CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution

Alex Gu[⋆] *gua@mit.edu*
MIT CSAIL
Baptiste Rozière *broz@meta.com*
Meta AI
Hugh Leather *hleather@meta.com*
Meta AI
Armando Solar-Lezama *asolar@csail.mit.edu*
MIT CSAIL
Gabriel Synnaeve *gab@meta.com*
Meta AI
Sida I. Wang *sida@meta.com*
Meta AI

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 generic recipe for generating our execution benchmark which can be used to create future variation of the benchmark. Second, we evaluate twenty code models on our benchmark and discover that many recent high-scoring models on HumanEval do not show the same improvements on our benchmark. Third, we show that simple CoT and fine-tuning schemes can improve performance on our benchmark but remain far from solving it. 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, highlighting the gap between open and closed source models. As no model is close to acing CRUXEval, we provide examples of consistent GPT-4 failures on simple programs as a lens into its code reasoning capabilities and areas for improvement.

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

[⋆] Work primarily done during an internship at Meta AI

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 write code more efficiently.

The primary way that the community has been evaluating 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. An example of a sample in CRUXEval is shown in Listings 1 and 2 (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 (in our opinion). CRUXEval 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.

Listing 1: Sample problem

```
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
```

Listing 2: Sample problem

```
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
```

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. We use this approach because while it is difficult to manually come up with example where the strongest models like GPT-4 fail completely, we observe that they fail quite often on random yet reasonable 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. While these models have impressively high HumanEval scores, they do not perform much better than their base models on CRUXEval.

We also observe that CoT and fine-tuning on input-output assertions are effective techniques for improving performance on CRUXEval, but are 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

LMs for Code Generation: There have been many efforts training LMs to generate code. Base models include Codex (Chen et al., 2021), CodeGeeX (Zheng et al., 2023), SantaCoder (Allal et al., 2023), PolyCoder (Xu et al., 2022), InCoder (Fried et al., 2022), CodeGen (Nijkamp et al., 2022), StarCoder (Li et al., 2023a), DeepSeek-Coder (AI, 2023), and Code Llama (Roziere et al., 2023). Later, some of these models were fine-tuned on instruction-like data distilled from GPT-3.5 and GPT-4, resulting in models like Phind (Royzen et al., 2023), WizardCoder (Luo et al., 2023), and Phi-1/Phi-1.5 (Li et al., 2023b; Gunasekar et al., 2023). We evaluate the performance of a selection of these models on our CRUXEval.

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., 2023h) 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., 2023c) augments HumanEval with better test cases after discovering many passing solutions are incorrect. ReCode (Wang et al., 2022a) is a variant of HumanEval with perturbed function names and docstrings. HumanEval-X (Zheng et al., 2023), MultiPLe (Cassano et al., 2022), and MBXP (Athiwaratkun et al., 2022) are extensions of HumanEval and MBPP with a focus on including programming languages outside of Python. APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), and LeetCode-Hard (Shinn et al., 2023) evaluate code generation on more difficult, interview or competition style problems.

There are also benchmarks to evaluate code generation in data science applications, such as DS-1000 (Lai et al., 2023), ARCADE (Yin et al., 2022), NumpyEval (Zhang et al., 2023b), and PandasEval (Jain et al., 2022). Going one step further, some benchmarks also measure ability to use API's or perform more general software engineering tasks, such as JuICe (Agashe et al., 2019), APIBench (Patil et al., 2023), RepoBench (Liu et al., 2023e), ODEX (Wang et al., 2022b), SWE-Bench (Jimenez et al., 2023), GoogleCodeRepo (Shrivastava et al., 2023), RepoEval (Zhang et al., 2023a), and Cocomic-Data (Ding et al., 2022).

Finally, there are a variety of benchmarks for other tasks, such as code translation (Roziere et al., 2020; Zhu et al., 2022; Ahmad et al., 2021), test case generation (Tufano et al., 2022; Watson et al., 2020), code search (Husain et al., 2019), type prediction (Mir et al., 2022; Wei et al., 2023; Malik et al., 2019), commit message generation (Liu et al., 2020), code summarization (LeClair et al., 2019; Iyer et al., 2016; Barone & Sennrich, 2017; Hasan et al., 2021; Alon et al., 2018), code security (Liguori et al., 2022; Pearce et al., 2022; Tony et al., 2023), program repair (Jiang et al., 2023b; Xia et al., 2022; Tufano et al., 2019; Haque et al., 2022; Jin et al., 2023; Gupta et al., 2017; Berabi et al., 2021), performance optimization (Garg et al., 2022; Madaan et al., 2023a), and so on.

To our knowledge, our CRUXEVAL is the first publicly available benchmark to measure the execution ability of code LMs. While some prior work has measured the output prediction ability of code LMs, we leverage our CRUXEVAL-O to perform a more thorough investigation of these capabilities. Our CRUXEVAL-I is the first to measure the ability of code LMs to perform input prediction.

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., 2023g), MBR-EXEC (Shi et al., 2022) TCoT (Tian & Chen, 2023), Algo (Zhang et al., 2023d), 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. Other works use RL-style execution feedback to improve code generation, including CodeRL (Le et al., 2022), Reflexion (Shinn et al., 2023), and PG-TD (Zhang et al., 2023e). (Chen et al., 2023; Olausson et al., 2023b; Madaan et al., 2023b; Peng et al., 2023; Zhang et al., 2023c) investigate self-repair, using error messages as feedback for models to improve.

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. Inspired by these works, we briefly touch on two primitive ways to improve performance on our benchmark, chain-of-thought and fine-tuning. Moving forward, we believe our CRUXEVAL could serve as a useful reference point as more techniques are designed to improve code execution abilities.

Failure modes of LM Reasoning: Another dream of the community is to better understand the failure modes of LMs on reasoning tasks. Bubeck et al. (2023); Liu et al. (2023b); Arkoudas (2023); Zhang et al. (2022); Dziri et al. (2023); Olausson et al. (2023a); Lee et al. (2023); Zhang et al. (2023f) all investigate and point out various failure modes of LMs on a wide variety of reasoning tasks. Other examples of reasoning failures include 1) understanding negation (Hosseini et al., 2021), 2) ignoring irrelevant context (Shi et al., 2023), 3) operating under counterfactual situations such as 1-indexed Python or base-9 addition (Wu et al., 2023), and 4) generating Python code after identifier swaps like `print, len = len, print` (Miceli-Barone et al., 2023). Taking a more theoretical perspective, Dziri et al. (2023); Zhou et al. (2023); Merrill & Sabharwal (2023); Giannou et al. (2023) characterize the types of reasoning tasks transformers can and cannot be expected to carry out. Merrill et al. (2021) argues that it is not possible to learn meaning from ungrounded form with context dependence and assuming that syntax is independent of semantics. In this work, we use CRUXEVAL to empirically examine failures in code execution / reasoning.

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 A.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 A.2 for more details). A sample prompt is shown in Listing 11.

We use a total of 69 different functions from the standard library: 47 from the `str`, 11 from `dict`, and 11 from `list` (see Appendix A.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 solvable by a human without extra memory.

Concretely, we filter based on the following criteria.

- 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`.
- Runtime: no float 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.
- Best effort to remove other undesirable code: function cannot have any imports (such as os, random), must be deterministic (random, set ordering), and cannot have side effects such as input, `__builtins__`.

3.3 Data size and measuring noise

The success of HumanEval (164 examples) shows that evaluation benchmarks can be small where faster and cheaper evaluation is an overlooked advantage. Since additional examples are easy to generate, we first overgenerate and then measure if the noise is sufficiently small on a smaller dataset.

Out of all the samples, Code Llama 34B outperforms Code Llama 13B as expected and we would like to retain this property with high confidence in a smaller dataset. To do this, we took bootstrap samples of size N out of ∼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 sources 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 $A > B$ in aggregate, the measured performance depends on which data points are included. We can measure both noise on each model individually, and also measure type 1) noise on pairs of models using bootstrap. Fortunately, we do not see major differences between models and the most important factor is just the size of dataset. Type 1) noise is generally around 1.5% for each model whereas type 2) 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 $\alpha = 0.05$ level for many model pairs.

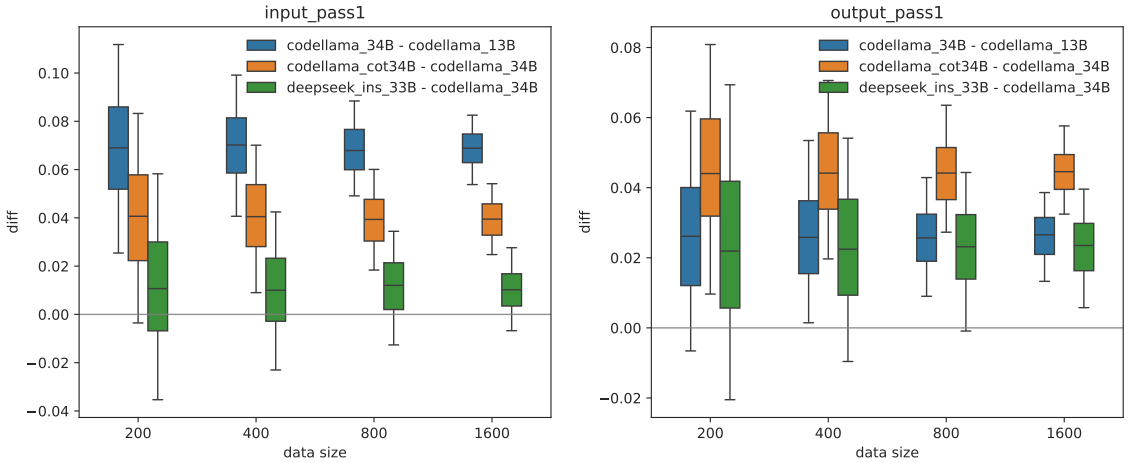


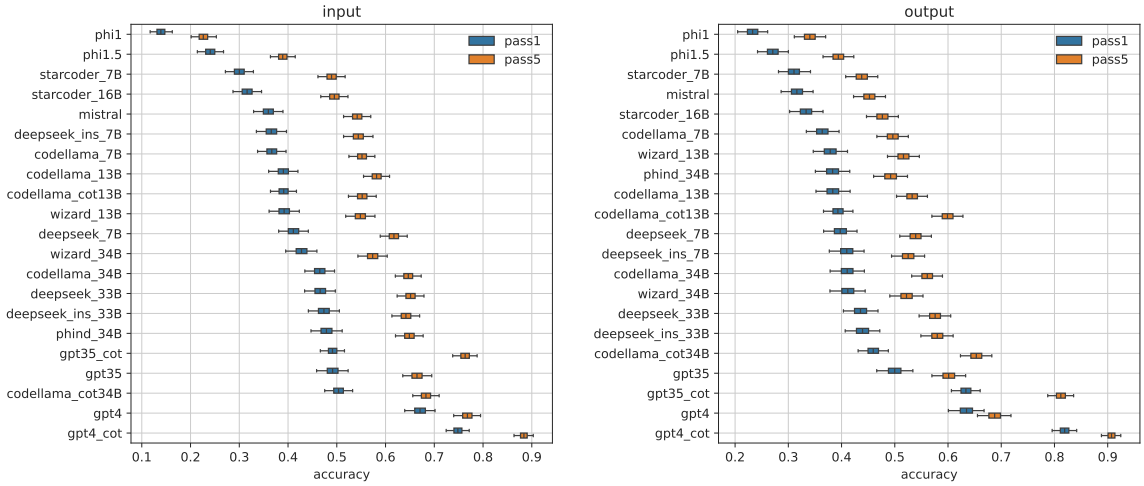
Figure 1: Difference between model pairs on bootstrap samples of various sizes. The whiskers show (2.5, 97.5) and boxes show (25, 75) percentiles.

4 Evaluation

We evaluate a selection of models on CRUXEval: StarCoder (Base 7B, 15.5B) (Li et al., 2023a), Mistral (7B) (Jiang et al., 2023a), WizardCoder (13B, 34B) (Luo et al., 2023), Phi-1 Gunasekar et al. (2023) and Phi-1.5 (Li et al., 2023b) (1.3B), Phind v2 (Royzen et al., 2023) (34B), Code Llama (Roziere et al., 2023) (Base and Python 7B, 13B, 34B), DeepSeek Coder (Base and Instruct 6.7B, 33B), GPT-3.5

(Brown et al., 2020; Ouyang et al., 2022), and GPT-4 (OpenAI, 2023). To facilitate reproducibility, the HuggingFace checkpoints of non-GPT models are in Appendix B and all prompts are in Appendix D.2.

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. 16. 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

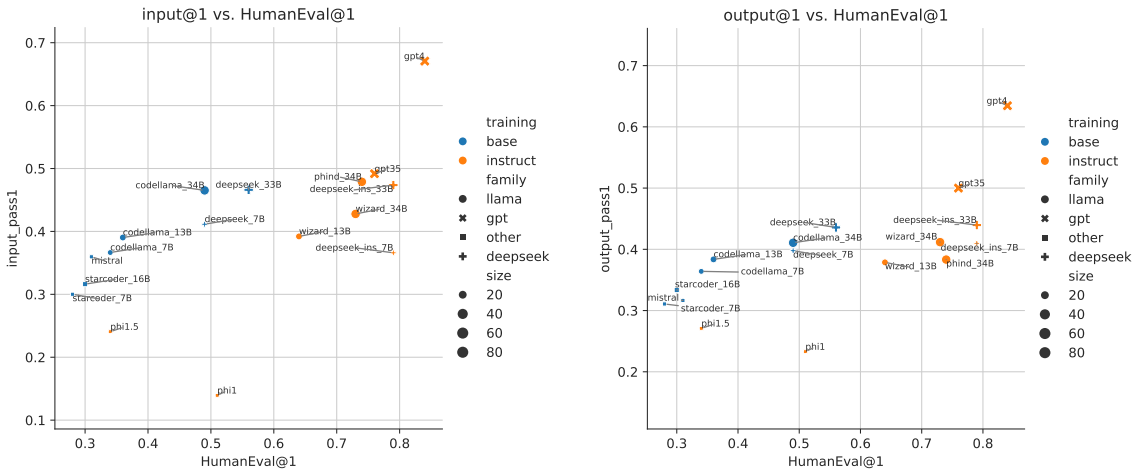
(b) CRUXEVAL-O Performance

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: After the release of Code Llama's model and GPT-3.5 and GPT-4's APIs, there have been many creative efforts to take data distilled from GPT models and use them to train more powerful code models such as WizardCoder (Luo et al., 2023), Phi-1 (Gunasekar et al., 2023), Phi-1.5 (Gunasekar et al., 2023), and Phind (Royzen et al., 2023). For example, WizardCoder 34B started with the Code Llama 34B base model and improved the HumanEval pass@1 score from 53.7% to 73.2%, a significant and impressive achievement. There remains curiosity about whether these models show more general improvements in other aspects of programming or code understanding (Gudibande et al., 2023). We measure this through CRUXEVAL.

In Fig. 3, we plot reported HumanEval scores (we did not reproduce them ourselves) against scores on CRUXEval. Indeed, we spot some interesting outliers: 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 both input and output predictions. In addition, the Phi-1 model outperforms most of the bigger models on HumanEval, but performs among the worst of all our evaluated models on CRUXEval. 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. On the other hand, for models such as StarCoder, Mistral, CodeLlama, and DeepSeek-Base, we still see a positive trend between HumanEval score and CRUXEval score, suggesting that code generation and execution/understanding abilities are correlated.



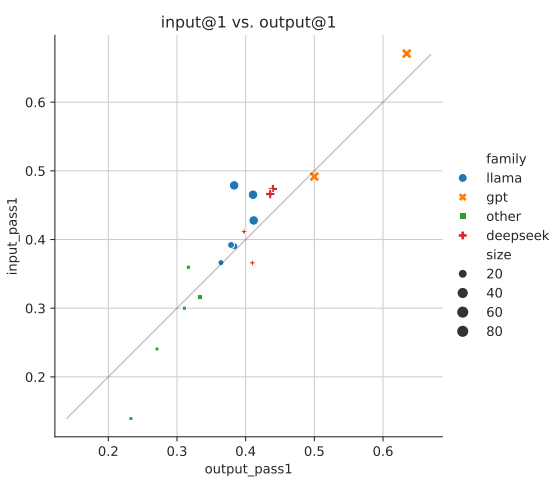
(a) CRUXEval-I vs. HumanEval

(b) CRUXEval-O vs. HumanEval

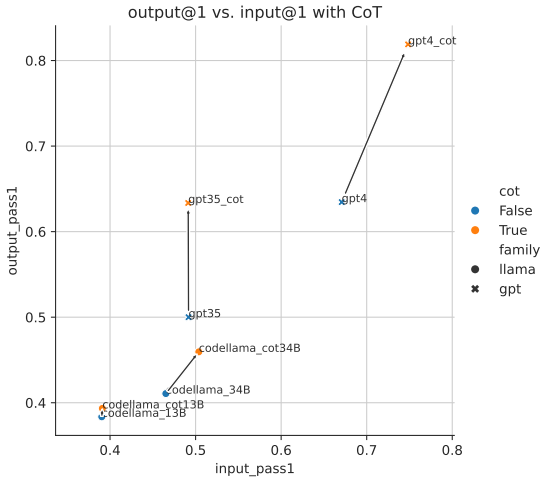
Figure 3: Correlation between HumanEval pass@1 scores and CRUXEval-O pass@1 scores

Base models show a weak correlation between HumanEval and CRUXEval. For HumanEval, distilled models (WizardCoder, Phind, Phi) significantly beat their base models, but for CRUXEval, no distilled model performs significantly better than Code Llama 34B.

Relationship between input prediction and output prediction: In Fig. 4a, we compare the input prediction and output prediction 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. This suggests the hypothesis that performance on relatively distinct coding-related tasks may be closely correlated. In addition, we see a relatively clear impact of scaling the model size on our two tasks.



(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

With the potential exception of GPT models, performance on CRUXEVAL-I and CRUXEVAL-O seem to be very correlated. As the tasks seem relatively different, this suggests that the code reasoning capabilities of models may generalize from task to task.

Confusion matrix/error correlation for different models. Fig. 5 shows the pairwise correlation of pass@1 scores for each pair of models. The correlation is chosen based on its highest signal among cosine distance, Spearman and Kendall. The middle section of "open"-ish models (StarCoder, Code Llama, DeepSeek, etc.) are strongly correlated with each other. Strong correlations are seen between sizes of the same model, between models of the same size, and between instruct and base (PHIND 34B, WIZARD 34B vs. CODE LLAMA 34B). CoT results also tend to have strong correlations with other CoT results, even GPT-4 vs Llama 13B. For the output task, DEEPSEEK forms a small sub-cluster of especially strong associations.

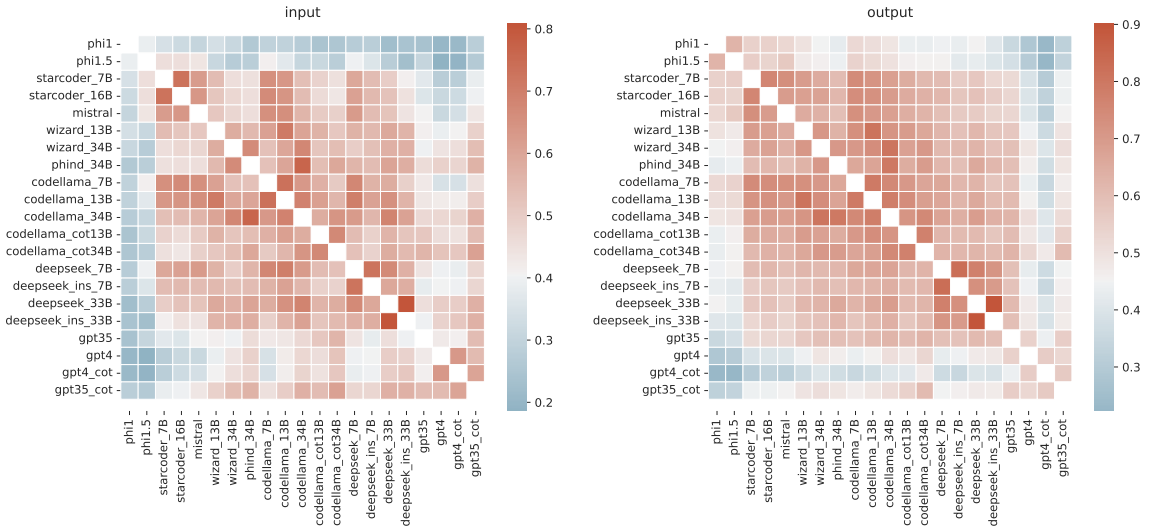


Figure 5: Correlation between predictions on input (left) and output (right)

Looking at model predictions, strong correlations are seen between sizes of the same model, between models of the same size, and between instruct and base models. 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.

5.1 Chain of Thought Prompting

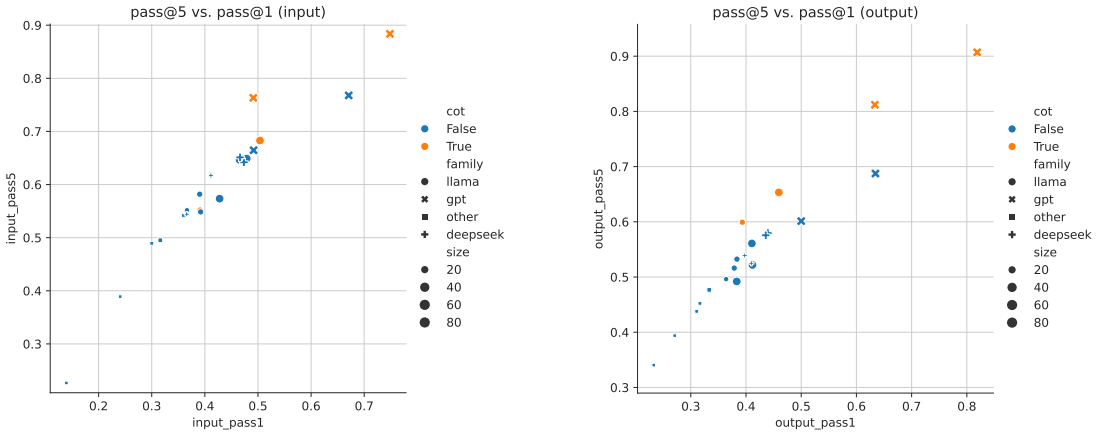
Next, we evaluate how the popular chain-of-thought (CoT) prompting method (Wei et al., 2022) affects the performance of Code Llama, GPT-3.5, and GPT-4 models on CRUXEVAL. The full prompts can be found in Appendix D.3. All results are reported using $N = 10$ samples other than CodeLlama 13B and 34B without 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 C.2.

Impact of CoT: We begin by focusing our attention on the pass@1 scores of models with and without CoT. In Fig. 4b, 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.

CoT helps Code Llama 34B and GPT-4 on both input and output prediction, GPT-3.5 on only output prediction, and Code Llama 13B on neither task. CoT also leads to larger boosts on output prediction than input prediction. GPT-4 benefits significantly more from CoT than other models, achieving the highest pass@1 of 74.8% on input prediction and 81.9% on output prediction but still far from acing the benchmark.

CoT widens the gap between pass@5 and pass@1 scores: In Fig. 6, we plot the pass@5 scores against the pass@1 scores for all models. For models without CoT (shown in blue), there is a positive correlation between pass@1 and pass@5 scores. For models with CoT (shown in orange), we see an increase in the gap between pass@5 and pass@1 scores. We believe this phenomenon may be due to the additional diversity induced by CoT, which we analyze in detail in Appendix C.3.

Because CoT increases the diversity of generated inputs and outputs, models with CoT see a larger gap between pass@1 and pass@5 score compared to models without.

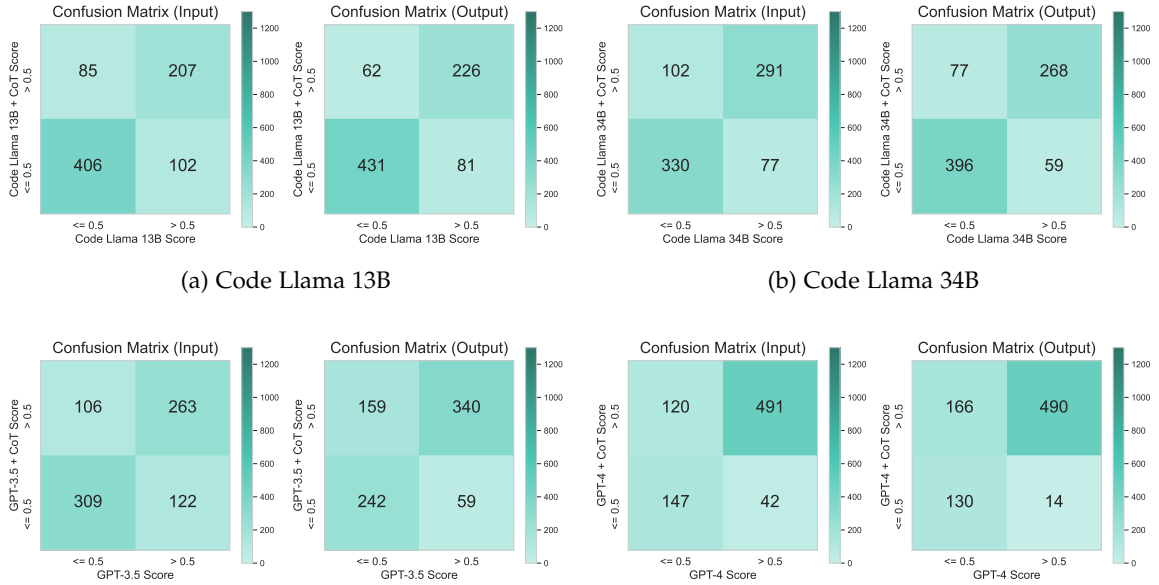


(a) Input prediction

(b) Output prediction

Figure 6: pass@5 score vs. pass@1 score with and without CoT

Predictions of CoT vs. Base Model: In Fig. 7, we show a confusion matrix over samples to better understand the correlations between direct output predictions and CoT predictions. For CodeLlama 13B, 34B, and GPT-3.5, we observe a large number of samples where direct prediction succeeds but CoT fails. However, with GPT-4, we observe that there are relatively few samples where this is the case.



(a) Code Llama 13B

(b) Code Llama 34B

(c) GPT-3.5

(d) GPT-4

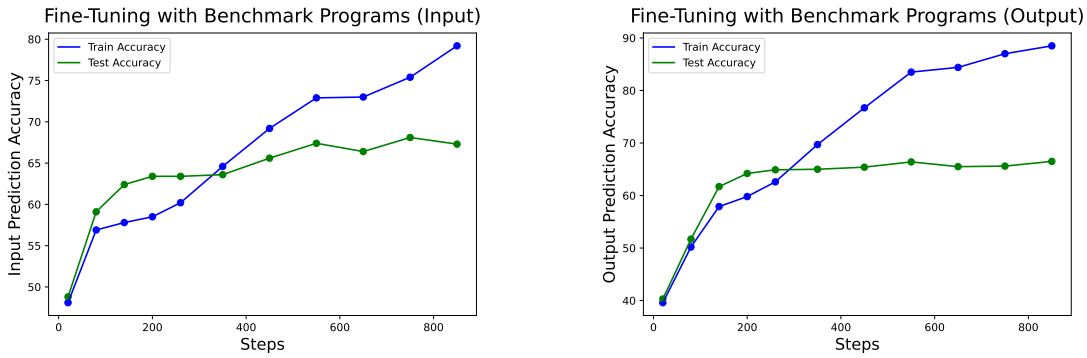
Figure 7: Confusion Matrix of Direct Prediction vs. CoT Prediction ($T = 0.2$)

While CoT generally improves performance overall, there are many individual samples where CoT actually hurts the prediction accuracy for Code Llama 13B/34B and GPT-3.5 on both input and output prediction. For GPT-4, CoT generally improves individual sample accuracy, more so for output prediction than for input prediction.

5.2 Fine-tuning Experiments

Next, we do a preliminary analysis to understand the effect of simple fine-tuning schemes on CRUXEval performance. We fine-tuned Code Llama 34B on nearly 140K samples of Python functions distilled with the procedure outlined in Sec. 3, without filtering. We perform weak decontamination, only removing samples where both the function and input-output pairs match samples in the benchmark.

In particular, we finetune on a mixture of 50% samples where the function is not in the benchmark and 50% samples where the function is in the benchmark but input-output pairs are not, a very liberal setup. The training and testing accuracy over time is shown in Fig. 8. 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. We defer a few other insights from fine-tuning to Appendix C.7 and suggest a few fine-tuning ideas for improving our benchmark in Sec. 7.



(a) Input prediction

(b) Output prediction

Figure 8: Improvements and Limits of CRUXEVAL Performance after Fine-Tuning

After fine-tuning on samples very similar to those in our benchmark, Code Llama 34B can match the performance of GPT-4 on both input and output prediction. However, accuracy plateaus at under 70% for both tasks, so simple finetuning is far from solving the benchmark.

6 Qualitative Analysis

All models except GPT4 has over 50% failure rate, showing they cannot do simple executions. In this section, we focus on GPT4 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 E for further examples of the failures highlighted in this section and 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 problem, 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 are indeed simple (Listings 3, 4). There are 2 problems that require counting to around 30 (Listing 5) and 2 problems (Listing 6) that require simulating a few actual steps, which might be difficult for direct generation but within scope for CoT.

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

Listing 3: GPT-4 has the right idea but cannot do the string concatenation correctly

```
def f(prefix, s):
    return str.removeprefix(prefix, s)
assert f('hymi', 'hymifulhxhzpnyihyf') == 'hymi'
# GPT4+CoT preds:
# input
# f('p', 'phymi')
# f('', 'hymi')
# output
# 'fulhxhzpnyihyf'
# 'fulhxhzpnyihyf'
```

Listing 4: GPT-4 might have been misled by the variable name prefix

```
def f(text, search_string):
    indexes = []
    while search_string in text:
        indexes.append(text.rindex(search_string)
    ↪ )
        text = text[:text.rindex(search_string)]
    return indexes
assert f('ONBPICJOHRHDJOSNCPNJ9ONTHBQCJ', 'J')
    ↪ == [28, 19, 12, 6]

# GPT4+CoT preds:
# input
# f("000012300001230000123000123", "123")
# f('bbbbbbabbbbbbabbbbbbabbbbbbab', 'a')
# f("abcxxxxxxabcxxxxxxabcxxxxxxabc","abc")
# output
# [23, 13, 8, 5]
# [25, 18, 15, 11, 6]
# [7, 10, 14, 18]
```

Listing 5: GPT-4 CoT failures where solutions requires counting to 30

```
def f(L):
    N = len(L)
    for k in range(1, N//2 + 1):
        i = k - 1
        j = N - k
        while i < j:
            # swap elements:
            L[i], L[j] = L[j], L[i]
            # update i, j:
            i += 1
            j -= 1
    return L
assert f([16, 14, 12, 7, 9, 11]) == [11, 14, 7,
    ↪ 12, 9, 16]
# GPT4+CoT preds:
# f([16, 9, 12, 7, 14, 11])
# f([16, 9, 7, 12, 14, 11])
# [11, 9, 7, 12, 14, 16]
# [11, 9, 7, 12, 14, 16]
```

Listing 6: GPT-4 CoT failure, cannot easily tell the answer without running the loops

Listings 7 and 8 show GPT-4 CoT failures for output prediction only. In Listing 8, 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 GPT4 CoT is consistently wrong. We manually tested scratchpad Nye et al. (2021) style prompts, which failed in the same way as regular CoT (Appendix E.3).

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

Listing 7: GPT-4 CoT output

```
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 8: GPT-4 CoT output

Failures of GPT-4, Input Prediction: Here are two simple failures on input prediction. Listings 9 and 10 show input prediction failures for concise and simple Python programs with and without CoT, respectively.

```
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'
```

Listing 9: GPT-4 CoT input

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

Listing 10: GPT-4 CoT input

Other GPT-4 Failures: Finally, we conclude with a set of six relatively simple string manipulation tasks that we discovered that GPT-4 fails on. We suspect the errors could be partially due to tokenization. The full GPT-4 outputs of these tasks can be found in Appendix E.5.

```
- What is a string containing 'a' three times, 'b' three times, 'c' twice, 'd' three times, and 'z'
    ↪ twice?
- In Python, what is " BaB ".rfind(" B ")?
- In Python, if I have a string s = 'iabnm~~~~~~~~~~', what is s[1::2]?
- In Python, what is "+".join(['*', '+', 'n', 'z', 'o', 'h'])?
- In Python, if text = "!123Leap and the net will appear" and res = 123, what is text[len(str(res)):]?
- What is "pomodoro".replace("or", "pomodoro")?
```

7 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 D.2 and D.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., 2023d) 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 E.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., 2023b; Madaan et al., 2023b; Peng et al., 2023; Zhang et al., 2023c; 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.

8 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 explore the potential of CoT (Sec. 5.1) and fine-tuning (Sec. 5.2) for improving performance. Fourth, we provide a qualitative analysis showcasing successes and failures of GPT-4 on our benchmark (Sec. 6 and Appendix E). Overall, we believe that CRUXEval provides a complimentary perspective to classical code LM