUXEval. To do so, we prompt it with the name of a function in the Python standard library such as `str.zfill` and ask it to generate a Python function that makes use of the library function in addition to 5 test inputs. We provide two varying few-shot examples in our prompt for improved diversity of generations (see Appendix B.2 for more details). A sample prompt is shown in Listing 3.

We use 69 different functions from the standard library: 47 from the `str`, 11 from `dict`, and 11 from `list` (see Appendix B.1 for the full list of functions). Overall, we generate a total of 102000 functions (46% `str`, 27% `dict`, 27% `list`) and 489306 input-output pairs.

3.2 Filtering Candidates

Next, we filter the generated candidates to ensure that the samples in the dataset are reasonable and of high quality. In order to avoid forcing the model to perform tasks such as arithmetic calculation, we design filters so that the benchmark only consists of samples that are solveable by a human without extra memory.

Concretely, we filter based on the following criteria: 1) Compile time: all arguments of the function must be used in the function, length of code is between 75 and 300 characters, no syntax errors, proper assertion `assert f(input) == output`. 2) Runtime: no floating point operations, true division, exp, other integer operations must have at least one argument ≤ 3, string and list operations must have at least one argument with length ≤ 3, finish running in 2 seconds, no uncaught exceptions. 3) Best effort to remove other undesirable code: function cannot have any imports (such as os, sys), must be deterministic (no random, set ordering), and cannot have side effects such as `input`, `__builtins__`, etc.

3.3 Data size and measuring noise

The success of HumanEval (164 examples) shows that evaluation benchmarks can be small, and having faster/cheaper evaluation is an overlooked advantage. We first overgenerate a larger set (~ 1700) of samples and then pick a minimal data size that leads to sufficiently small noise.

On the overgenerated set, Code Llama 34B outperforms Code Llama 13B (as expected), and we would like to retain this property with high confidence in our smaller dataset. To do this, we took bootstrap samples of size N out of $\sim$1700 samples to measure the probability that the performance would be reversed, shown in Fig. 1. 800 examples are enough to test that Code Llama 34B > Code Llama 13B, Code Llama cot > Code Llama and as well as between Deepseek 33B > Code Llama 34B (output).

We measure two types of noise: 1) sampling which data points to include in the benchmark, and 2) sampling candidates from models for each data point (temperature > 0). Of these, 1) dominates 2). For 1) since model A does not always outperform model B on all data points even if model $A >$ model B in aggregate, the measured performance depends on which data points are included. We measure both noise types on each model individually and type 1) noise on pairs of models using the bootstrap. Fortunately, we do not see major differences in noise level between models and the most important factor is just the size of

the dataset. Type 1) noise is generally around 1.5% whereas type 2) noise is around 0.2% at $N = 800$. Type 1) noise usually becomes smaller on pairs of models due to correlation, yielding statistically significant results at the $\alpha = 0.025$ (one-sided) for many model pairs.

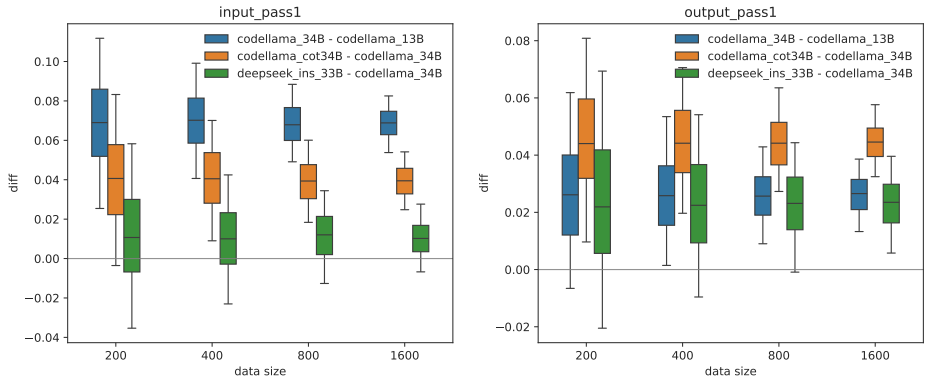
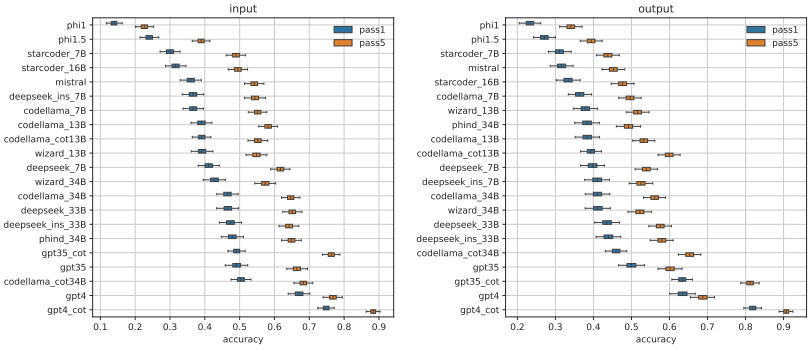


Figure 1: Difference between model pairs on bootstrap samples of various sizes. The whiskers show (2.5, 97.5) and boxes show (25, 75) percentiles.

4 Evaluation

We use $N = 100$ samples for all non-GPT models and $N = 10$ samples for GPT models. We report both pass@1 scores ($T = 0.2$) and pass@5 scores ($T = 0.8$). The results are shown in Fig. 2, and raw scores are provided in the Appendix in Table 2. In Fig. 2, we show the intervals generated by 10000 bootstrap samples from the dataset, where non-overlapping whiskers would be significant at the 2.5% level. To get more statistical power, we compare pairs of models on each bootstrapped sample. We show how each model compares to Code Llama 34B in Fig. 12. The intervals generally decreases due to correlations. On all models vs. Code Llama 34B, if the median bar clears the whisker in Fig. 2, then the difference actually holds with >97.5% probability under paired bootstrap. For example, Code Llama 34B is better than wizard_34B on input and Code Llama 34B is worse than deepseek_33B on output prediction with >97.5% probability.

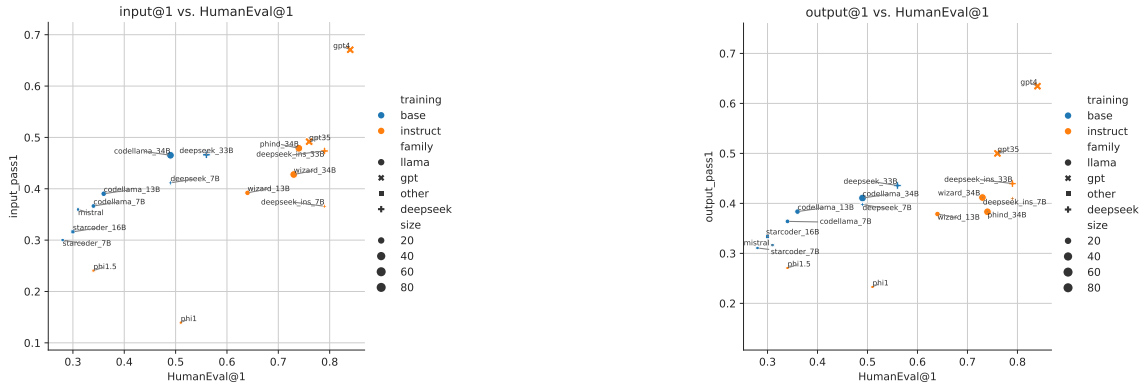


(a) CRUXEval-I Performance (b) CRUXEval-O Performance

Figure 2: Main Results. Boxes show (25, 75) percentiles, whiskers show (2.5, 97.5), and the middle bar shows the median (≈ mean).

5 Quantitative Analysis

Correlation between scores on HumanEval and CRUXEval: In Fig. 3, we plot reported HumanEval scores against scores on CRUXEval. When comparing the distilled models WizardCoder 34B and Phind 34B to Code Llama 34B, we see that the distilled models score over 20% more than Code Llama on HumanEval but *do not show this drastic improvement when evaluated on either input or output prediction*. Overall, this suggests that models optimized for the HumanEval task by distilling data from GPT-3.5 and GPT-4 (WizardCoder, Phind, Phi) may not have learned other code reasoning capabilities along the way.

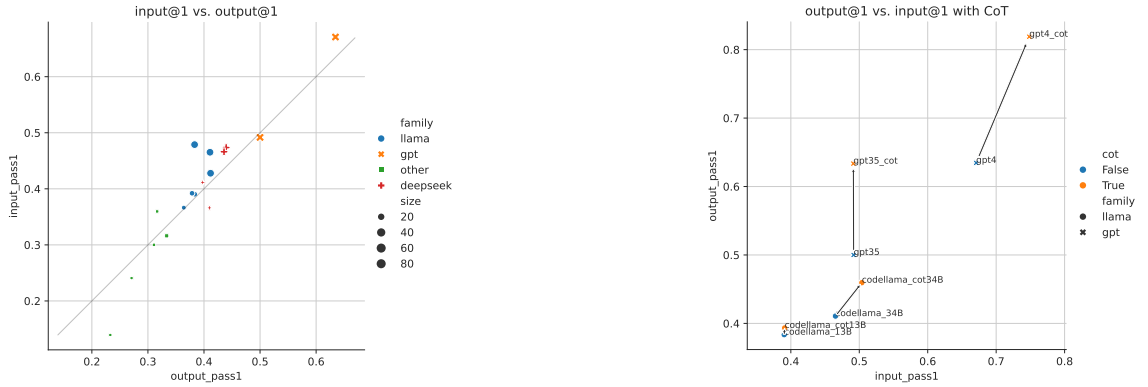


(a) CRUXEval-I vs. HumanEval

(b) CRUXEval-O vs. HumanEval

Figure 3: Correlation between HumanEval pass@1 scores and CRUXEval-O pass@1 scores

Relationship between input and output prediction: In Fig. 4 (left), we compare the input and output pass@1 scores with each other. Conceptually, the two tasks seem relatively different: output prediction is directly testing code execution ability, while input prediction requires a higher-level understanding of the code's functionality. However, we discover that there is a strong correlation between their performance, so performance on relatively distinct coding-related tasks may be closely correlated.



(a) Input vs. Output Prediction, Direct

(b) Output vs. Input Prediction, CoT

Figure 4: Correlation between Input and Output Prediction Scores, with and without CoT

5.1 Chain of Thought Prompting

Next, we evaluate how the popular chain-of-thought (CoT) prompting method (Wei et al., 2022) affects the performance of Code Llama, GPT-3.5, and GPT-4 models on CRUXEval. The full prompts can be found in Appendix E.3. All results are reported using $N = 10$ samples other than CodeLlama 13B and 34B without CoT, which are reported with $N = 100$ samples. As before, pass@1 is reported with $T = 0.2$ and pass@5 with $T = 0.8$. Additional results can be found in Appendix D.2.

Impact of CoT: In Fig. 4, we plot the input and output prediction scores of each model with and without CoT. First, GPT-4 benefits significantly more than other models. Second, output prediction boosts are generally larger than input prediction. In fact, CoT does not seem to improve Code Llama 13B and GPT-3.5 performance on input prediction. This is intuitive, as input prediction involves a more difficult reasoning task, while output prediction only requires executing the program step by step. We defer raw numbers to the Appendix in Table 3.

6 Qualitative Analysis

All models except GPT-4 have over 50% failure rate, showing they cannot do simple executions. In this section, we focus on GPT-4 with CoT and verify that the remaining 20% failures are due to the model, are consistent and are indeed on *simple* programs. We refer the reader to Appendix F for further examples of the failures highlighted in this section as well as some impressive successes.

Failures of GPT-4 CoT. GPT-4 CoT scored 0/10 on 54 output prediction tasks and 65 input prediction tasks. On 22 problems, it scored 0/10 on both input and output prediction tasks. We manually check the 22 problems if they pass our criteria of being *simple* problems. Most failures are indeed on simple programs, like the one below.

```
def f(string, sep):
    cnt = string.count(sep)
    return((string+sep) * cnt)[::-1]
assert f('caabcfcabfc', 'ab') == 'bacfbacfcbaacbacfbacfcbaac'
# GPT4+CoT preds:
# f('baa', 'c')
# 'bacfbacfcbaabcabfbacfcbaabc'
```

Listing 2 shows GPT-4 CoT failures for output prediction only. In Listing 2 (bot), the model fails because it concludes that 6173 is not less than 1000. Small perturbations like changing to `0 < num and num < 1000` or changing the strings also failed. Both problems only have 2 possible answers and other models sometimes get them correctly whereas GPT-4 CoT is consistently wrong. We manually tested scratchpad Nye et al. (2021) style prompts, which failed in the same way as regular CoT (Appendix F.3).

```
def f(text, suffix):
    if suffix == '':
        suffix = None
    return text.endswith(suffix)
assert f('uMeGndkGh', 'kG') == ??
# GPT-4 CoT: True, should be False
```

```
def f(num):
    if 0 < num < 1000 and num != 6174:
        return 'Half Life'
    return 'Not found'
assert f(6173) == ??
# GPT-4 CoT: 'Half Life', should be 'Not found'
```

Listing 2: GPT-4 CoT output failures

Failures of GPT-4, Input Prediction: Below are two simple failures on input prediction where GPT-4 failed for concise and simple Python programs even with CoT.

```
def f(text, repl):
    trans = str.maketrans(text.lower(), repl.lower())
    return text.translate(trans)
assert f(??) == 'lwwer case'
# GPT4 CoT: 'lower case', 'ow'
# could be 'lower case', 'lwwer case'
```

```
def f(text):
    string = ''
    for char in text:
        string += char + char.lower()
    return string
assert f(??) == 'llaallaakk'
# GPT-4 CoT: 'LAK'
# should be 'lalak'
```

7 Conclusion

We propose CRUXEval, a new benchmark consisting of simple Python functions to evaluate the input and output prediction abilities of code LMs. First, we propose a three-part recipe to distill our benchmark consisting of large-scale distillation, filtering, and data size selection via a statistical noise analysis (Sec. 3). Second, we conduct a qualitative analysis by evaluating 20 models on our benchmark (Sec. 4). Our analysis leads to insights regarding the correlation between HumanEval and our benchmark, the correlation between input and output prediction, differences between various code LMs, and the diversity of different models. Third, we provide a qualitative analysis of GPT-4's successes and failures (Sec. 6 and Appendix F). Overall, we believe that CRUXEval provides a complementary perspective to classical code LM evaluation such as HumanEval and MBPP and encourage creators of future code LMs to try out our benchmark!

References

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

L. Berglund, M. Tong, M. Kaufmann, M. Balesni, A. C. Stickland, T. Korbak, and O. Evans. The reversal curse: Llms trained on" a is b" fail to learn" b is a". *arXiv preprint arXiv:2309.12288*, 2023.

T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*, 2023.

P. Haluptzok, M. Bowers, and A. T. Kalai. Language models can teach themselves to program better. *arXiv preprint arXiv:2207.14502*, 2022.

D. Key, W.-D. Li, and K. Ellis. I speak, you verify: Toward trustworthy neural program synthesis. *arXiv preprint arXiv:2210.00848*, 2022.

C. Liu, S. Lu, W. Chen, D. Jiang, A. Svyatkovskiy, S. Fu, N. Sundaresan, and N. Duan. Code execution with pre-trained language models. *arXiv preprint arXiv:2305.05383*, 2023a.

J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023b.

P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023c.

A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

M. Mizrahi, G. Kaplan, D. Malkin, R. Dror, D. Shahaf, and G. Stanovsky. State of what art? a call for multi-prompt llm evaluation. *arXiv preprint arXiv:2401.00595*, 2023.

A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-t. Yih, S. Wang, and X. V. Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR, 2023.

M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama. Demystifying gpt self-repair for code generation. *arXiv preprint arXiv:2306.09896*, 2023.

R. OpenAI. Gpt-4 technical report. arxiv 2303.08774. *View in Article*, 2023.

L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

B. Peng, M. Galley, P. He, H. Cheng, Y. Xie, Y. Hu, Q. Huang, L. Liden, Z. Yu, W. Chen, and J. Gao. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*, 2023.

B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

B. Shen, J. Zhang, T. Chen, D. Zan, B. Geng, A. Fu, M. Zeng, A. Yu, J. Ji, J. Zhao, et al. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936*, 2023.

F. Shi, D. Fried, M. Ghazvininejad, L. Zettlemoyer, and S. I. Wang. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*, 2022.

Z. Tian and J. Chen. Test-case-driven programming understanding in large language models for better code generation. *arXiv preprint arXiv:2309.16120*, 2023.

G. Tyen, H. Mansoor, P. Chen, T. Mak, and V. Cărbune. Llms cannot find reasoning errors, but can correct them! *arXiv preprint arXiv:2311.08516*, 2023.

J. Uesato, N. Kushman, R. Kumar, F. Song, N. Siegel, L. Wang, A. Creswell, G. Irving, and I. Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. Lou. Large language models meet nl2code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, 2023.

K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787, Toronto, Canada, July 2023a. Association for Computational Linguistics.

K. Zhang, D. Wang, J. Xia, W. Y. Wang, and L. Li. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *arXiv preprint arXiv:2305.14591*, 2023b.

T. Zhang, T. Yu, T. Hashimoto, M. Lewis, W.-t. Yih, D. Fried, and S. Wang. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*, pages 41832–41846. PMLR, 2023c.

Z. Zhang, C. Chen, B. Liu, C. Liao, Z. Gong, H. Yu, J. Li, and R. Wang. A survey on language models for code. 2023d.

M. Zhong, G. Liu, H. Li, J. Kuang, J. Zeng, and M. Wang. Codegen-test: An automatic code generation model integrating program test information. *arXiv preprint arXiv:2202.07612*, 2022.

Appendix A. Limitations and Future Work

Correlations between various code tasks: While our benchmark serves as an interesting lens to analyze code LMs, one might object that output prediction can simply be done with a Python interpreter and that input prediction abilities can be greatly enhanced by equipping a LM with an interpreter, like in GPT-4 Code Interpreter mode. While this is true, we believe that a good code LM still ought to have good code understanding and execution capabilities, similar to that of a strong programmer. We see that base models have a reasonably strong correlation between HumanEval, input prediction, and output prediction score. An interesting future direction is to more deeply investigate the correlations between performance on various code-related tasks such as code completion, execution, bug-finding, and code summarization.

Distilling future execution benchmarks: Our benchmark only measures the input and output prediction accuracy of relatively simple and self-contained Python functions distilled from a single model (Code Llama 34B). It would also be interesting to measure these capabilities on longer and more difficult code snippets, open-domain code samples, or code in other programming languages. As our distillation technique is relatively general, we welcome others to create their own benchmarks measuring the execution of code snippets from other distributions.

Variation due to prompt and temperature: The accuracy of a model on our benchmark may be very sensitive to the prompt and task format (Mizrahi et al., 2023). We try our best to address this by using prompts that are similar as possible across models (see Appendix E.2 and E.3) but understand that some prompts may improve the performance of certain models while decrease the performance on others. There are also countless prompting techniques (see (Liu et al., 2023c) for a comprehensive survey) that can be tried to improve the performance. We also run all our experiments with $T = 0.2$ and $T = 0.8$ due to budget constraints, but different temperatures will lead to different performance for all models. One must always be cautious and critical when using benchmarks to compare models. For example, for input prediction, while Phind v2's 47.9% pass@1 may seem to beat CodeLlama's 46.5%, the standard deviations of both models with respect to the 800 samples selected turns out to be around 1.5%, so this conclusion cannot be made.

Information loss due to pass@1: While the average pass@k metric is common in the code generation literature, it compresses a large amount of information into one number. While we suggest reporting pass@1 and pass@5 for our benchmark, we comment that pass@k is only one perspective of measuring execution ability. We try to shed more light on behaviour by including a bit more analysis throughout this work, but encourage the development of different evaluation and analysis techniques.

Fine-tuning: In our first fine-tuning experiment, we only check for exact string match when decontaminating the fine-tuning set, so there may still be semantic duplication or similar programs with small modifications, which may lead to a higher performance than if those examples were removed. In this work, we only consider the most direct and straightforward fine-tuning scheme. We believe there is room for improvement via more sophisticated techniques, such as using process supervision (Uesato et al., 2022), fine-tuning on correct CoT generations, or fine-tuning on snippets of code while including the program state after each step. Seeing that models like Phi, WizardCoder, and Phind outperformed

Code Llama on HumanEval but not CRUXEval inspires the need for a deeper investigation of the utility of finetuning on distilled data from a more powerful model. Lastly, it remains a curiosity whether fine-tuning on execution information can help code generation abilities.

Jointly improving code generation and code execution: As we discovered, distilled models like Phi, Phind, and WizardCoder that are fine-tuned on code generation do not improve significantly on CRUXEval compared to their base models. It is unknown whether the opposite is true: does improved fine-tuning on code execution lead to better code generation abilities? It would also be interesting to explore techniques that can lead to improved performance on both code generation and code execution simultaneously.

Understanding reasoning from the lens of code: As future work, we believe that our benchmark serves as a good starting point towards understanding the code reasoning abilities of LM. Many further execution evaluations may be possible, such as testing execution of recursive functions, execution from a natural language description and an input, or execution of a composition of two functions. We find that output prediction serves as a good testbed for understanding CoT failures, because each step clearly corresponds to an operation with a ground truth, so reasoning failures can be pinpointed. We observed many examples of CoT failures due to simple mistakes that the model seems to have knowledge about (see Appendix F.3.2 for examples), and it should be possible to analyze and characterize this behaviour more systematically.

Self-repair: Lately, self-repair has been used to improve the reasoning and programming abilities of LLMs (Chen et al., 2023; Olausson et al., 2023; Madaan et al., 2023; Peng et al., 2023; Zhang et al., 2023a; Tyen et al., 2023). From our qualitative analysis, we find that when using CoT, many output prediction failures are recitation errors of information the model may already understand. Therefore, we believe that these mistakes may be easier to repair than when the correct reasoning path is not found in the first place, and that CRUXEval may be a simpler task to better understand model repair capabilities.

Appendix B. Benchmark Construction and Statistics

In this section, we provide more details about the process of constructing our benchmark. A sample prompt for generating functions and test cases is shown in Listing 3. The prompt is constructed by including two few-shot examples, one containing a specified `str` function and one containing a specified `list` function. The full list of specified functions is given in B.1, and the full list of few-shot examples chosen from is given in B.2. We learned that having random-looking inputs instead of common words and phrases in the few-shot prompts significantly increased the difficulty of the benchmark.

Listing 3: Sample prompt for generating functions and test cases

```
You will be given a function name between [TASK] and [/TASK] tags. Following the examples given, write
    ↪ a Python function that makes use of the given function and 5 test inputs for that function.

[TASK]
str.center
[/TASK]
[PYTHON]
def f(text):
    ls = list(text)
    for i in range(1, len(ls) - 1):
        ls.insert(i, '+')
    return ''.join(ls).center((len(ls) - 1) * 2)
[/PYTHON]
[TEST]
assert f('lynel') == ??
assert f('nzoh') == ??
assert f('u') == ??
assert f('anfsoixz') == ??
assert f('xzd') == ??
[/TEST]

[TASK]
list.append
[/TASK]
[PYTHON]
def f(nums):
    count = len(nums)
    for i in range(-count+1, 0):
        nums.append(nums[i])
    return nums
[/PYTHON]
[TEST]
assert f([2, 6, 1, 3, 1]) == ??
assert f([7, 1, 2, 6, 0, 2]) == ??
assert f([4, 3, 2, 1, 2, -1, 4, 2]) == ??
assert f([0, 6, 2, -1, -2]) == ??
assert f([-6, -2, 1, -3, 0, 1]) == ??
[/TEST]

[TASK]
str.zfill
[/TASK]
[PYTHON]
```

B.1 Functions used in prompt

For each of `str`, `list`, and `dict`, we use all the non-dunder methods under that class. The resulting list of methods is as follows:

- `str`: capitalize, casefold, center, count, encode, endswith, expandtabs, find, format, format_map, index, isalnum, isalpha, isascii, isdecimal, isdigit, isidentifier, islower, isnumeric, isprintable, isspace, istitle, isupper, join, ljust, lower, lstrip, maketrans, partition, removeprefix, removesuffix, replace, rfind, rindex, rjust, rpartition, rsplit, rstrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, zfill
- `list`: append, clear, copy, count, extend, index, insert, pop, remove, reverse, sort
- `dict`: clear, copy, fromkeys, get, items, keys, pop, popitem, setdefault, update, values

Motivated by seeing a GPT-4 failure of treating the `^` symbol as an exponential rather than an xor, we also attempted using all the non-dunder methods from `operator`. However, we found that the majority of the functions obtained were either too simple and uninteresting, or too computational, since many of the methods under `operator` are bit-manipulation or calculational operations. Therefore, we excluded it from our final benchmark.

B.2 Few-shot Examples

We use 10 handwritten few-shot examples, 5 using `str` functions and 5 using `list` functions. For each prompt, we include two few-shot examples, one string few-shot example and one list few-shot example, for a total of 25 different combinations of few-shot prompts. We generate programs and inputs using Code Llama 34B with temperature $T = 1$.

One interesting observation is that for a fixed pair of few-shot examples, there seems to be a limit to the number of diverse functions that can be generated: after about 60000 generations, only about 5000 of them were unique. Using all 25 combinations of few-shot prompts helps to overcome this duplication bottleneck.

The full set of few-shot examples can be found in Listing 5.

B.3 Dataset Statistics

In Fig. 5, we show the distribution of character count and line count of the 800 samples in our benchmark.

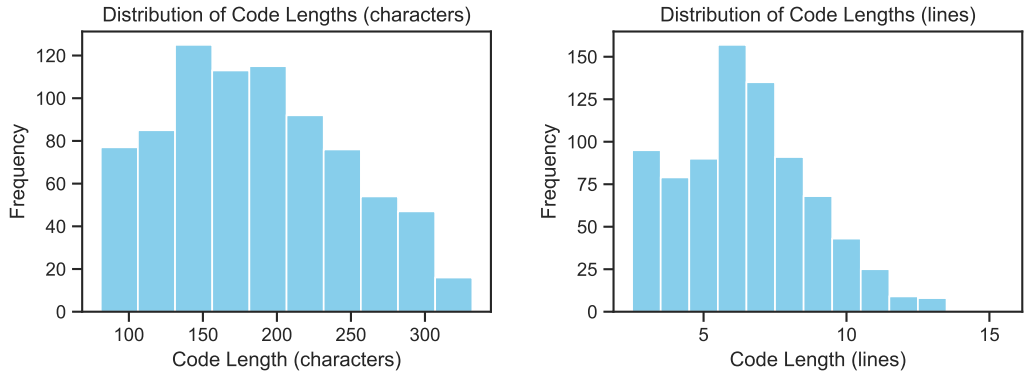


Figure 5: Dataset Distributions

In Fig. 6, we show the distribution of the "step count" of programs (with one outlier of 3175 steps excluded). Here, steps roughly correspond to Python bytecode operations during

execution. The precise definition can be understood by checking the "numsteps" variable in our code here.

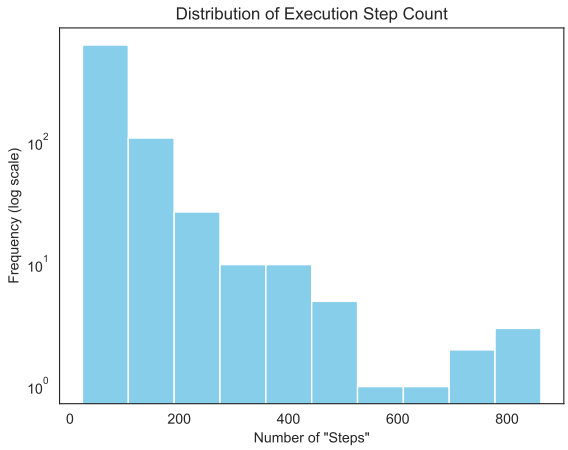
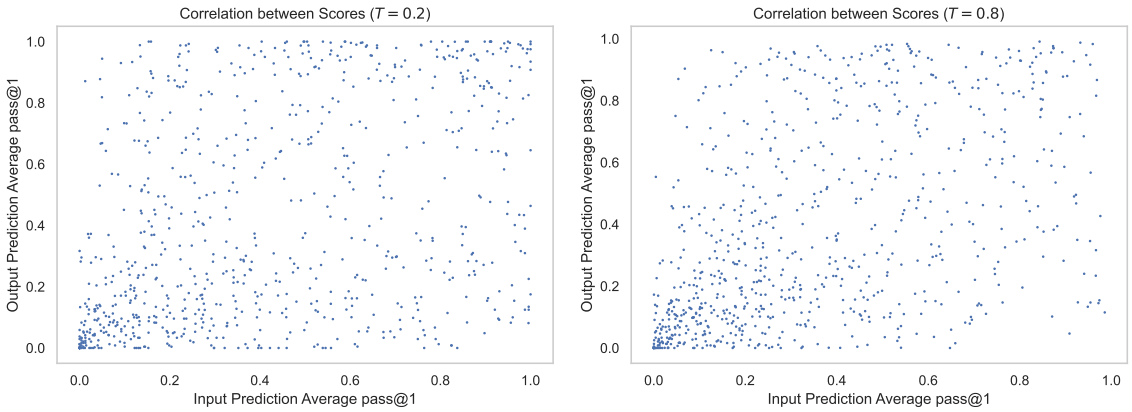


Figure 6: Number of steps

In Fig. 7, we plot the output prediction pass@1 scores against the input prediction pass@1 scores for each sample, observing little to no correlation between the difficulty of the two.



(a) $T = 0.2$ (b) $T = 0.8$

Figure 7: Sample-by-sample correlation between Input Prediction and Output Prediction

Method-level statistics: In Fig. 8, we show the number of samples containing each method in `str`, `list`, and `dict`. Even though we distilled the same number of samples using each `str` function and about twice as many for each `list` and `dict` functions, we observe that the resulting distribution is highly non-uniform. This is due to a few reasons. First, about 30% of the time, Code Llama 34B sometimes fails to follow the instruction of including the library method in the resulting function. Second, some functions naturally lead to more operations that are removed by the filter. Third, common functions such as `str/list.index` or `list.append` are used in methods they are not prompted in.

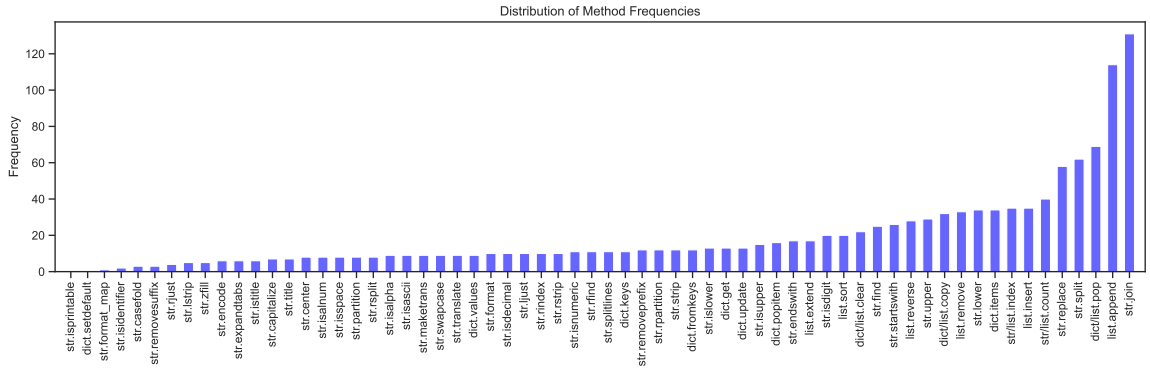
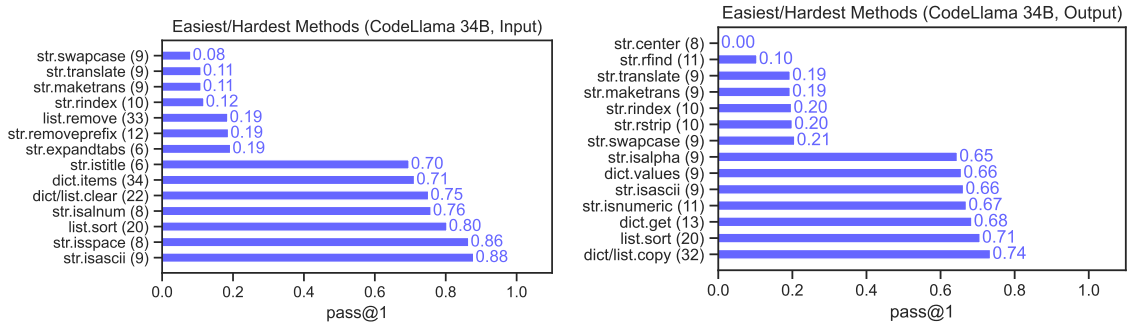


Figure 8: Frequency of various methods in CRUXEval

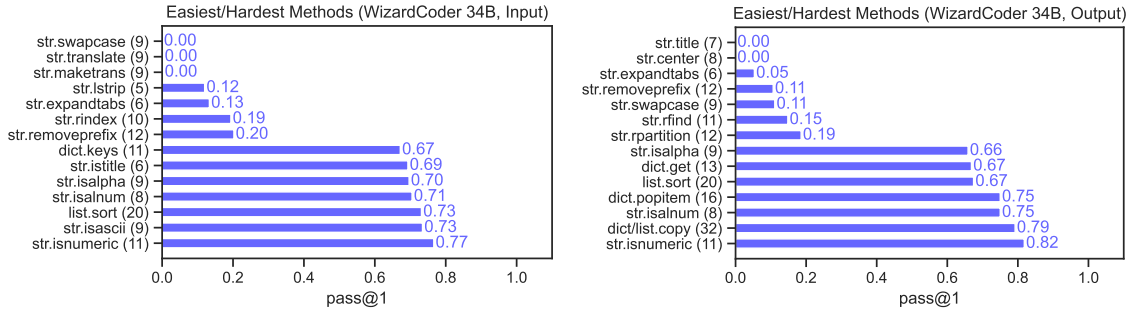
Next, we investigate trends of which methods are easier/harder for code LMs. For each method in the `list`, `str`, and `dict` libraries listed in Appendix B.1 with at least 5 samples, we calculate the average input prediction and output prediction score of benchmark samples containing that method. We show the 7 easiest and hardest methods for Code Llama 34B (Fig. 9), WizardCoder 34B (Fig. 10), and GPT-4 (Fig. 11). Some of the hardest methods, such as `str.rsplit, str.maketrans, str.rfind`, seem to be more obscure. We hypothesize that they may be underrepresented in pretraining corpora, explaining models' worse performance. While the distilled datasets of models like Phi, Phind, and WizardCoder are not yet available to the public, we speculate that they may include of fewer instances of these underrepresented functions and that distilling more obscure methods may help the model better learn their syntax and semantics.



(a) Input Prediction

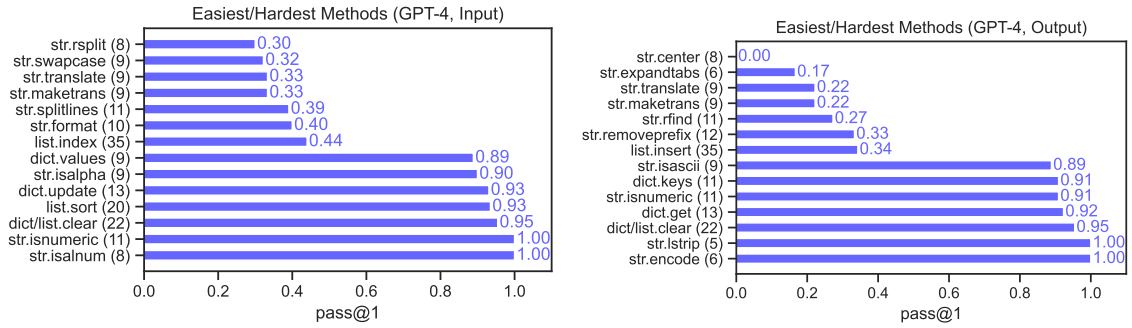
(b) Output Prediction

Figure 9: Easiest and hardest methods for Code Llama 34B input and output prediction, by pass@1 score ($T = 0.2$)



(a) Input Prediction (b) Output Prediction

Figure 10: Easiest and hardest methods for WizardCoder 34B input and output prediction, by pass@1 score ($T = 0.2$)



(a) Input Prediction (b) Output Prediction

Figure 11: Easiest and hardest methods for Code Llama 34B input and output prediction, by pass@1 score ($T = 0.2$)

Appendix C. Model URLs

For evaluation, we used the `gpt-3.5-turbo` and `gpt-4` models on October 26, 2023. Note that this is before the OpenAI developer day release of GPT-4-Turbo. The corresponding HuggingFace model URLs for the rest of the evaluated models are listed in Table 1.

Table 1: Models and HuggingFace URLs

Model Name	**HuggingFace URL**
Mistral (7B)	`https://huggingface.co/mistralai/Mistral-7B-v0.1`
Phi-1 (1.3B)	`https://huggingface.co/microsoft/phi-1`
Phi-1.5 (1.3B)	`https://huggingface.co/microsoft/phi-1_5`
DeepSeek Instruct (6.7B)	`https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct`
DeepSeek Instruct (33B)	`https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct`
DeepSeek Base (6.7B)	`https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-base`
DeepSeek Base (33B)	`https://huggingface.co/deepseek-ai/deepseek-coder-33b-base`
StarCoderBase (15.5B)	`https://huggingface.co/bigcode/starcoderbase`
StarCoderBase (7B)	`https://huggingface.co/bigcode/starcoderbase-7b`
WizardCoder (13B)	`https://huggingface.co/WizardLM/WizardCoder-Python-13B-V1.0`
WizardCoder (34B)	`https://huggingface.co/WizardLM/WizardCoder-Python-34B-V1.0`
Phind (34B)	`https://huggingface.co/Phind/Phind-CodeLlama-34B-v2`
CodeLlama 7B	`https://huggingface.co/codellama/CodeLlama-7b-hf`
CodeLlama (13B)	`https://huggingface.co/codellama/CodeLlama-13b-hf`
CodeLlama (34B)	`https://huggingface.co/codellama/CodeLlama-34b-hf`
CodeLlama Python (7B)	`https://huggingface.co/codellama/CodeLlama-7b-Python-hf`
CodeLlama Python (13B)	`https://huggingface.co/codellama/CodeLlama-13b-Python-hf`
CodeLlama Python (34B)	`https://huggingface.co/codellama/CodeLlama-34b-Python-hf`

Appendix D. Evaluation Results

D.1 Main Results

Table 2 shows the pass@1 and pass@5 results of all evaluated models on CRUXEval, and Fig. 12 shows them in box-plot form.

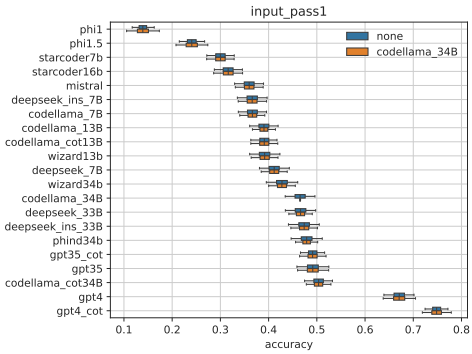
Table 2: Results of all models on CRUXEval

Model	Size	Input Prediction		Output Prediction	
		Pass@1	Pass@5	Pass@1	Pass@5
CodeLlama	7B	36.6%	55.2%	36.4%	49.6%
	13B	39.0%	58.2%	38.4%	53.2%
	34B	46.5%	64.7%	41.1%	56.1%
CodeLlama Python	7B	36.3%	56.0%	36.4%	49.7%
	13B	40.5%	58.0%	37.8%	50.8%
	34B	41.5%	59.2%	40.7%	53.7%
StarCoder-Base	7B	30.0%	48.9%	31.1%	43.8%
	15.5B	31.6%	49.5%	33.3%	47.7%
WizardCoder	13B	39.2%	54.8%	37.9%	51.6%
	34B	42.8%	57.3%	41.2%	52.2%
Phi-1	1.3B	13.9%	22.6%	23.3%	34.0%
Phi-1.5	1.3B	24.1%	38.9%	27.1%	39.4%
Phind v2	34B	47.9%	64.9%	38.3%	49.2%
Deepseek Coder-Base	6.7B	41.1%	61.7%	39.8%	53.9%
	33B	46.6%	65.1%	43.6%	57.6%
Deepseek Coder-Instruct	6.7B	36.6%	54.4%	41.0%	52.5%
	33B	47.4%	64.2%	44.0%	58.0%
Mistral	7B	36.0%	54.2%	31.7%	45.2%
GPT-3.5	-	49.2%	66.5%	50.0%	60.1%
GPT-4	-	67.1%	76.8%	63.4%	68.7%

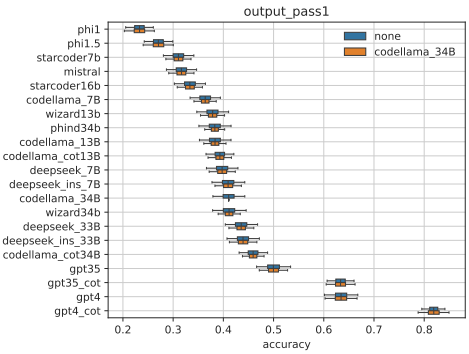
D.2 Additional Results on Impact of CoT

Table 3 shows the results of including CoT on Code Llama 13B, 34B, GPT-3.5, and GPT-4.

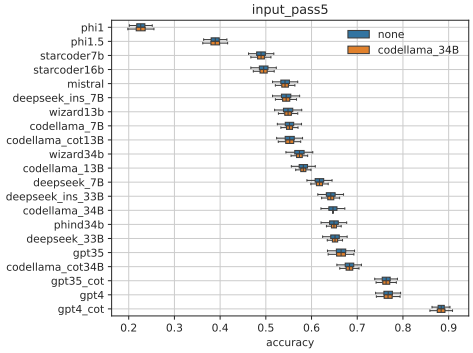
Sample-wide improvements from CoT: In Fig. 13, we show a histogram of how much CoT improves the pass@1 score of each sample (negative values means that CoT decreased the accuracy). We observe that CoT leads to little improvement for the majority of samples, this effect is partly due to samples already having high pass@1 scores. As evidenced by Fig. 13d, we see that CoT is much more effective for GPT-4 output prediction



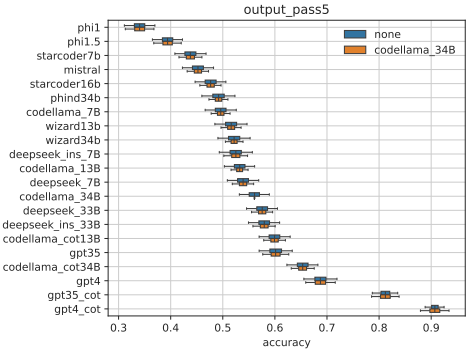
(a) Main Results, pass@1 (Input)



(b) Main Results, pass@1 (Output)



(c) Main Results, pass@5 (Input)



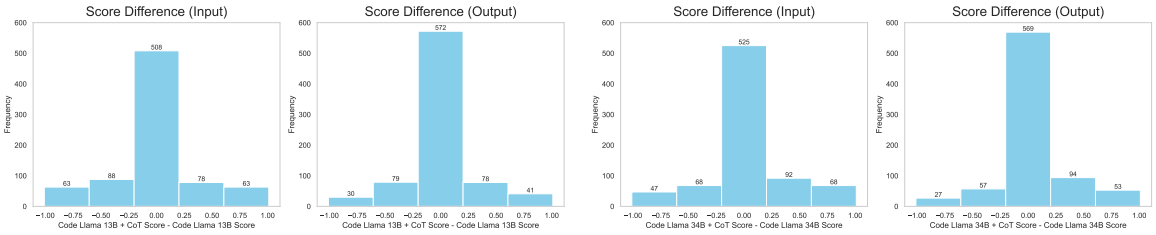
(d) Main Results, pass@5 (Output)

Figure 12: Main Results with confidence intervals compared to codellama_34B.

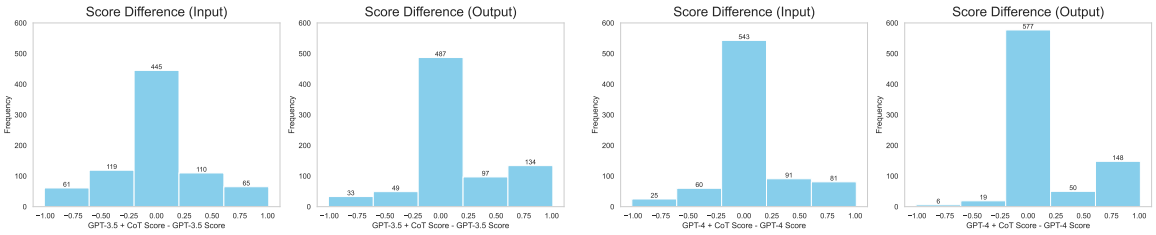
compared to both GPT-4 input prediction and other models. For the other models, however, we observe a large proportion of samples for which CoT actually decreases the pass@1.

Table 3: Impact of CoT on CRUXEVAL

Model	CoT	Input Prediction		Output Prediction	
		Pass@1	Pass@5	Pass@1	Pass@5
Code Llama 13B	✗	39.0%	58.2%	38.4%	53.2%
	✓	39.1%	55.2%	39.3%	59.9%
	-	**+0.1%**	**-3.0%**	**+0.9%**	**+6.7%**
Code Llama 34B	✗	46.5%	64.7%	41.1%	56.1%
	✓	50.4%	68.3%	46.0%	65.3%
	-	**+3.9%**	**+3.6%**	**+4.9%**	**+9.2%**
GPT-3.5	✗	49.2%	66.5%	50.0%	60.1%
	✓	49.1%	76.3%	63.3%	81.2%
	-	**-0.1%**	**+9.8%**	**+13.3%**	**+21.1%**
GPT-4	✗	67.1%	76.8%	63.4%	68.7%
	✓	74.8%	88.4%	81.9%	90.7%
	-	**+7.7%**	**+11.6%**	**+18.5%**	**+22.0%**



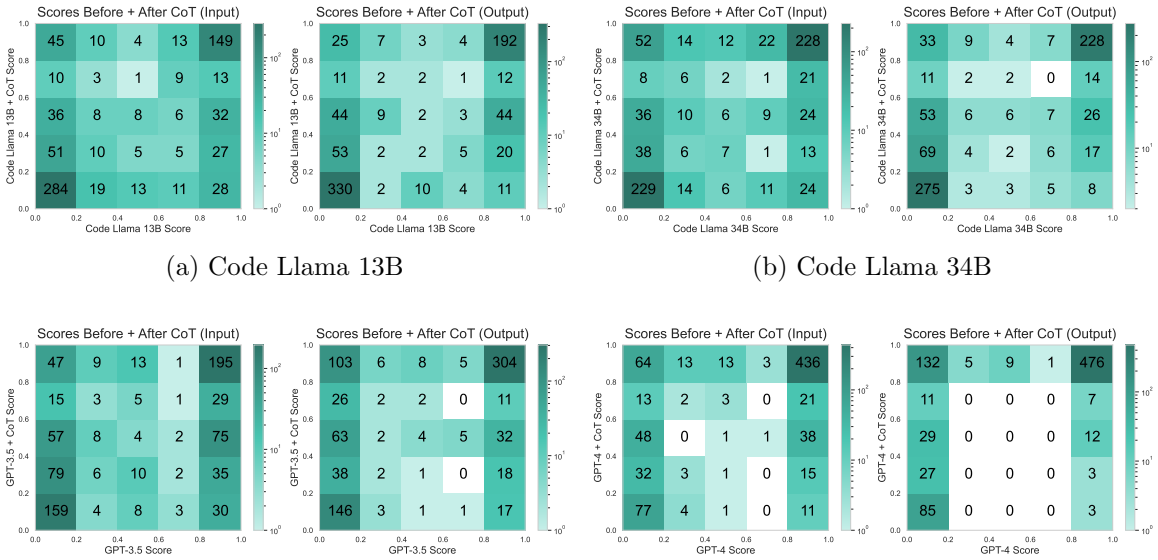
(a) Code Llama 13B (b) Code Llama 34B



(c) GPT-3.5 (d) GPT-4

Figure 13: Histogram of Score Differences between CoT and Original ($T = 0.2$)

In Fig. 14, we show a more granular perspective of Fig. **??**, which again highlights that CoT often decreases the pass@1 score of many samples. Again, we observe a stark difference between the impact of CoT on GPT-4 and other models.



(a) Code Llama 13B

(b) Code Llama 34B

(c) GPT-3.5

(d) GPT-4

Figure 14: Confusion Matrix of Direct Prediction vs. CoT Prediction ($T = 0.2$), Granular Version

Qualitative example of CoT harming performance: Finally, we show one example of input prediction and one example of output prediction where GPT-4 succeeds without CoT and fails with CoT.

```
# Output Prediction
def f(phone_number):
    while phone_number.find('77777') != -1:
        phone_number = phone_number.replace('77777', 'seg', 1)
    return phone_number
assert f('7747777722') == '774seg22'

# GPT-4 CoT says that '77777' is not in '7747777722', returning '7747777722'
```

```
# Input Prediction
def f(mylist):
    revl = mylist[:]
    revl.reverse()
    mylist.sort(reverse=True)
    return mylist == revl
assert f([5, 8]) == True

# GPT-4 CoT correctly says that "we need to provide a list that remains the same when sorted in
    ↪ descending order and when reversed," but then says the list should already be sorted in
    ↪ descending order, returning f([5, 4, 3, 2, 1]).
```

Correlations between failures of different models: Fig. 15 shows $P(Y \mid X = 0)/P(Y)$, the accuracy of model Y given that model X fails completely relative to the original accuracy of model Y. Although what is hard for a better model tend to be hard for worse models on average, worse models succeeded on some examples where the better models fail completely, showing idiosyncrasies in failures even for the best GPT-4 CoT model.

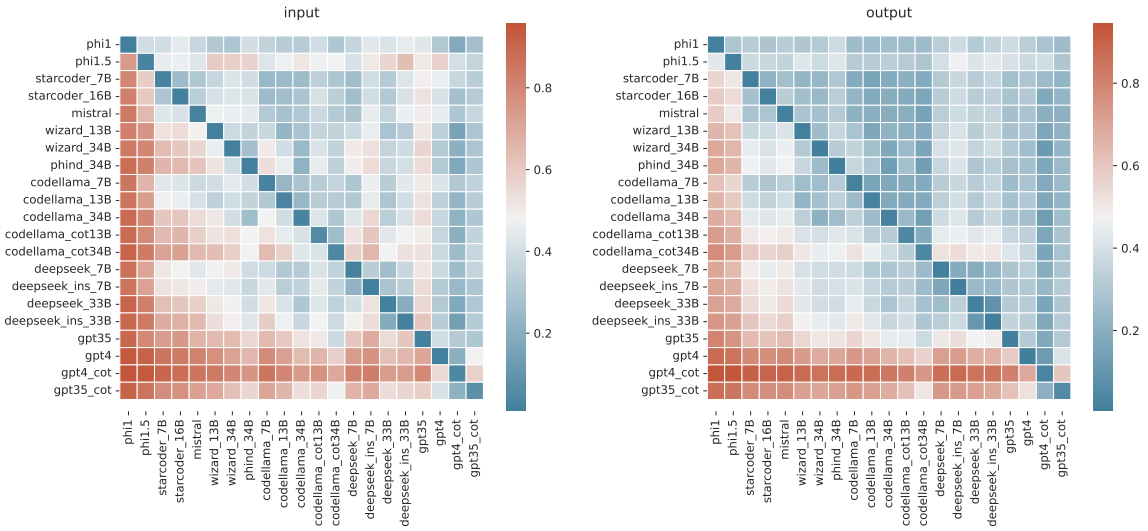
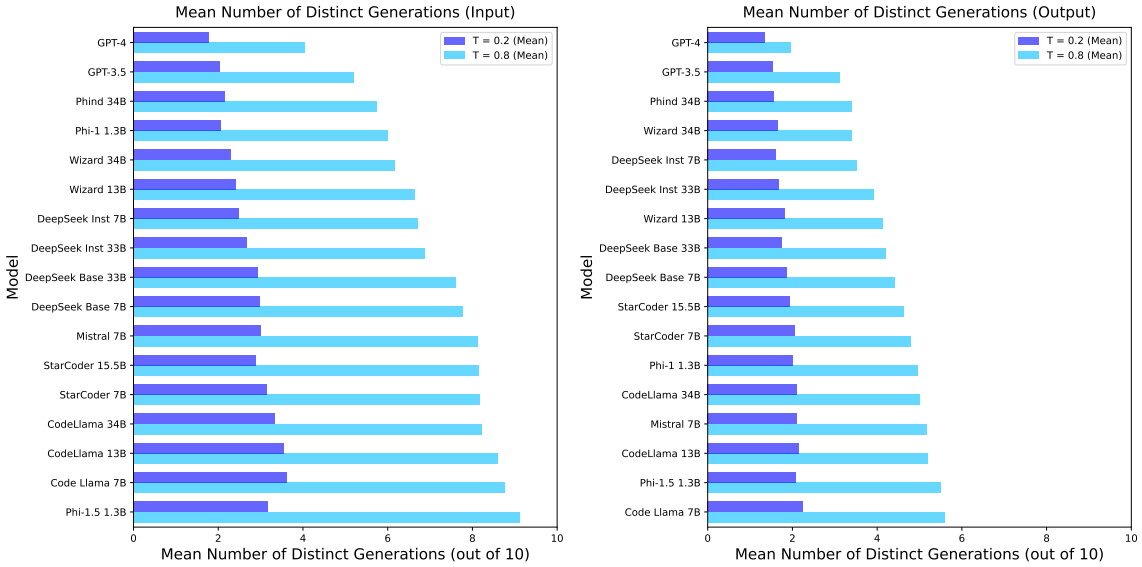


Figure 15: $P(Y \mid X = 0)/P(Y)$ where each Y is the accuracy of models in each row (X for column).

D.3 Results on Diversity of Generations

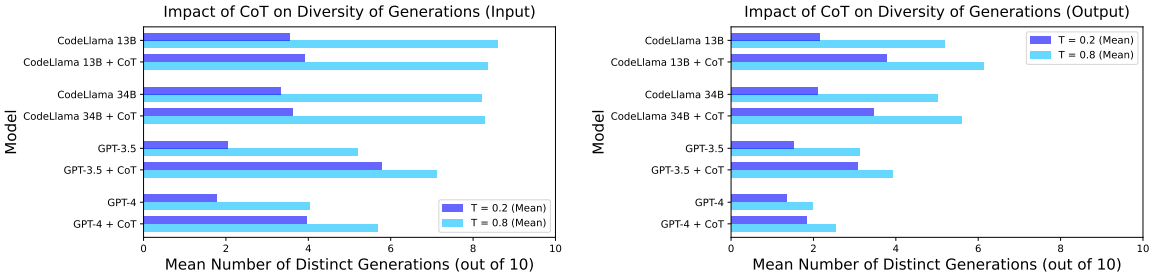
Diversity of generations across models: Next, we analyze the diversity of generated inputs and outputs across various models (without regard to correctness). In Fig. 16, we plot the mean and median number of unique answers generated across samples of CRUXEval for a selection of evaluated models. First, we observe the drastic increase in diversity between using $T = 0.2$ and $T = 0.8$. Second, by comparing Fig. 16a with Fig. 16b, we note that input prediction generally has a larger diversity of generations than output prediction. This may be due to the fact that output prediction only has one correct answer, but input prediction may have multiple correct answers. Third, we observe that at the same temperatures, Code Llama models have the highest diversity, while distilled models like Phind and WizardCoder have a lower diversity.



(a) Input prediction (b) Output prediction

Figure 16: Number of distinct generations of various models (out of 10) at $T = 0.2$ and $T = 0.8$

CoT increase the diversity of generations: In Fig. 17, for each model, we plot the average number of distinct generations across all samples, where different chains of thought with the same input or output prediction are considered identical. We see again the trend of input prediction generations being more diverse than output prediction generations. Interestingly, we observe that using CoT increases the diversity at both temperatures.



(a) Input prediction (b) Output prediction

Figure 17: Number of distinct generations of various models (normalized to be out of 10) at $T = 0.2$ and $T = 0.8$ with and without CoT. We observe that CoT increases the diversity of generations.

Functional diversity via distribution of pass@1 scores: In Fig. 18, for each model, we plot the percentage of samples where the pass@1 score ($T = 0.2$) is between 0.1 and 0.9, exclusive, indicating that the sample is neither "too easy" nor "too hard" for the model. This is a measure of functional diversity because models with more diversity are likely to generate both correct and incorrect samples, leading to more intermediate pass@1 scores. We make a

few observations relatively consistent with our prior observations. First, the percentages are relatively low across the board, indicating that at a temperature of $T = 0.2$, models are generally producing a majority of correct or a majority of incorrect outputs. Second, distilled models have a much lower functional diversity than base models, for example comparing Phind 34B to CodeLlama 34B or DeepSeek Instruct 33B to DeepSeek Base 33B. Third, CoT greatly increases the functional diversity of models, which is very evident when looking at GPT-3.5 and GPT-4.

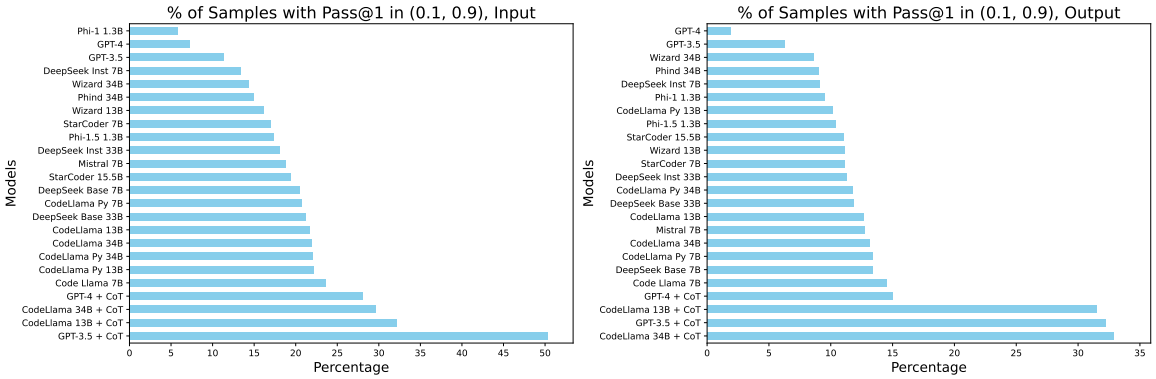


Figure 18: Percentage of samples where pass@1 score is in $(0.1, 0.9)$, exclusive.

D.4 Difficulty of Benchmark Samples

Distribution of sample difficulties: In Fig. 19, we show the average pass@1 score across all models for $T = 0.2$ in order to get a sense of the difficulty distribution of CRUXEval.

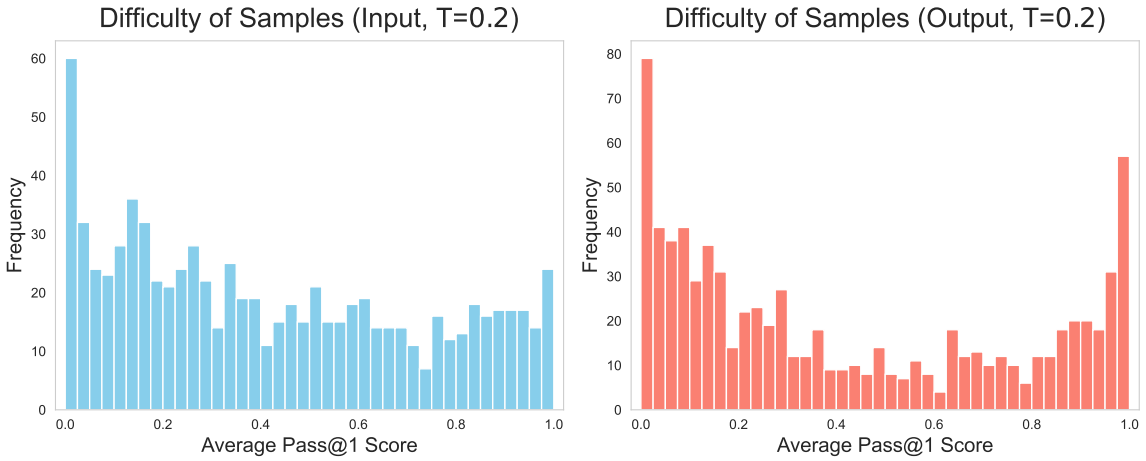


Figure 19: Difficulty of all samples of our benchmark, averaged across all models ($T = 0.2$)

In Fig. 20, we show the pass@1 distributions of a few of the best-performing models at $T = 0.8$. Compared to the overall distribution, the distribution appears to be more bimodal. The output prediction distribution is more bimodal than the input prediction distribution, perhaps reflecting the differences in the tasks themselves. We also see the familiar trend of

CoT increasing the number of samples with intermediate scores (in this case between 0.25 and 0.75).

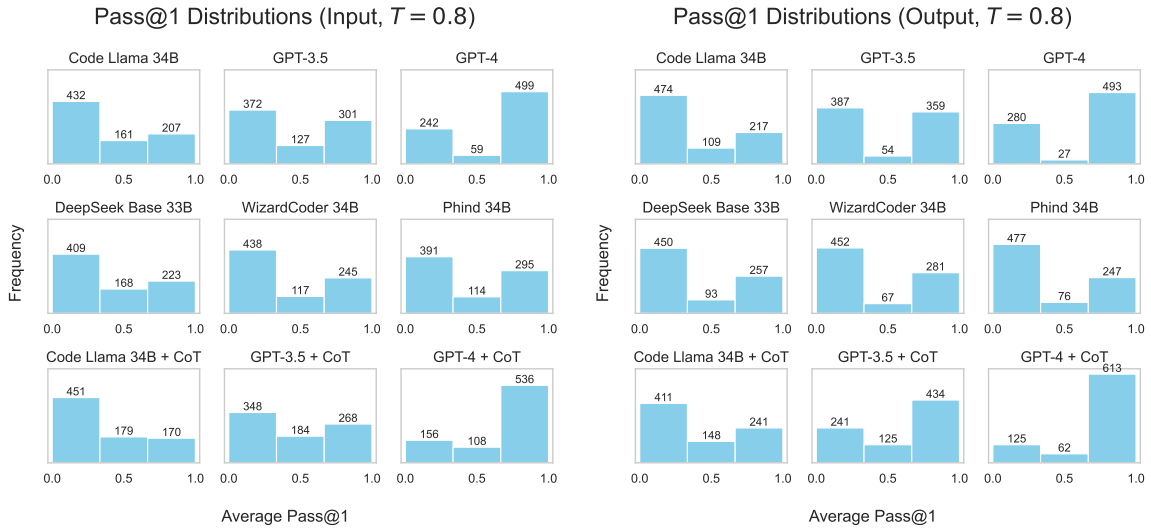


Figure 20: Pass@1 Distributions of Selected Models

Fully solved and unsolved samples: In Figs. 21 and 22, we examine a different metric, the percentage of examples with pass@1 score equal to 0 and 1, respectively, at $T = 0.8$. In a sense, this metric captures the ability of models to solve problems. It is also related to diversity, as with a higher diversity, the likelihood of solving the problem may increase. A few observations arise from looking at this metric.

Fig. 21, shows the percentage of samples that are completely unsolved by each model, i.e. with 0 pass@1. We analyze this metric for $T = 0.8$, because it leads to more diversity, which would improve this metric. First, when considering non-CoT modes, while GPT-3.5 and GPT-4 (red) are the two best-performing models at pass@1, they perform considerably worse at this metric than models such as Code Llama 34B and DeepSeek Base 33B. Second, instruction-tuned and distilled models (DeepSeek Instruct, Phind, WizardCoder) perform worse than their base counterparts, suggesting that their diversity may have been stifled from adherence to their instruction tuning datasets. Third, we observe that for the two Code Llama models, CoT actually makes this metric worse, but for GPT models, CoT makes it better. For GPT models, we hypothesize that this may be due to the increased diversity of CoT.

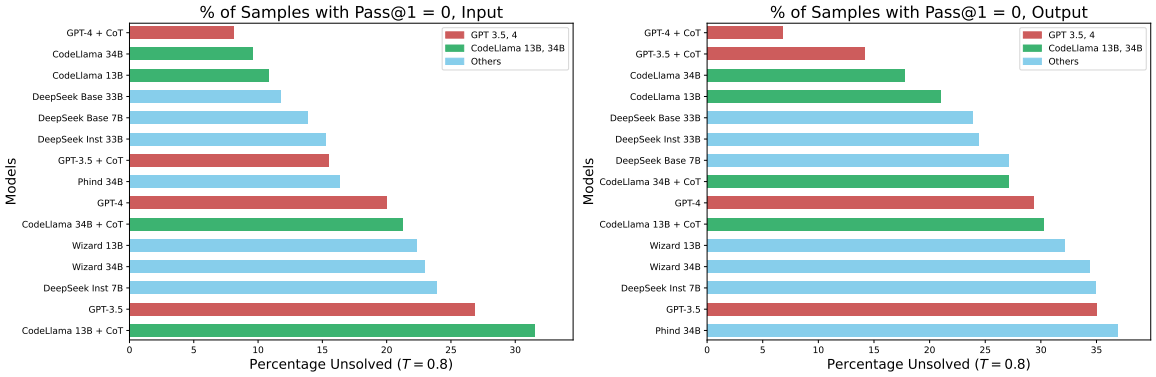


Figure 21: Percentage of samples unsolved, where pass@1 is 0 ($T = 0.8$)

In contrast, Fig. 22 shows the percentage of samples that models get fully correct, i.e. with a perfect pass@1. We analyze this metric for $T = 0.2$, as it would lead to more consistency, improving this metric. First, we see that GPT-4 excels, achieving over 60% for both input and output prediction. Second, when comparing base models with instruction tuned models, we see a trend matching the one before: since instruction tuned models are more consistent, they score better on this metric. Third, for output prediction, even though GPT-4 + CoT generally increases diversity, we see that consistency is not sacrificed!

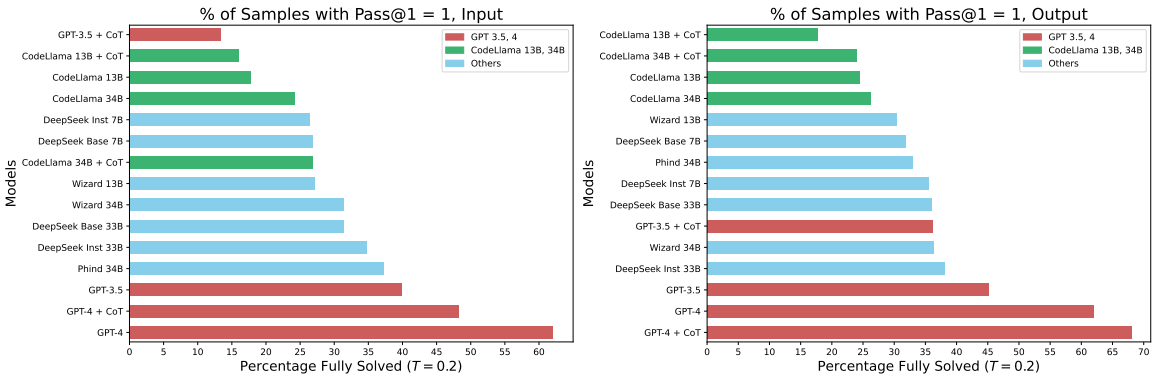


Figure 22: Percentage of samples fully solved, where pass@1 score is 1 ($T = 0.2$)

D.5 Impact of Anonymizing Functions

As a small ablation to understand the effect of variable names on execution ability, we also test CodeLlama 7B, 13B, and 34B on an anonymized version of a subset of the benchmark, where variable names are replaced with `x1, x2, ...` identifiers. An example of an anonymized function is shown in Listing 4. We use the same few-shot prompt without anonymization and report both pass@1 ($T = 0.2$) and pass@5 ($T = 0.8$) results on the anonymized benchmark with $N = 10$ samples. The results are shown in Table 4. This strengthens the case against memorization affects.

Listing 4: Sample of benchmark and anonymized version

```
Original:
```

```
def f(s):
    nums = ''.join(filter(lambda c:c.isdecimal(), s))
    if nums == '': return 'none'
    m = max([int(num) for num in nums.split(',')])
    return str(m)
assert f('01,001') == '1001'

Anonymized:
def f(x0):
    x1 = ''.join(filter(lambda x2: x2.isdecimal(), x0))
    if x1 == '':
        return 'none'
    x3 = max([int(x4) for x4 in x1.split(',')])
    return str(x3)
assert f('01,001') == '1001'
```

Table 4: Impact of Anonymization on CRUXEval

Model	Anonymized	Input Prediction		Output Prediction	
		Pass@1	Pass@5	Pass@1	Pass@5
CodeLlama 7B	✗	36.6%	48.0%	36.4%	43.5%
	✓	37.5%	53.3%	34.0%	46.9%
	Δ	**+0.9%**	**+5.3%**	**-2.4%**	**+3.4%**
CodeLlama 13B	✗	39.0%	50.2%	38.3%	44.7%
	✓	40.0%	55.8%	36.1%	50.6%
	Δ	**+1.0%**	**+5.6%**	**-2.2%**	**+5.9%**
CodeLlama 34B	✗	46.5%	57.4%	41.1%	47.5%
	✓	48.0%	63.8%	39.1%	54.0%
	Δ	**+1.5%**	**+6.4%**	**-2.0%**	**+6.5%**

D.6 Impact of Data-Generating Model

In the early phases of this work, we were concerned that using Code Llama 34B to generate the benchmark would give the model an unfair advantage. Therefore, we checked the performance of a few models when generating data with Code Llama 13B, GPT-3.5, and GPT-4. The results are shown in Table 5.

These samples were generated using a different prompt and a much more relaxed filter, so the raw scores differ from those in the main text. Across all datasets, we see that the relative ordering of Code Llama 13B, Code Llama 34B, and GPT-3.5 are preserved. We also observed that generating data with GPT-3.5 led to a significantly easier benchmark. After looking at a few samples manually, we believe this is because the resulting inputs are much more predictable and guessable, such as `f("abcde")` rather than `f("mai2!")`. Including few-shot examples with random inputs did not improve this issue, and we believe this is an artifact of instruction tuning. We believe that together with the anonymization results in Appendix D.5, these results provide some evidence that evaluating a model on its own generated data does not seem to provide it a significant advantage.

Table 5: Impact of Data Generating Model

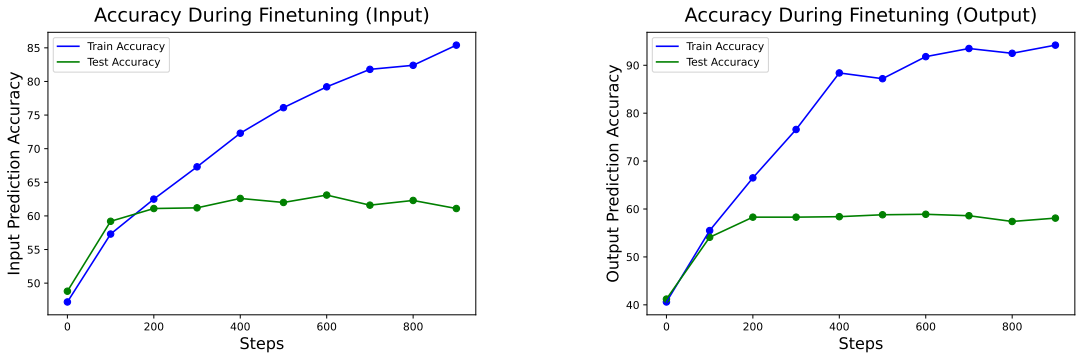
Data Model	Evaluation Model	Input Pass@1	Output Pass@1
CL 13B	CL 13B	28.1%	28.4%
CL 13B	CL 34B	33.8%	29.2%
CL 34B	CL 13B	25.1%	24.3%
CL 34B	CL 34B	29.9%	25.4%
CL 34B	GPT-3.5	40.5%	36.6%
GPT-3.5	CL 13B	42.3%	49.7%
GPT-3.5	CL 34B	52.1%	50.7%
GPT-3.5	GPT-3.5	67.1%	67.2%
GPT-4	CL 13B	28.1%	42.4%
GPT-4	CL 34B	37.0%	44.6%

D.7 Fine-tuning

We discover three interesting insights from fine-tuning. In the main text, we only discuss insight 3. As a refresher, we fine-tuned Code Llama 34B on 138889 samples of Python functions distilled with the procedure outlined in Sec. 3, without filtering. For the output prediction task, the model was fine-tuned on assertions of the form `assert f(input) == output`, and for the input prediction task, the model was fine-tuned on assertions of the form `assert output == f(input)`. During evaluation time, the fine-tuned model was asked to complete assertions of the same format as given in fine-tuning.

1. Direct fine-tuning leads to modest performance improvements: In the first setup, we analyze a stronger decontamination setup than that in the main text. Specifically, we remove samples that match functions used in the benchmark, even if the input-output pairs are different. In Fig. 23, we show the train and test accuracy of the model during the finetuning process. For ease of evaluation, the train accuracy is reported on a random subset of 500 samples from the finetuning set. The reported test accuracy is on a superset of CRUXEval.

First, we observe that fine-tuning is able to significantly increase performance on both input and output prediction tasks. Second, we observe that while the training accuracy is steadily increasing and the model is able to overfit the training set, the testing accuracy plateaus relatively quickly. This suggesting that simple fine-tuning may not be enough to achieve near-perfect scores on CRUXEval. Third, we observe that it is easier to overfit the training set on the output prediction benchmark than on the input prediction benchmark. We hypothesize this may be partially due to the fact that `assert output == f(input)` is a less natural format for assertions and partially due to the fact that input prediction requires a more sophisticated level of reasoning compared to output prediction.

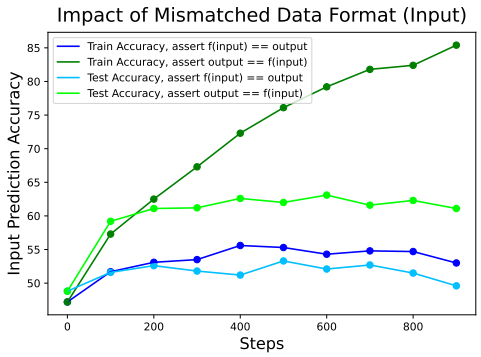


(a) Input prediction

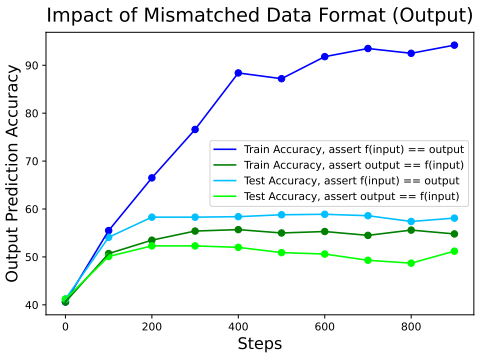
(b) Output prediction

Figure 23: Train accuracy (500 random samples) and test accuracy (superset of CRUXEval) while finetuning. For both tasks, there is improvement, and the model steadily fits the training set while plateauing on the testing set.

2. The format of fine-tuning data greatly impacts its effectiveness: We also discovered that it is important that the finetuning assertions be formatted in the same way as when evaluating the model at test time. As evidence of this, we fine-tune Code Llama 34B with two different sets of assertions, one on `assert output == f(input)` assertions and the other on `assert f(input) == output` assertions. We compare the accuracy of the two finetuned models on both input and output prediction in Fig. 24. We observe that when the format of the fine-tuning data and the testing data are different, the model even has difficulty overfitting the training set, showing that it may not have fully learned the equivalence of the two formats and the meaning of the `==` operator. This is perhaps another example of the "reversal curse" of LLMs (Berglund et al., 2023). The corresponding testing accuracy also plateaued at a lower accuracy when the format was misaligned. For example, in Fig. 24a, comparing the light green line with the light blue line shows almost a 10% difference in testing accuracy for input prediction when trained on a misaligned format. That being said, fine-tuning still improved performance relative to the base model, even with a mismatched format, showing that the fine-tuning with a mismatched format did still instill some information into the model.



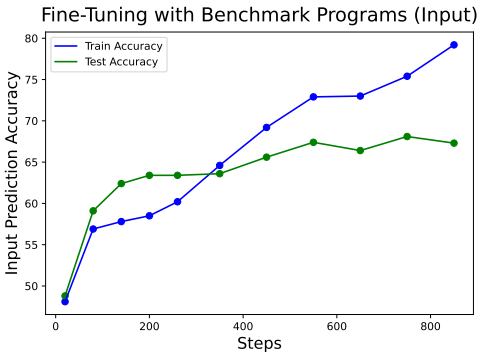
(a) Input prediction



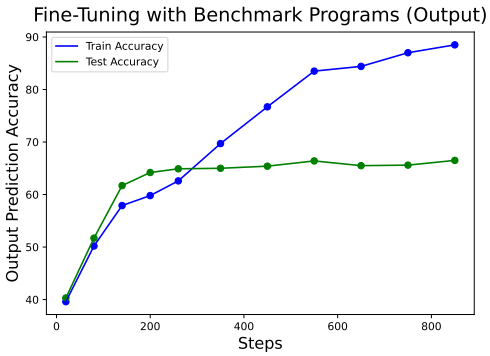
(b) Output prediction

Figure 24: Aligning the fine-tuning data format with the evaluation data format is very important for benchmark performance.

3. Including benchmark programs still cannot improve test accuracy beyond 70%: Finally, we explore the upper limits of fine-tuning on functions and assertions via a "cheating" setup. We curate a small set of *7259 samples consisting only of programs in the benchmark* but with different input-output pairs. We finetune on a mixture of 50% of the original finetuning set and 50% of this new set, showing the training and testing accuracy over time in Fig. 25. Despite finetuning on programs very similar to the benchmark, we still observe a plateauing effect in the test accuracy, suggesting that our execution tasks may be too difficult to learn from this simple fine-tuning scheme. Therefore, we suggest a few more fine-tuning ideas for improving our benchmark in Sec. A.



(a) Input prediction



(b) Output prediction

Figure 25: Finetuning 50% on the original finetuning set and 50% on "cheating" data

Appendix E. Prompts

In this section, we list all the prompts we use throughout the paper. Other than ensuring that generations could be parsed properly, all prompts were not optimized towards any particular models.

E.1 Benchmark Generation Few-Shot Prompts

Listing 5: All few-shot examples used for benchmark generation

```
string_1 = """[TASK]
str.split
[/TASK]
[PYTHON]
def f(text):
    words = text.split()
    result = []
    for i in range(len(words)):
        if i % 2 == 0:
            result.append(words[i][::-1])
        else:
            result.append(words[i].upper())
    return ' '.join(result)
[/PYTHON]
[TEST]
assert f("am7 fiDfd n") == ??
assert f("bnasadl") == ??
assert f("a j c n x X k") == ??
assert f("98 bask2 asoijdf9") = ??
assert f("") == ??
[/TEST]"""

string_2 = """[TASK]
str.capitalize
[/TASK]
[PYTHON]
def f(text):
    a = []
    words = text.split(' ')
    for i in range(len(words)):
        if words[i][0].isdigit():
            return 'no'
        if i%2 == 0:
            a.append(words[i].capitalize())
        else:
            a.append(words[i])
    return ' '.join(a)
[/PYTHON]
[TEST]
assert f("20xk flkawhf") == ??
assert f("lkw hj sfaibw fi 9") == ??
assert f("abbot 2929 mbpu") == ??
assert f("rotor zisxrs fh29nx") == ??
assert f("pxk 5 bxD 9") == ??
[/TEST]"""

string_3 = """[TASK]
str.rindex
[/TASK]
[PYTHON]
def f(text, char):
    index = text.rindex(char)
```

```
    result = list(text)
    while index > 0:
        result[index] = result[index-1]
        result[index-1] = char
        index -= 2
    return ''.join(result)
[/PYTHON]
[TEST]
assert f('mnjs krupa', 'u') == ??
assert f('kqwomn0xj', 'm') == ??
assert f('qpfi jzm', 'j') == ??
assert f('102x0zoq', '0') == ??
assert f('nzu  ei,', 'e') == ??
[/TEST]"""

string_4 = """[TASK]
str.rpartition
[/TASK]
[PYTHON]
def f(text, char):
    if char in text:
        pref, char, suff = text.rpartition(char)
        suff = suff[:-len(char)] + char + suff[len(char):]
        return suff + pref
    return text
[/PYTHON]
[TEST]
assert f('smswfwe-r', '-') == ??
assert f(',wpzpppdl/', 'p') == ??
assert f('9284701', '2') == ??
assert f('nvizoh2ja', 'c') == ??
assert f('aaa0a1', 'a') == ??
[/TEST]"""

string_5 = """[TASK]
str.center
[/TASK]
[PYTHON]
def f(text):
    ls = list(text)
    for i in range(1, len(ls) - 1):
        ls.insert(i, '+')
    return ''.join(ls).center((len(ls) - 1) * 2)
[/PYTHON]
[TEST]
assert f('lynel') == ??
assert f('nzoh') == ??
assert f('u') == ??
assert f('anfsoixz') == ??
assert f('xzd') == ??
[/TEST]"""

list_1 = """[TASK]
list.pop
[/TASK]
[PYTHON]
def f(names, num):
    queue = names
    while len(queue) > 1:
        for _ in range(num):
            queue.append(queue.pop(0))
        queue.pop(0)
    return queue.pop()
[/PYTHON]
```

```
[TEST]
assert f(['aiwn', 'xke', 'mpwiy'], 2) == ??
assert f(['y', 'z', 'cc', '2', '5', '.', 'zksdfjn'], 7) == ??
assert f(['98bfaj', 'cn11', 'fakldj', 'tjasl', 'a'], 10) == ??
assert f(['aghbvm'], 1) == ??
assert f(['mnv', 'fjw', 'fnk'], 0) == ??
[/TEST]"""

list_2 = """[TASK]
list.insert
[/TASK]
[PYTHON]
def f(text, position, value):
    length = len(text)
    index = position % (length + 1)
    if position < 0 or index < 0:
        index = length // 2
    new_text = list(text)
    new_text.insert(index, value)
    return ''.join(new_text)
[/PYTHON]
[TEST]
assert f('h grateful k', 3, 'h') == ??
assert f('umjwi', -5, 'm') == ??
assert f('coscifysu', 0, 'd') == ??
assert f('fnmart', 4, 'o') == ??
assert f('rzti', -1, 'a') == ??
[/TEST]"""

list_3 = """[TASK]
list.remove
[/TASK]
[PYTHON]
def f(array, elem):
    array.reverse()
    try:
        while elem in array:
            array.remove(elem)
    finally:
        array.reverse()
    return array
[/PYTHON]
[TEST]
assert f([-1, 2, 1, -8, 2], 2) == ??
assert f([], 2) == ??
assert f([1], 1) == ??
assert f([3, 6, 4, -2, 5], 4) == ??
assert f([3, 2, 1, 2, 7, 1], 1) == ??
[/TEST]"""

list_4 = """[TASK]
list.append
[/TASK]
[PYTHON]
def f(nums):
    count = len(nums)
    for i in range(-count+1, 0):
        nums.append(nums[i])
    return nums
[/PYTHON]
[TEST]
assert f([2, 6, 1, 3, 1]) == ??
assert f([7, 1, 2, 6, 0, 2]) == ??
assert f([4, 3, 2, 1, 2, -1, 4, 2]) == ??
```

```
assert f([0, 6, 2, -1, -2]) == ??
assert f([-6, -2, 1, -3, 0, 1]) == ??
[/TEST]"""

list_5 = """[TASK]
list.index
[/TASK]
[PYTHON]
def f(nums, swap1, swap2):
    i1 = nums.index(swap1)
    i2 = nums.index(swap2)
    nums[i1], nums[i2], nums[i1 + 1], nums[i2 + 1] = nums[i2], nums[i1], nums[i2 + 1], nums[i1 + 1]
    return nums
[/PYTHON]
[TEST]
assert f([6, 2, 1, 3, 4, 5], 3, 4) == ??
assert f([1, 1, 5, 3, 1, 2], 1, 2) == ??
assert f([1, 2, 1, 4, 1], 4, 2) == ??
assert f([6, 2, 3, 1, 7, 5, 7], 3, 7) == ??
assert f([2, 8, 8, 3, 8, 3, 9], 3, 2) == ??
[/TEST]"""
```

E.2 Direct Prediction Prompts

In Listings 6, 7, 8, 9, and 10, we include the prompts we use for our evaluation. We use a few-shot prompt for all models other than GPT models. For many models, we observed that using the zero-shot prompt leads to a generation that are not in a easily parsable format, and including the few-shot examples led to predictable formatting. For fairness, we also measured the performance of several few-shot prompts on the GPT models for a randomly sampled subset of the benchmark (instead of the full benchmark for cost reasons). However, we observed a decrease in performance compared to the zero-shot prompts for both input prediction and output prediction. Therefore, we decided to use the zero-shot prompt for GPT models and report numbers using that prompt. In addition, we use a separate output prediction prompt for Phind because the prompt in Listing 8 often led to explanation text before completing the assert.

Listing 6: Input Prediction (non-GPT)

```
You will be given a function f and an output in the form f(??) == output. Find any input such that
     ↪ executing f on the input leads to the given output. There may be multiple answers, but you
     ↪ should only output one. Think step by step before arriving at an answer. Finally, surround the
     ↪  answer, with no additional words, with [ANSWER] and [/ANSWER] tags. Express your answer as a
     ↪ function call that when executed will give the output.

[PYTHON]
def f(my_list):
    count = 0
    for i in my_list:
        if len(i) % 2 == 0:
            count += 1
    return count
assert f(??) == 3
[/PYTHON]
[ANSWER]
f(["mq", "px", "zy"])
[/ANSWER]

[PYTHON]
```

```
def f(s1, s2):
    return s1 + s2
assert f(??) == "banana"
[/PYTHON]
[ANSWER]
f("ba", "nana")
[/ANSWER]

[PYTHON]
{function}
assert f(??) == {output}
[/PYTHON]
[ANSWER]
```

Listing 7: Input Prediction (GPT)

```
You will be given a function f and an output in the form output == f(??). Output the completion of the
    ↪ last line so that the code will run without errors by finding any input such that executing f
    ↪ on the input leads to the given output. There may be multiple answers, and you can output any
    ↪ one. Do NOT output any additional information.

{function}
assert {output} == f(
```

Listing 8: Output Prediction (non-GPT, non-Phind)

```
Based on the given Python code, which may contain errors, complete the assert statement with the
    ↪ output when executing the code on the given test case. Do NOT output any extra information,
    ↪ even if the function is incorrect or incomplete. Do NOT output a description for the assert.

def f(n):
    return n
assert f(17) == 17

{function}
assert f({input}) ==
```

Listing 9: Output Prediction (GPT)

```
Based on the given Python code, which may contain errors, complete the assert statement with the
    ↪ output when executing the code on the given test case. Do not output any extra information,
    ↪ even if the function is incorrect or incomplete.

{function}
assert f({input}) ==
```

Listing 10: Output Prediction (Phind)

```
Based on the given Python code, which may contain errors, complete the assert statement with the
    ↪ output when executing the code on the given test case. Do NOT output any extra information,
    ↪ even if the function is incorrect or incomplete. Output "# done" after the assertion.

def f(n):
    return n
assert f(17) == 17 # done

{function}
assert f({input}) ==
```

E.3 Chain of Thought Prompts

Below, we include the prompts we use for the chain of thought experiments. For the same reasons mentioned in Appendix E.2, use a one-shot prompt for Code Llama models and a zero-shot prompt for GPT models.

Listing 11: CoT input prediction prompt (Code Llama)

```
You will be given a function f and an output in the form f(??) == output. Your task is to find any
    ↪ input such that executing f on the input leads to the given output. There may be multiple
    ↪ answers, but only output one. First, think step by step. Then, surround the answer with [
    ↪ ANSWER] and [/ANSWER] tags. Express your answer as a function call that when executed will
    ↪ give the output.

def f(x):
    return x + 1
assert f(??) == 17

To find an input such that executing f on the input leads to the given output, we can work backwards
    ↪ from the given assertion. We know that f(??) == 17.

Since the function f(x) returns x + 1, for f(??) to be equal to 17, the value of ?? should be 16.

Therefore, the function call that will give the output as 17 is:
[ANSWER]f(16)[/ANSWER]

{function}
assert f(??) == {output}
```

Listing 12: CoT input prediction prompt (GPT)

```
You will be given a function f and an output in the form f(??) == output. Your task is to find any
    ↪ input such that executing f on the input leads to the given output. There may be multiple
    ↪ answers, but only output one. First, think step by step. Then, surround the answer with [
    ↪ ANSWER] and [/ANSWER] tags. Express your answer as a function call that when executed will
    ↪ give the output.

{function}
assert f(??) == {output}
```

Listing 13: CoT output prediction prompt (Code Llama)

```
You are given a function and an input. Complete the assertion with the output of executing the
    ↪ function on the input. First, reason step by step before arriving at an answer. Then, surround
    ↪  the answer as an assertion with [ANSWER] and [/ANSWER] tags.

def f(s):
    return s + "a"
assert f("hi") == ??

The function f takes a string s as input and returns the concatenation of s with the string "a".

To determine the output of executing the function f on the input "hi", we need to concatenate "hi"
    ↪ with "a".

Therefore, the output of executing the function f on the input "hi" is "hia".

[ANSWER]assert f("hi") == "hia"[/ANSWER]

{function}
assert f(input) == ??
```

Listing 14: CoT output prediction prompt (GPT)

```
What should the output of this code be so that the assertion is correct? Reason step by step before
    ↪ arriving at an answer. Finally, surround the answer, with no additional words, with [ANSWER]
    ↪ and [/ANSWER] tags.

{function}
```

Appendix F. Qualitative Analysis

In this section, we see some examples of interesting successes and failures of the best performing model, GPT-4, with and without CoT. GPT-4 is relatively sensitive to its prompt, and slight tweaks in the prompt may lead correct examples to fail or incorrect examples to succeed. However, we believe that these examples are nevertheless interesting and reveal insights into the operating modes of GPT-4. Note that some of these examples may not be in the benchmark and were taken from a larger set of generated examples.

F.1 Output Prediction without CoT

F.1.1 GPT-4 Successes without CoT, Output Prediction

Even without CoT, we found that GPT-4 achieves impressively high pass@1 scores on output prediction. We highlight a few GPT-4 successes below that we found impressive, suggesting that GPT-4 has the capability to perform somewhat complex reasoning and code execution.

```
def f(text):
    if ',' in text:
        before, _, after = text.partition(',')
        return after + ' ' + before
    return ',' + text.partition(' ')[-1] + ' 0'
assert f('244, 105, -90') == ' 105, -90 244'

# GPT-3.5 output: '-90 244'
# CodeLlama 34B output: '244, 105, -90 0'
```

```
def f(text):
    text = text.lower()
    count = 0
    for char in text:
        if char.isalpha():
            count += 1
    return count
assert f("The computer factory") == 18

# GPT-3.5 output: 3
# CodeLlama 34B output: 16
```

```
def f(text):
    d = {}
    updated = []
    for ch in text:
        if ch in d:
            d[ch] += 1
        else:
            d[ch] = 1
    while len(d) != 0:
        el = d.popitem()
        for i in range(el[1]):
            updated.append(el[0])
    return ''.join(updated)
assert f('pdrq') == 'qrdp'

# GPT-3.5 output: 'pdrq'
# CodeLlama 34B output: 'qprd'
```

```
def f(a, b):
```

```
    b.reverse()
    c = b.copy()
    b.extend(a.copy())
    b.extend(c)
    return b
assert f([5, 2, 3], [4, 9, 3, 1]) == [1, 3, 9, 4, 5, 2, 3, 1, 3, 9, 4]

# GPT-3.5 output: [1, 3, 9, 4, 5, 2, 3]
# CodeLlama 34B output: [4, 9, 3, 1, 5, 2, 3, 4, 9, 3, 1]
```

```
def f(s):
    ret = ';'.join(sorted([c for c in s if c.isalnum()]))
    return ret
assert f('%*^8938a(6^' * 3) == '3;3;3;6;6;6;8;8;8;8;8;8;9;9;9;a;a;a'

# GPT-3.5 and CodeLlama 34B both do not terminate after 500 tokens
```

```
def f(nums, a, b):
    new_nums = []
    for n in nums:
         if n < a or n > b:
             new_nums.append(n)
    new_nums.sort()
    new_nums.extend(nums)
    return new_nums
assert f([25, 44, 24, 22, 38, 5, 35, 15], 20, 44) == [5, 15, 25, 44, 24, 22, 38, 5, 35, 15]

# GPT-3.5 output: [5, 15, 22, 24, 25, 35, 38, 44, 25, 44, 24, 22, 38, 5, 35, 15]
# CodeLlama 34B output: [5, 15, 22, 24, 25, 35, 38, 44, 25, 44, 24, 22, 38, 5, 35, 15]
```

F.1.2 GPT-4 Failures without CoT, Output Prediction

We still find a set of relatively simple failures on output prediction, which we expect would be relatively simple without CoT.

```
def f(nums):
    nums.reverse()
    return "".join(map(str, nums))
assert f([-1, 9, 3, 1, -2]) == '-2139-1'

# GPT-4 output: "-2, 1, 3, 9, -1"
```

```
def f(nums, num):
    for i in nums:
        if nums[i]==num:
            return num
    return 'Not Found'
assert f({'elad': 186, 'colton': 162, '12': 5}, 5) == '5'

# GPT-4 output: 'Not found'
```

```
def f(text):
    dups = list(text)
    dups.append(dups[0])
    return ''.join(dups)
assert f('u') == 'uu'

# GPT-4 output: 'u'
```

```
def f(match, fill, n):
    return fill[:n] + match
assert f('9', '8', 2) == '89'

# GPT-4 output: '889'
```

```
def f(string, prefix):
    if string.startswith(prefix):
        return string.removeprefix(prefix)
    return string
assert f("Vipra", "via") == 'Vipra'

# GPT-4 output: ""
```

F.2 Input Prediction without CoT

Similarly, we highlight examples from input prediction.

F.2.1 GPT-4 Successes without CoT, Input Prediction

```
def f(l, elems):
    l.reverse()
    l.extend(elems)
    l.extend(l)
    l.reverse()
    l.reverse()
    del l[(len(l)-1):]
    return l
assert f([], [-1, 2, 7, 2, 8]) == [-1, 2, 7, 2, 8, -1, 2, 7, 2]

# GPT-3.5 output: f([2, 7, 2, 8], [-1])
# CodeLlama 34B output: f([-1, 2, 7, 2, 8], [-1, 2, 7, 2])
```

```
def f(text, position):
    length = len(text)
    index = position % length
    if position < 0 or index < 0:
        index = length // 2
    new_text = list(text)
    new_text.pop(index)
    return ''.join(new_text)
assert f('voxnzcuo', 7) == 'voxnzcu'

# GPT-3.5 output: f('voxnzcu', 42)
# CodeLlama 34B output: f('voxnzcu', -4)
```

```
def f(data, num):
    new_dict = {}
    temp = list(data.items())
    for i in range(len(temp) - 1, num - 1, -1):
        new_dict[temp[i]] = None
    return temp[num:] + list(new_dict.items())
assert f({2: 10, 3: 1}, 0) == [(2, 10), (3, 1), ((3, 1), None), ((2, 10), None)]

# GPT-3.5 output: f({(2, 10): None, (3, 1): None}, 0)
# CodeLlama 34B output: f({2: 10, 3: 1}, 2)
```

```
def f(nums):
    nums = [y for y in nums if y > 0]
    if len(nums) <= 3:
        return nums
    nums.reverse()
    half = len(nums)//2
    return nums[:half] + [0]*5 + nums[half:]
assert f([10, -1, 3, 2, -5, -6, 2, 6]) == [6, 2, 0, 0, 0, 0, 0, 2, 3, 10]

# GPT-3.5 output: f([6, 2, 0, 0, 0, 0, 0, 2, 3, 10])
# CodeLlama 34B output: f([6, 2, 3, 10])
```

```
def f(text):
    upper = 0
    lower = 0
    for i in range(len(text)):
        if text[i].islower():
            lower+=1
        else:
            upper+=1
    if lower > upper:
        return text.swapcase()
    return text
assert f('iAv,sual6eaParntEnthuhsisth,{kasae}') == 'IaV,SUAL6EApARNTeNTHUHSISTH,{KASAE}'

# GPT-3.5 output: f("\'iAv,sual6eaparnteNthuhsisth,{kasae}\'")
# CodeLlama 34B output: f("IaV,sual6eApArnTeNthuhsiStH,{KASAE}")
```

F.2.2 GPT-4 Failures without CoT, Input Prediction

```
def f(name):
    return [name[0], name[1][::-1][0]]
assert f(??) == ['m', 'a']

# GPT-4 output: f(['m', 'apple'])
```

```
def f(nums):
    if 0 in nums:
        nums.pop(nums.index(0))
        nums.insert(0, 0)
    if 1 in nums:
        nums.pop(nums.index(1))
        nums.append(1)
    return nums
assert f(??) == [0, 0, 1, 0, 1]

# GPT-4 output: f([0, 0, 1, 0, 1])
```

```
def f(text, speaker):
    while text.startswith(speaker):
        text = text[len(speaker):]
    return text
assert f(??) == 'Do you know who the other was? [NEGMENDS]'

# GPT-4 output: f('Do you know who the other was? [NEGMENDS]', '')
```

```
def f(text):
    if text.isdecimal():
        return text
```

```
    else:
        s = ''.join(sorted(text))
        if s.isdecimal():
            return 'true'
        else:
            return 'false'
assert f(??) == '133476815'

# GPT-4 output: 134678315
```

```
def f(length, text):
    if len(text) == length:
        return text[::-1]
    return False
assert f(??) == False

# GPT-4 output: f(5, "Hello")
```

F.3 Output Prediction with CoT

F.3.1 GPT-4 Successes with CoT, Output Prediction

We see that when equipped with CoT, GPT-4 has some abilities to simulate an execution process, similar to what was seen in (Nye et al., 2021).

In this example, we see both the ability to reason about the flow of the code and the ability to count uppercase letters.

```
def f(string):
    upper = 0
    for c in string:
        if c.isupper():
            upper += 1
    return upper * (2,1)[upper % 2]
assert f('PoIOarTvpoead') == 8

GPT-4: The function f(string) counts the number of uppercase letters in the input string. If the
     ↪ number of uppercase letters is even, it multiplies this count by 2. If the number of uppercase
     ↪  letters is odd, it multiplies this count by 1.

In the string 'PoIOarTvpoead', there are 4 uppercase letters: 'P', 'I', 'O', and 'T'. Since 4 is an
     ↪ even number, the function will return 4 * 2 = 8.

So, the correct output of the function for the input 'PoIOarTvpoead' is 8.

[ANSWER] 8 [/ANSWER]
```

In the next two examples, we see the ability of GPT-4 with CoT to do relatively complex string manipulation tasks.

```
def f(text):
    replaced = list(text)
    while ' ' in replaced: replaced.remove(' ')
    while '@' in replaced: replaced.remove('@')
    while '&' in replaced: replaced.remove('&')
    return ''.join(replaced)
assert f('&C C#BiOt &[(15-5%#-u&1@t') == 'CC#BiOt[(15-5%#-u1t'

GPT-4: The function `f(text)` takes a string as input and returns a new string where all spaces (' '),
     ↪  '@' symbols, and '&' symbols have been removed.
```

```
Let's apply this function to the string '&C C#BiOt &[(15-5%#-u&1@t':

1. Remove all spaces: '&CC#BiOt&[(15-5%#-u&1@t'
2. Remove all '@' symbols: '&CC#BiOt&[(15-5%#-u&1t'
3. Remove all '&' symbols: 'CC#BiOt[(15-5%#-u1t'

So, the output of the function for the given string should be 'CC#BiOt[(15-5%#-u1t'.

[ANSWER] 'CC#BiOt[(15-5%#-u1t' [/ANSWER]
```

```
def f(widths):
    reversed_names = []
    for skill in sorted(widths, key=widths.get, reverse=True):
        reversed_names.append("{0}={1}".format(widths[skill], skill))
    return '&'.join(reversed_names)
assert f({'craftsmanship':2, 'upholstery':1, 'tailoring':3}) == '3=tailoring&2=craftsmanship&1=
    ↪ upholstery'

GPT-4: The function f takes a dictionary as an argument. The dictionary keys are skill names and the
    ↪ values are their corresponding widths. The function sorts the dictionary based on the values
    ↪ in descending order. Then it creates a list of strings where each string is a combination of
    ↪ the value and the key separated by an equal sign. Finally, it joins all the strings in the
    ↪ list with an ampersand (&) and returns the resulting string.

Let's break down the steps:

1. The dictionary {'craftsmanship':2, 'upholstery':1, 'tailoring':3} is sorted based on the values in
    ↪ descending order, resulting in the list ['tailoring', 'craftsmanship', 'upholstery'].

2. For each skill in the list, a string is created with the format "{value}={key}". The resulting list
    ↪  is ['3=tailoring', '2=craftsmanship', '1=upholstery'].

3. The strings in the list are joined with an ampersand (&), resulting in the string '3=tailoring&2=
    ↪ craftsmanship&1=upholstery'.

So, the output of the function f({'craftsmanship':2, 'upholstery':1, 'tailoring':3}) should be '3=
    ↪ tailoring&2=craftsmanship&1=upholstery'.

[ANSWER]'3=tailoring&2=craftsmanship&1=upholstery'[/ANSWER]
```

In the next two examples, we see the ability to simulate and reason about for/while loops.

```
def f(nums):
    count = 0
    for i in range(len(nums)):
        if len(nums) == 0:
            break
        if count % 2 == 0:
            nums.pop()
        else:
            nums.pop(0)
        count += 1
    return nums
assert f([3, 2, 0, 0, 2, 3]) == []

GPT-4: The function `f(nums)` takes a list of numbers as input. It initializes a counter `count` to 0.
    ↪  Then it enters a loop that runs for the length of the list.

In each iteration, it checks if the length of the list is 0. If it is, it breaks the loop. This
    ↪ condition will never be true because the loop runs for the initial length of the list and not
    ↪ the current length of the list.
```

```
Then it checks if the counter is even. If it is, it removes the last element from the list. If it's
    ↪ not, it removes the first element from the list.

After each iteration, it increments the counter by 1.

So, for the input list [3, 2, 0, 0, 2, 3], the function will perform the following operations:

- Remove the last element (3) because the counter is 0 (even). The list becomes [3, 2, 0, 0, 2].
- Remove the first element (3) because the counter is 1 (odd). The list becomes [2, 0, 0, 2].
- Remove the last element (2) because the counter is 2 (even). The list becomes [2, 0, 0].
- Remove the first element (2) because the counter is 3 (odd). The list becomes [0, 0].
- Remove the last element (0) because the counter is 4 (even). The list becomes [0].
- Remove the first element (0) because the counter is 5 (odd). The list becomes [].

So, the function should return an empty list for the input [3, 2, 0, 0, 2, 3].

[ANSWER]
[]
[/ANSWER]
```

```
def f(number):
    multiplier = 2
    power = 1
    while number > 0:
        remainder = number % multiplier
        number //= multiplier
        if remainder == 0:
            power += 1
        if number == 1:
            return f'{2 ** power}-free'
assert f(8) == '16-free'

# GPT-4: The function f takes a number as input and checks if it is divisible by 2. If it is, it
    ↪ increments the power by 1 and continues to divide the number by 2 until the number is no
    ↪ longer divisible by 2. The function then returns a string that says the number is "2 to the
    ↪ power of (power + 1)-free".

In the case of the number 8, the function will divide 8 by 2 three times before it is no longer
    ↪ divisible by 2. This means the power will be incremented three times, from 1 to 4. Therefore,
    ↪ the function will return the string "2 to the power of 4-free", or "16-free".

So, the output of the code should be "16-free" for the assertion to be correct.

[ANSWER]"16-free"[/ANSWER]
```

F.3.2 GPT-4 Failures with CoT, Output Prediction

Now, we highlight some failure cases of GPT-4 output prediction with CoT. One interesting failure mode is that the CoT makes a mistake on information that GPT-4 seems to know when prompted independently. For example, in the first example below, the CoT insists that 6173 is less tan 1000. However, when prompted independently, GPT-4 knows this is not true. We provide a few examples of these paired mistakes with proof of knowledge below. The outputs are abbreviated for brevity and to highlight the failures.

The first example shows a failure in number comparison.

```
Prompt: What is the output of the code?

def f(num):
    if 0 < num < 1000 and num != 6174:
        return 'Half Life'
```

```
    return 'Not found'
assert f(6173) == ??

GPT-4: ... In the case of `f(6173)`, the number 6173 is greater than 0, less than 1000, and not equal
     ↪ to 6174, so the function will return 'Half Life'...
```

```
Prompt: If num = 6173, is it true that 0 < num < 1000 and num != 6174?

GPT-4: No, the statement is not true. The variable num is assigned the value 6173, so the condition 0
     ↪ < num < 1000 is false because 6173 is not less than 1000. The second condition, num != 6174,
     ↪ is true because 6173 is indeed not equal to 6174. However, since the first condition is false,
     ↪  the overall statement is false.
```

This example is showing a semantic misunderstanding of `str.islower`.

```
Prompt: What is the output of the code?

def f(text):
    if not text.islower():
        temp_text = ''
        for i in range(len(text)):
            if text[i].islower():
                temp_text += text[i]
        return temp_text
    return text
assert f("816a") == ??

GPT-4: ... The function `f` takes a string `text` as input and checks if the string is not all
     ↪ lowercase using the `islower()` method ...

...In the given assertion, the input string is `"816a"`. This string is not all lowercase because it
     ↪ contains digits, which are neither lowercase nor uppercase. Therefore, the function will go
     ↪ into the loop and check each character
...
```

```
Prompt: What is "816a".islower()?

GPT-4: ... The string "816a" contains digits and a lowercase alphabetic character 'a'. Since the only
     ↪ alphabetic character in the string is lowercase, the `islower()` method would return `True`
     ↪ when called on this string...
```

This example shows a failure with sorting `"h e l l o"`, which seems like it could be a tokenization artifact.

```
Prompt: What is the output of this code?

def f(str):
    return str and ''.join(sorted(str))
assert f("h e l l o") == ??

GPT-4: ...
1. The input string `"h e l l o"` is truthy, so the condition `str` is `True`.
2. The `sorted` function is called on the string, which results in the list `[' ', ' ', ' ', 'e', 'h',
     ↪  'l', 'l', 'o']`.
...
```

```
Prompt: In Python, what is sorted("h e l l o")?

GPT-4: ...Here's what happens:

```python
result = sorted("h e l l o")
```
```

```
```

The `result` will be a list that contains the characters of the string, including spaces, sorted in
    ↪ ascending order based on their Unicode code points. The space character has a lower code point
    ↪ than any of the letters, so it will appear first in the sorted list. The result will look
    ↪ like this:

```python
[' ', ' ', ' ', ' ', 'e', 'h', 'l', 'l', 'o']
```
```
```

This example shows a failure with the `str.strip` method.

```
Prompt: What is the output of this code?

def f(text, strip_chars):
    return text[::-1].strip(strip_chars)[::-1]
assert f('tcmfsmj', 'cfj') == ??

GPT-4: The function `f` takes two arguments: `text` and `strip_chars`. It reverses the `text` string,
    ↪ strips characters from both ends that are found in `strip_chars`, and then reverses the string
    ↪  back to its original order.

Let's break down what happens step by step with the input `f('tcmfsmj', 'cfj')`:

1. The string ``tcmfsmj`` is reversed, resulting in ``jmsfmct``.
2. The `strip` method is called with ``cfj`` as the argument, which removes any leading or trailing
    ↪ characters found in ``cfj`` from the reversed string ``jmsfmct``.
3. The leading ``j`` and the trailing ``c`` and ``t`` are stripped from the string, resulting in ``
    ↪ msfm``.
4. The stripped string ``msfm`` is then reversed again, resulting in ``mfsm``.

Therefore, the assertion should be:

```python
assert f('tcmfsmj', 'cfj') == 'mfsm'
```

The output of the code, if the assertion passes, is nothing since assertions do not produce output if
    ↪ the condition is true.
```

```
Prompt: In Python, what is "jmsfmct".strip("cfj")?

GPT-4: ... Here's what happens step by step:

- The first character 'j' is in the set of characters to remove ('cfj'), so it is removed.
- The next character 'm' is not in the set of characters to remove, so the stripping stops at the
    ↪ beginning of the string.
- Looking at the end of the string, the last character 't' is not in the set of characters to remove,
    ↪ so the stripping stops at the end of the string without removing anything.

The resulting string after the `strip` operation will be `"msfmct"`.
```

In this example, we observe a list indexing error.

```
Prompt: What is the output of this code?

def f(nums):
    count = len(nums)
    for i in range(-count+1, 0):
        nums.extend([nums[i], nums[i]])
    return nums
assert f([0, 6, 2, -1, -2]) == ??
```

```
GPT-4: ...
- Initial list: `[0, 6, 2, -1, -2]`
- i = -4: Extend with two copies of `nums[-4]` which is `6`. New list: `[0, 6, 2, -1, -2, 6, 6]`
- i = -3: Extend with two copies of `nums[-3]` which is `2`. New list: `[0, 6, 2, -1, -2, 6, 6, 2, 2]`
...
```

```
Prompt: In Python, what is the value at index -3 of [0, 6, 2, -1, -2, 6, 6]?

GPT-4: In Python, negative indices count from the end of the list, with -1 being the last element. To
    ↪ find the value at index -3, we count three places from the end of the list.

Given the list:

```
[0, 6, 2, -1, -2, 6, 6]
```

The indices of the elements are as follows:

```
 0   1   2   3   4   5   6  (positive indices)
-7  -6  -5  -4  -3  -2  -1  (negative indices)
```

So, the value at index -3 is `-2`.
```

In this example, we see a string indexing error.

```
Prompt: In Python, what is the output of this code?

def f(text):
    for i in range(-len(text), 0):
        if text[i].islower():
            return text[:i] + text[i].swapcase() + text[-i:]
    return text
assert f('Boo5') == ??

GPT-4: ...
- When `i = -3`, `text[i]` is `'o'`, which is lowercase. So the function will return a new string
    ↪ according to the rule above...

- `text[-i:]` is `'o5'` (the substring from the character after the current character to the end of
    ↪ the string).

Therefore, the function will return `'BOo5'`...
```

```
Prompt: In Python, if text = "Boo5" and i = -3, what is text[-i:]?

GPT-4: ...Given `text = "Boo5"` and `i = -3`, let's calculate `-i`:

`-i = -(-3) = 3`

So, `text[-i:]` is equivalent to `text[3:]`, which means we start from the 3rd index (keeping in mind
    ↪ that indexing starts at 0) and take all characters from that position to the end of the string
    ↪ .

The character at index 3 in the string "Boo5" is '5'. Therefore, `text[-i:]` will return '5'.
```

F.4 Input Prediction with CoT

F.4.1 GPT-4 Successes with CoT, Output Prediction

The following examples highlights code reasoning abilities.

```
def f(text, size):
    counter = len(text)
    for i in range(size-int(size%2)):
        text = ' '+text+' '
        counter += 2
        if counter >= size:
            return text
assert f(??) == '     7     '

GPT-4: The function f takes two arguments: a text and a size. It adds spaces to the beginning and end
     ↪ of the text until the length of the text is greater than or equal to the size. The function
     ↪ stops adding spaces when the length of the text is greater than or equal to the size.

In this case, the output is '     7     ', which is a string of length 11. The text in the middle is
     ↪ '7', which is of length 1. The function adds 5 spaces to the beginning and end of the text to
     ↪ reach a total length of 11.

So, the input to the function should be '7' and 11.

[ANSWER]
f('7', 11)
[/ANSWER]
```

```
def f(text):
    uppercase_index = text.find('A')
    if uppercase_index >= 0:
        return text[:uppercase_index] + text[text.find('a') + 1 :]
    else:
        return ''.join(sorted(text))
assert f(??) == '   DEGHIVjkptx'

GPT-4: The function f takes a string as input. It first checks if there is an uppercase 'A' in the
     ↪ string. If there is, it removes the first occurrence of lowercase 'a' from the string and
     ↪ returns the string. If there is no uppercase 'A', it sorts the string in alphabetical order
     ↪ and returns it.

In this case, the output is '   DEGHIVjkptx', which is a sorted string. This means that the input
     ↪ string did not contain an uppercase 'A', and the function sorted the string.

To get this output, we can input a string that contains the same characters but in a different order.
     ↪ For example, the string 'xtpkjVIGHED   ' contains the same characters and when sorted by the
     ↪ function, it will give the output '   DEGHIVjkptx'.

So, the answer is 'xtpkjVIGHED   '.

[ANSWER]
'xtpkjVIGHED   '
[/ANSWER]
```

The next example shows some ability in hexadecimal conversion. This may be partially due to memorization and data contamination effects, but this exact hex string was not found after a cursory Google search.

```
def f(text):
    hex_string = text.encode('utf-16')
    result = "0x"
    for c in hex_string:
```

```
        result += "%.2X" % c
    return result
assert f(??) == '0xFFFE66006E007700200052006F006D0065002100'

GPT-4: The function f takes a string as input, encodes it in utf-16 format, and then converts each
     ↪ character into a hexadecimal representation. The output '0
     ↪ xFFFE66006E007700200052006F006D0065002100' is the hexadecimal representation of a string in
     ↪ utf-16 format.

To find the input, we need to convert the hexadecimal representation back to a string. We can do this
     ↪ by first removing the '0x' prefix and then converting each pair of hexadecimal digits back to
     ↪ a character.

The hexadecimal string 'FFFE66006E007700200052006F006D0065002100' corresponds to the utf-16 encoded
     ↪ string 'nw Rome!'.

So, the input to the function f that will give the output '0xFFFE66006E007700200052006F006D0065002100'
     ↪  is 'nw Rome!'.

[ANSWER] f('nw Rome!') [/ANSWER]
```

Just like we see in output prediction, GPT-4 is able to reason about non-trivial string manipulation code.

```
def f(text):
    text = text.split(',')
    text.pop(0)
    text.insert(0, text.pop(text.index('T')))
    return 'T' + ',' + ',.join(text)
assert f(??) == 'T,T,Sspp,G ,.tB,Vxk,Cct'

GPT-4: The function f takes a string as input, splits it into a list of substrings divided by commas,
     ↪ removes the first element of the list, finds the index of the substring 'T', removes it from
     ↪ its current position and inserts it at the beginning of the list. Finally, it joins the list
     ↪ into a string separated by commas and adds 'T' at the beginning of the string.

To get the output 'T,T,Sspp,G ,.tB,Vxk,Cct', we need to have 'T' as the second element in the list
     ↪ after splitting the input string by commas. The rest of the elements can be in any order. So,
     ↪ one possible input could be 'any,T,Sspp,G ,.tB,Vxk,Cct'.

Let's check this:

f('any,T,Sspp,G ,.tB,Vxk,Cct') will split the input into ['any', 'T', 'Sspp', 'G ', '.tB', 'Vxk', 'Cct
     ↪ '], remove the first element to get ['T', 'Sspp', 'G ', '.tB', 'Vxk', 'Cct'], find the index
     ↪ of 'T' (which is 0), remove it from the list and insert it at the beginning to get the same
     ↪ list ['T', 'Sspp', 'G ', '.tB', 'Vxk', 'Cct'], join the list into a string 'T,Sspp,G ,.tB,Vxk,
     ↪ Cct' and add 'T' at the beginning to get 'T,T,Sspp,G ,.tB,Vxk,Cct', which is the desired
     ↪ output.

So, the answer is 'any,T,Sspp,G ,.tB,Vxk,Cct'.

[ANSWER] 'any,T,Sspp,G ,.tB,Vxk,Cct' [/ANSWER]
```

In the following example, GPT-4 shows simple code analysis abilities, understanding that the first case will produce an error.

```
def f(nums):
    count = len(nums)
    if count == 0:
        nums = [0] * int(nums.pop())
    elif count % 2 == 0:
        nums.clear()
    else:
```

```
        del nums[:count//2:]
    return nums
assert f(??) == []

GPT-4: The function f takes a list of numbers as input. If the length of the list is 0, it tries to
     ↪ pop an element from the list, which will cause an error because the list is empty. If the
     ↪ length of the list is even, it clears the list. If the length of the list is odd, it deletes
     ↪ the first half of the list.

To get an empty list as output, we can input an even-length list.

For example, we can input a list with two elements: [1, 2].

So, the answer is [1, 2].

[ANSWER]
[1, 2]
[/ANSWER]
```

F.4.2 GPT-4 Failures with CoT, Input Prediction

As with output prediction, also notice some relatively simple failures of CoT. One observation is that GPT-4 often tries to check the incorrect input against the assertion and says that it passes, even when it does not. We believe this is not necessarily just due to the lack of execution ability, but a prior that the solution it found is more likely to be correct.

In these first three examples, the model goes directly to a wrong answer without much step-by-step reasoning.

```
def f(text):
    odd = ''
    even = ''
    for i, c in enumerate(text):
        if i % 2 == 0:
            even += c
        else:
            odd += c
    return even + odd.lower()
assert f(??) == 'Mmohamt'

GPT-4: The function f takes a string as input and separates the characters at even and odd indices. It
     ↪  then concatenates the characters at even indices with the characters at odd indices converted
     ↪  to lowercase.

To get the output 'Mmohamt', we can see that the even indexed characters are 'M', 'o', 'a', 't' and
     ↪ the odd indexed characters are 'm', 'h', 'm'.

So, the input string can be 'Mmohamt'.

[ANSWER]
f('Mmohamt')
[/ANSWER]
```

```
def f(text):
    string = ''
    for char in text:
        string += char + char.lower()
    return string
assert f(??) == 'llaallaakk'
```

```
GPT-4: The function f takes a string as input and for each character in the string, it appends the
    ↪ character itself and its lowercase version to a new string. The function then returns this new
    ↪  string.

Given the output 'llaallaakk', we can see that each character is repeated twice, once in its original
    ↪ form and once in lowercase. This means that the input string must have been 'LAK'.

So, the input that leads to the given output when passed to function f is 'LAK'.

[ANSWER]
'LAK'
[/ANSWER]
```

```
def f(nums, i):
    nums.append(nums.pop(i % len(nums)))
    nums[i % len(nums)] = '?'
    split = nums[::2]
    split.remove('?')
    return split + nums[1::2]
assert f(??) == [2, 4, 4]

GPT-4: The function f takes a list of numbers and an index as input. It pops the element at the index
    ↪ (modulo the length of the list) and appends it to the end of the list. Then it replaces the
    ↪ element at the same index with a '?'. It then splits the list into two lists: one with
    ↪ elements at even indices and one with elements at odd indices. It removes the '?' from the
    ↪ first list and returns the concatenation of the two lists.

To get the output [2, 4, 4], we need to find a list and an index such that after popping the element
    ↪ at the index and appending it to the end, and replacing the element at the index with '?', the
    ↪  list split into even and odd indexed elements (with '?' removed) gives [2, 4, 4].

One possible input could be the list [2, 4, 4, 4] and the index 1. After popping the element at index
    ↪ 1 and appending it to the end, the list becomes [2, 4, 4, 4]. Replacing the element at index 1
    ↪  with '?' gives [2, '?', 4, 4]. Splitting into even and odd indexed elements gives [2, 4] and
    ↪ ['?', 4]. Removing '?' from the first list gives [2, 4], and concatenating the two lists gives
    ↪  [2, 4, 4].

So, the input f([2, 4, 4, 4], 1) will give the output [2, 4, 4].
```

The following example is a bit tricky, because the naming of the variables does not match the semantics of str.removeprefix.

```
def f(prefix, s):
    return str.removeprefix(prefix, s)
assert f(??) =='hymi'

GPT-4: First, we need to understand what the function does. The function f takes two arguments: a
    ↪ prefix and a string s. It then removes the prefix from the string s and returns the result.

Given that the output is 'hymi', we can infer that the string s must have ended with 'hymi' after the
    ↪ prefix was removed. Therefore, we can choose any prefix and append 'hymi' to it to form the
    ↪ string s.

For example, if we choose the prefix to be 'hello', then the string s would be 'hellohymi'. When the
    ↪ prefix 'hello' is removed from 'hellohymi', the result is 'hymi', which is the desired output.

So, the input to the function f that would give the output 'hymi' is ('hello', 'hellohymi').

[ANSWER]
('hello', 'hellohymi')
[/ANSWER]
```

The following example shows a case where GPT-4 "checks" the forward execution incorrectly.

```
def f(text):
    count = len(text)
    for i in range(-count+1, 0):
        text = text + text[i]
    return text
assert f(??) =='wlace Alc l  '

GPT-4: The function f takes a string as input and appends the characters from the end of the string to
     ↪  the string itself. The number of characters appended is equal to the length of the original
     ↪ string minus one.

To get the output 'wlace Alc l  ', we need to find a string that when appended with its last len(
     ↪ string)-1 characters, gives 'wlace Alc l  '.

By observing the output, we can see that the last 3 characters are spaces. This means that the last 3
     ↪ characters of the original string must also be spaces.

The first character of the output is 'w', which is the first character of the original string. The
     ↪ second character of the output is 'l', which is the last character of the original string.

From this, we can infer that the original string is 'w l  '.

Let's check this:

f('w l  ') = 'w l  ' + ' l  ' = 'wlace Alc l  '}

So, the input that gives the output 'wlace Alc l  ' when passed to the function f is 'w l  '.

So, the answer is:

[ANSWER] 'w l  ' [/ANSWER]
```

F.4.3 GPT-4 Failures with CoT, Examples Requiring Insights

Unlike output prediction, where stepping through the code line by line suffices, some input prediction examples require insights and observations that can be difficult for models like GPT-4 to make. These examples are a bit more challenging, and passing them can require making these insights. Here, we highlight some of these examples, corresponding relevant insights, and GPT-4's failure modes.

In the following example, `new_nums` consists of a sorted portion and an unsorted portion. One insight is that the sorted portion must be a subarray of the unsorted portion. GPT-4 always takes `[5, 15, 25, 44]` to be the sorted portion and `[24, 22, 38, 5, 35, 15]` to be the unsorted portion, which cannot be the case because 44 (from the sorted portion) is not in the unsorted portion.

```
def f(nums, a, b):
    new_nums = []
    for n in nums:
         if n < a or n > b:
             new_nums.append(n)
    new_nums.sort()
    new_nums.extend(nums)
    return new_nums
assert f(??) == [5, 15, 25, 44, 24, 22, 38, 5, 35, 15]
```

In the following example, the simplest solution is to bypass the while loop completely, but the model does not find it. However, the model chooses an output like `"baec"` because it back-translates the "a" to "i" using the translation table. Unfortunately, it does not take into account that other characters also get translated in the translation table.

```
def f(input_string):
    table = str.maketrans('aioe', 'ioua')
    while 'a' in input_string or 'A' in input_string:
        input_string = input_string.translate(table)
    return input_string
assert f(??) == 'biec'
```

In the following example, one must notice that `x` cannot end with a number and `y` cannot start with a number. Reasoning about `x+y = 'nisou79-85233'`, GPT-4 incorrectly deduces that the `y` portion must be `85233`, while in reality, it must be `79-85233` because `x` cannot end with a digit.

```
def f(urn):
    pieces = urn.split('--')
    x = pieces[0].rstrip('0123456789')
    y = pieces[1].lstrip('0123456789')
    return x+y
assert f(??) == 'nisou79-85233'
```

In the following example, it is very easy to solve the task by choosing `source` and `target` to make `target in source` false, going directly to the last line. However, GPT-4 tries to go down the more complicated path.

```
def f(source, target):
    if target in source:
        prefix, suff = source.split(target)
        prefix = prefix[:len(prefix) // 2] + target + prefix[len(prefix) // 2:]
        return prefix + suff
    return source
assert f('abcjava', 'python') =='abcjava'
```

In the following example, an important insight is that the last line of the function is a `"...."`.`join()`, but that `"...."` does not occur in the output. This confuses the model.

```
def f(letters):
    letters_only = letters.strip(".,!?*")
    return "....".join(letters_only.split(" "))
assert f(??) =='h,e,l,l,o,wo,r,ld'
```

F.5 Simple and Isolated GPT-4 Failures

Some of the failures from both input prediction and output prediction can be isolated to capabilities on smaller subtasks. Here, we highlight some interesting subtasks, as well as GPT-4's response to these subtasks.

```
Prompt: What is a string containing 'a' three times, 'b' three times, 'c' twice, 'd' three times, and
    ↪ 'z' twice?

GPT-4: A string containing the specified number of each letter can be constructed in many different
    ↪ ways, as the order of the characters is not specified. Here is one example of such a string:

‘aaabbbccdzz‘
```

```
Prompt: In Python, what is " BaB ".rfind(" B ")?

GPT-4: In Python, the ‘rfind()‘ method of a string object returns the highest index of the substring
    ↪ if found in the given string. If the substring is not found, it returns -1.

The string ‘" BaB "‘ has a space, followed by a capital ’B’, followed by another space as the
    ↪ substring we are looking for with ‘rfind(" B ")‘.

Let’s break down the string ‘" BaB "‘:

- Index 0: space (’ ’)
- Index 1: ’B’
- Index 2: ’a’
- Index 3: ’B’
- Index 4: space (’ ’)

We are looking for the substring ‘" B "‘ (space, ’B’, space). This substring appears only once in the
    ↪ string, starting at index 3 and ending at index 5 (if we consider the space after the last ’B
    ↪ ’).

Therefore, ‘" BaB ".rfind(" B ")‘ will return 3, because that is the highest index at which the
    ↪ substring ‘" B "‘ starts.
```

```
Prompt: In Python, if I have a string s = ’iabnm~~~~~~~~~~’, what is s[1::2]?

GPT-4: In Python, the slicing operation ‘s[start:stop:step]‘ allows you to extract a substring from a
    ↪ string ‘s‘ by specifying a starting index ‘start‘, a stopping index ‘stop‘, and a step ‘step‘.
    ↪  If ‘start‘ is omitted, it defaults to the beginning of the string (index 0). If ‘stop‘ is
    ↪ omitted, it defaults to the end of the string. If ‘step‘ is omitted, it defaults to 1.

In your example, ‘s[1::2]‘ means that you want to start at index 1 of the string ‘s‘, go until the end
    ↪  of the string (since no stop index is provided), and select every second character (because
    ↪ the step is 2).

Given the string ‘s = ’iabnm~~~~~~~~~~’‘, let’s break down what ‘s[1::2]‘ will return:

- Start at index 1: ’a’
- Every second character from index 1: ’abnm~~~~~~~~~~’ -> ’an~’

So, ‘s[1::2]‘ will return the string ‘’an~’‘.
```

```
Prompt: In Python, what is "+".join([’*’, ’+’, ’n’, ’z’, ’o’, ’h’])?

GPT-4: In Python, the ‘join‘ method is used to concatenate a sequence of strings with a specified
    ↪ separator. In the example you provided, ‘"+".join([’*’, ’+’, ’n’, ’z’, ’o’, ’h’])‘, the ‘join‘
    ↪  method is called on the string ‘"+"‘, which is the separator, and it concatenates the
    ↪ elements of the list ‘[’*’, ’+’, ’n’, ’z’, ’o’, ’h’]‘ with a plus sign between each element.

The result of this operation would be a single string where each element from the list is separated by
    ↪  a plus sign:

‘‘‘
*+*+n+z+o+h
‘‘‘

Each element in the list is a string, and they are joined together in the order they appear in the
    ↪ list, with the ‘+‘ character inserted between each pair of elements.
```

```
Prompt: In Python, if text = "!123Leap and the net will appear" and res = 123, what is text[len(str(
    ↪ res)):]?

GPT-4: In the given Python code snippet, you have a string ‘text‘ and an integer ‘res‘.
```

```
```python
text = "!123Leap and the net will appear"
res = 123
```

You want to know what `text[len(str(res)):]` evaluates to. Let's break it down step by step:

1. `str(res)` converts the integer `res` to a string, which is `"123"`.
2. `len(str(res))` calculates the length of the string `"123"`, which is `3`.
3. `text[len(str(res)):]` slices the string `text` starting from the index equal to the length of the
    ↪ string representation of `res` (which is `3`) to the end of the string.

So, `text[len(str(res)):]` will return the substring of `text` starting from the 4th character (since
    ↪ indexing in Python starts at 0) to the end of the string.

The 4th character in the string `text` is the character `'L'` (the first three characters are `"!12"`)
    ↪ , so the result will be:

```python
"Leap and the net will appear"
```

This is the substring of `text` starting from the 4th character to the end.
```

```
Prompt: In Python, what is "pomodoro".replace("or", "pomodoro")?

GPT-4: In Python, the `replace` method is used to replace occurrences of a substring within a string
    ↪ with another substring. When you call `"pomodoro".replace("or", "pomodoro")`, you are asking
    ↪ Python to replace every occurrence of the substring `"or"` in the string `"pomodoro"` with the
    ↪  substring `"pomodoro"`.

Here's what happens step by step:

1. Python finds the first occurrence of `"or"` in `"pomodoro"`, which is at the end of the string.
2. Python replaces this occurrence with `"pomodoro"`, resulting in the string `"pomodopomodoro"`.

There is only one occurrence of `"or"` in the original string, so the final result of the `replace`
    ↪ method in this case is `"pomodopomodoro"`.
```

F.6 Failing Gems

Next, we identify a collection of examples that we find GPT-4 often fails on, even with CoT. Some of these examples overlap with examples shown above, but others are new. Overall, we believe this collection of examples provides an interesting lens to understand the behaviour of GPT-4 on input and output prediction.

F.6.1 Failing Gems, Output Prediction

```python
def f(nums):
    for i in range(-len(nums), 0):
        nums.insert(-i, nums[i])
    return nums
assert f([-6, -2, 1, -3, 0, 1]) == [-6, -6, -2, 1, 1, 1, -3, 0, 0, 1, 1, -6]

def f(text):
    if not text.islower():
        temp_text = ''
        for i in range(len(text)):
```

```
            if text[i].islower():
                temp_text += text[i]
        return temp_text
    return text
assert f("816a") == '816a'

def f(list, separator):
    text = separator.join(list)
    return separator.join(reversed(text))
assert f(['is', 'it', 'top'], '@') == 'p@o@t@@@t@i@@@s@i'

def f(text, res):
    for c in '*\n"':
        text = text.replace(c, '!' + str(res))
    if text.startswith('!'):
        text = text[len(str(res)):]
    return text
assert f('"Leap and the net will appear', 123) == '3Leap and the net will appear'

def f(num):
    if 0 < num < 1000 and num != 6174:
        return 'Half Life'
    return 'Not found'
assert f(6173) == 'Not found'

def f(date):
    return  date[6:] + date[4:6] + date[0:4]
assert f("08-10-2009") == '20090-08-1'

def f(text, suffix):
    if suffix and suffix[-1] in text:
        return f(text.rstrip(suffix[-1]), suffix[:-1])
    else:
        return text
assert f('rpyttc', 'cyt') == 'rpytt'

def f(s, x):
    count = 0
    for i, c in enumerate(s):
        if x in s[i:] and x not in s[:i]:
            count += 1
    return count
assert f('fvyijrtwrjrsasgt', 'g') == 15

def f(text):
    segments = text.split()
    for i in range(len(segments)):
        segments[i] = segments[i][0].upper() + segments[i][1:-1] + segments[i][-1].upper()
    return ' '.join(segments)
assert f("hey !") == 'HeY !!'

def f(pattern, items):
    result = []
    for text in items:
        pos = text.rfind(pattern)
        if pos >= 0:
            result.append(pos)

    return result
assert f(" B ", [" bBb ", " BaB ", " bB", " bBbB ", " bbb"]) == []

def f(str):
    return str and ''.join(sorted(str))
assert f("h e l l o") == '    ehllo'
```

```
def f(t):
    return t.replace('or', t.center(len(t), 'o'))
assert f("pomodoro") == 'pomodpomodoroo'
```

F.6.2 Failing Gems, Input Prediction

```
def f(dimension):
    dinline = str(dimension)[1:].zfill(2)
    return dinline[0] * int(dinline[1])
assert f(??) == 'kkkkk'

def f(text):
    for elem in text:
        if elem.isupper():
            try:
                text.remove(elem)
            except ValueError:
                pass
    return text
assert f(??) == ''

def f(text):
    ls = list(text)
    for i in range(0, len(ls)):
        if ls[i]!='+':
            ls.insert(i, '+')
            ls.insert(i, '*')
            break
    return '+'.join(ls)
assert f('nzoh') == '*+++n+z+o+h'

def f(text):
    new_text = list(text)
    dict = {}
    for char in new_text:
        dict[char] = new_text.count(char)
    return dict
assert f('aaabbbccdddzz') == {'a': 3, 'b': 3, 'c': 2, 'd': 3, 'z': 2}

def f(text):
    odd = ''
    even = ''
    for i, c in enumerate(text):
        if i % 2 == 0:
            even += c
        else:
            odd += c
    return even + odd.lower()
assert f('Mammoth') == 'Mmohamt'

def f(nums, i):
    nums.append(nums.pop(i % len(nums)))
    nums[i % len(nums)] = '?'
    split = nums[::2]
    split.remove('?')
    return split + nums[1::2]
assert f([4, 2, 4, 2], 0) == [2, 4, 4]

def f(prefix, s):
    return str.removeprefix(prefix, s)
assert f('hymi', 'hymifulhxhzpnyihyf') == 'hymi'
```

```
def f(text):
    if ',' in text:
        before, _, after = text.partition(',')
        return after + ' ' + before
    return ',' + text.partition(' ')[-1] + ' 0'
assert f('244, 105, -90') == ' 105, -90 244'

def f(s):
    return '{}{}{}'.format(s[3:], s[2], s[5:8])
assert f('jbucwc') == 'cwcuc'

def f(nums):
    for i in range(len(nums)):
        nums.insert(i, nums[i]**2)
    return nums
assert f([1, 2, 4]) == [1, 1, 1, 1, 2, 4]

def f(c, text):
    t = c
    for c in reversed(text):
        t = c + t*2
        t = c + t
    return t + text
assert f(';?', 'i') == 'ii;?;?i'

def f(nums, location, item):
    if len(nums) >= location and 0 <= location:
        return nums.insert(location, item)
    return nums
assert f([1, 2, 3, 4, 5, 6], -5, -5) == [1, 2, 3, 4, 5, 6]

def f(text):
    return max(text.find(ch) for ch in 'aeiou')
assert f("qsqgijwmmhbchoj") == 13
```