LARGE LANGUAGE MODELS ARE STATE-OF-THE-ART EVALUATORS OF CODE GENERATION

Terry Yue Zhuo
Monash University and CSIRO's Data61
terry.zhuo@monash.edu

ABSTRACT

Recent advancements in the field of natural language generation have facilitated the use of large language models to assess the quality of generated text. Although these models have shown promising results in tasks such as machine translation and summarization, their applicability in code generation tasks remains limited without human involvement. The complexity of programming concepts required for such tasks makes it difficult to develop evaluation metrics that align with human judgment. Token-matching-based metrics, such as BLEU, have demonstrated weak correlations with human practitioners in code generation tasks. Moreover, the utilization of human-written test suites to evaluate functional correctness can be challenging in domains with low resources. To overcome these obstacles, we propose a new evaluation framework based on the GPT-3.5 (`GPT-3.5-turbo`), for code generation assessments. Our framework addresses the limitations of existing approaches by achieving superior correlations with functional correctness and human preferences, without the need for test oracles or references. We evaluate the efficacy of our framework on two different aspects (**human preference** and **execution success**) and four programming languages, comparing its performance with the state-of-the-art CodeBERTScore metric, which relies on a pre-trained model. Our results demonstrate that our framework surpasses CodeBERTScore, delivering high levels of accuracy and consistency across various programming languages and tasks. We also make our evaluation framework and datasets available to the public at **`https://github.com/terryyz/llm-code-eval`**, encouraging further research in the evaluation of code generation.[1]

1 INTRODUCTION

Natural language generation (NLG) systems have seen significant progress with the development of large language models (LLMs). These models have shown great promise in generating high-quality and diverse texts that can be difficult to distinguish from human-written texts (Ouyang et al., 2022). However, evaluating the quality of NLG systems remains a challenging task, primarily due to the limitations of traditional evaluation metrics. Token-matching-based metrics, such as BLEU (Papineni et al., 2002) and ROUGE (Lin, 2004), have been widely used to evaluate NLG systems but have demonstrated poor correlation with human judgment and a lack of ability to capture semantic meanings (Kocmi et al., 2021). Furthermore, these metrics require reference output, which can be challenging to obtain for new tasks and low-resource domains (Liu et al., 2023).

In recent years, the use of LLMs as reference-free evaluators for Natural Language Generation (NLG) tasks has gained attention among researchers. This approach has demonstrated a strong alignment with human preferences, even when reference texts are not available (Liu et al., 2023; Fu et al., 2023). The underlying assumption behind this approach is that LLMs possess a profound understanding of human-generated text and task instructions, enabling them to evaluate various NLG tasks through prompts. The exceptional performance of LLMs in contextual understanding and natural language generation, as evidenced by studies (Brown et al., 2020), further supports this assumption. Moreover, LLMs trained on both textual and code-based data have showcased remarkable capabilities in diverse downstream tasks related to source code, including code generation (OpenAI,

[1]Confession: Prompt engineering is all you need, as of now.

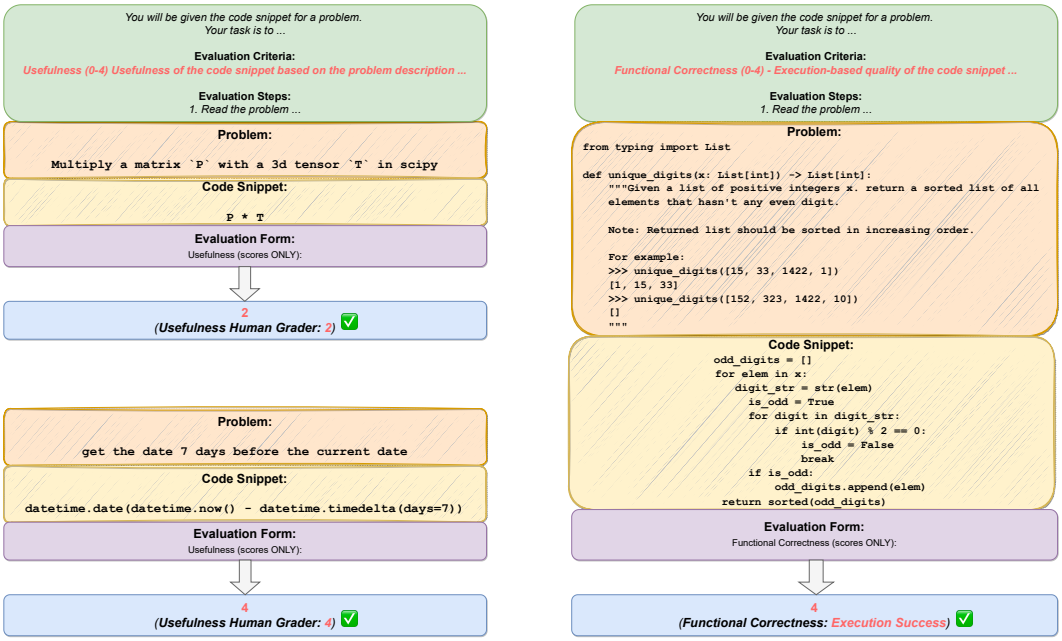

Figure 1: Example inputs and outputs of LLM-based evaluators of code generation. On the left-hand side, we assess the usefulness of the generated code snippet, from the human perspective. On the right-hand side, we evaluate the functional correctness of the generated code based on the execution.

2023; Allal et al., 2023). While a performance gap still exists between LLMs and human developers in code-related tasks, recent research has illustrated that LLMs can be enhanced to handle various source code tasks with appropriate guidance (Chen et al., 2023; Madaan et al., 2023). This indicates the significant potential of LLMs in comprehending and working with source code.

Code generation evaluation presents unique challenges, requiring a deeper understanding of programming concepts and more complex syntax than natural language generation (Hindle et al., 2016). Traditional reference-based evaluation metrics for code generation, such as BLEU Papineni et al. (2002), ROUGE (Lin, 2004), and chrF (Popović, 2015), rely on token matching to assess performance automatically. However, these metrics have demonstrated poor correlation with human evaluation (Evtikhiev et al., 2022) since they often underestimate the variety of outputs with the same semantic logic. While some studies have incorporated programming features to improve these metrics, they have shown limited gains and poor correlation with functional correctness (Eghbali & Pradel, 2022; Tran et al., 2019). Alternatively, researchers have proposed using well-designed test suites to objectively evaluate code generation performance at the function level (Chen et al., 2021; Zheng et al., 2023; Cassano et al., 2023). However, developing these test suites requires programming expertise, which can be impractical and costly in low-resource scenarios. Additionally, executing model-generated code poses a security risk and must be run in an isolated sandbox, which is technically cumbersome.

More recently, CodeBERTScore (Zhou et al., 2023), a neural-model-based evaluation metric, has been proposed, showing higher correlation with functional correctness and human preferences by capturing the semantic information of reference code and generated code. However, CodeBERTScore still relies on high-quality references that can be difficult and expensive to obtain. Moreover, the limited performance of the CodeBERT (Feng et al., 2020) backbone suggests that it has not yet reached human-level understanding of source code, limiting the effectiveness of CodeBERTScore. Therefore, more advanced evaluation frameworks are needed that can better capture the complex syntax and semantics of code generation tasks.

To address these challenges, we propose a novel evaluation framework based on LLMs trained on both text and code, shown in Figure 1. Our framework leverages the recent NLG framework, G-EVAL (Liu et al., 2023), but achieves superior correlations with subjective human preferences and

objective functional correctness, both at the example and corpus levels, without requiring specific designs of auto chain-of-thoughts and scoring functions.

We evaluate our proposed framework on four programming languages (Java, Python, C, C++, and JavaScript) from two aspects (**human-based usefulness** and **execution-based functional correctness**). Our evaluation approach does not require costly or impractical test suites and does not rely on high-quality references, making it more accessible and cost-effective for a wider range of applications. Additionally, our framework utilizes the powerful context understanding capabilities of LLMs like GPT-3.5, making it highly effective for evaluating code generation performance.

Based on our extensive evaluation, we have summarized our findings as follows:

- LLMs demonstrate profound understanding of source code and functionalities in different programming languages, suggesting that they can serve as powerful tools for evaluating code generation quality.
- The use of reference code can help LLMs assess the generated code. Without high-quality references, the performance of LLM-based evaluation is limited.
- The reliability of LLM-based evaluation can be further improved by using zero-shot-CoT (Kojima et al.), which allows LLMs to capture and reason about the underlying structure and logic of source code, enhancing their ability to evaluate code generation performance accurately and effectively.

Our experimental findings confirm the effectiveness of our proposed evaluation framework based on LLMs trained on both text and code. In addition to these findings, we also show that these findings are minimally affected by data contamination issues (Aiyappa et al., 2023; OpenAI, 2023; Magar & Schwartz, 2022). We further discuss the challenges associated with evaluating downstream tasks of source code and the potential for extending our evaluation paradigm to these tasks. We have publicly released our evaluation framework and datasets to encourage the development of more systematic criteria and open-source datasets of human judgment in these downstream tasks. Our hope is to foster the development of more accurate and effective evaluation metrics for tasks involving source code.

2 METHOD

Our prompt-based evaluation method, inspired by G-EVAL, consists of two main components: 1) task definition, evaluation criteria, and detailed evaluation steps, and 2) a given problem and generated code snippet for evaluation.

2.1 INSTRUCTIONS FOR CODE GENERATION EVALUATION

The evaluation of code quality involves two main aspects: 1) human judgment of code usefulness, and 2) execution-based functional correctness. To provide a comprehensive evaluation, we adopt the design of G-EVAL for the general task instruction, as follows:

> *You will be given the code snippet for a problem. Your task is to rate the code snippet only on one metric. Please make sure you read and understand these instructions carefully. Please keep this document open while reviewing, and refer to it as needed.*

Regarding the task-agnostic prompt, we have designed the following evaluation criteria and evaluation steps for assessing **code usefulness**. These criteria are aligned with previous human evaluations of code quality (Evtikhiev et al., 2022):

> *Evaluation Criteria:*
> *Usefulness (0-4) Usefulness of the code snippet based on the problem description.*
> *- A score of 0: Snippet is not at all helpful, it is irrelevant to the problem.*
> *- A score of 1: Snippet is slightly helpful, it contains information relevant to the problem, but it is easier to write the solution from scratch.*
> *- A score of 2: Snippet is somewhat helpful, it requires significant changes*

(compared to the size of the snippet), but is still useful.
- A score of 3: Snippet is helpful, but needs to be slightly changed to solve the problem.
- A score of 4: Snippet is very helpful, it solves the problem.

Evaluation Steps:
1. Read the problem carefully and identify required functionalities of the implementation.
2. Read the code snippet and compare it to the problem. Check if the code snippet covers all required functionalities of the problem, and if it presents them in a clear and logical order.
3. Assign a score for usefulness on a scale of 0 to 4, where 0 is the lowest and 4 is the highest based on the Evaluation Criteria.

To evaluate **functional correctness**, we emphasize the importance of considering unit tests during the evaluation process. We present the following criteria for evaluating functional correctness:

Evaluation Criteria:
Functional Correctness (0-4) - Execution-based quality of the code snippet combined with the problem. The correctness is measured by the all possible unit tests, and the comparison of the reference code. The combination of the code snippet and the problem should pass all the possible tests based on your understanding of the reference code. The length of the code snippet can not determine the correctness. You need to assess the logics line by line.
- A score of 0 (failing all possible test) means that the code snippet is totally incorrect and meaningless.
- A score of 4 (passing all possible test) means that the code snippet is totally correct and can handle all cases.

Evaluation Steps:
1. Read the problem carefully and identify required functionalities of the implementation.
2. Read the code snippet and compare it to the problem. Check if the code snippet covers all required functionalities of the problem.
3. Assign a score for functional correctness on a scale of 0 to 4, where 0 is the lowest and 4 is the highest based on the Evaluation Criteria.

2.2 Inputs of Code Generation Evaluation

It is worth noting that most code generative models do not take formatting into account, resulting in unformatted code that requires post-processing of code formatting to be understood, compiled, and executed (Zheng et al., 2023). Additionally, automatic evaluation metrics for code generation, such as CodeBLEU (Ren et al., 2020) and RUBY (Tran et al., 2019), still rely on language-specific program parsers [2]. However, based on prior findings that LLMs can robustly understand input data (Huang et al., 2022; Zhuo et al., 2023), we hypothesize that LLMs can also understand programming context without proper formatting. Therefore, for evaluation, we input the problems and generated code (and reference code, if provided). When reference code is provided, we slightly modify the evaluation steps in the prompt to incorporate it.

3 Experiment Setup

We evaluate the effectiveness of LLMs as evaluators of code generation using `GPT-3.5-turbo`[3] across multiple datasets and programming languages. We conduct two experiments to investigate

[2] `https://tree-sitter.github.io/`
[3] `https://platform.openai.com/docs/models/gpt-3-5`

the correlation between LLM-based evaluations and human preference and functional correctness, respectively. We compare the performance of LLM-based evaluations against seven automatic evaluation metrics, including the state-of-the-art CodeBERTScore (Zhou et al., 2023). To measure the correlation with human preference, we use the CoNaLa dataset (Yin et al., 2018) and corresponding human annotation on the generated code from various models trained on the dataset (Evtikhiev et al., 2022). To measure the correlation with functional correctness, we use the HumanEval-X dataset (Zheng et al., 2023). We do not consider **distinguishability** as an evaluation option, as prior work (Zhou et al., 2023) has shown it to be an unreliable meta-metric that cannot substitute for execution-based or human-based ratings.

3.1 Automatic Evaluation Metric Baselines

We compare our evaluation framework with 7 predominant automatic evaluation metrics. These metrics can be classified into two groups: **string-based** and **neural-model-based** evaluation.

String-based Evaluation Most evaluation metrics in code generation have been adapted from natural language generation (NLG) and rely on comparing the generated code to reference code. The most commonly used metric is BLEU (Papineni et al., 2002), which computes the overlaps of n-grams in the generated output with those in the reference, where the n-grams are tokenized using a language-specific tokenizer (Post, 2018). Other metrics include ROUGE-L (Lin, 2004), a recall-oriented metric that looks for the longest common subsequence between the reference and the generated code, and METEOR (Banerjee & Lavie, 2005), which is based on unigram matching between the generated code and the reference. However, studies have shown that BLEU may yield similar results for models with different quality levels from the perspective of human graders in code generation (Evtikhiev et al., 2022), leading to the proposal of new evaluation metrics such as RUBY (Tran et al., 2019). RUBY takes the code structure into account and compares the program dependency graphs (PDG) of the reference and the candidate. If the PDG is impossible to build, the metric falls back to comparing the abstract syntax tree (AST), and if the AST is also impossible to build, it compares the weighted string edit distance between the tokenized reference and candidate sequence. Another recent metric is CodeBLEU (Ren et al., 2020), which is a composite metric that computes a weighted average of four sub-metrics treating code differently: as a data-flow graph, as an abstract syntax tree, and as text. CodeBLEU is designed to evaluate the quality of generated code for code generation, code translation, and code refinement tasks.

Neural-model-based Evaluation Neural-model-based evaluation is becoming increasingly important for evaluating the quality of code generated by deep learning models. CodeBERTScore (Zhou et al., 2023) is one of the latest approaches that leverages pretrained code models like CodeBERT (Feng et al., 2020) and best practices from natural language generation evaluation to assess the quality of generated code. CodeBERTScore encodes the generated code and reference code independently and considers natural language context, contextual information of each token, and implementation diversity. It enables the comparison of code pairs that are lexically different and calculates precision and recall based on the best matching token vector pairs. This approach provides an effective way to evaluate the effectiveness of deep learning models for code generation tasks. Note that the authors of CodeBERTScore provided both F1 and F3 scores, with the optional source input. Therefore, we use these four language-specific variants of CodeBERTScore in our experiments.

3.2 Datasets and Evaluation Aspects

Human-based Usefulness Experiments Similar to Zhou et al. (2023), we conduct an evaluation on the CoNaLa benchmark (Yin et al., 2018), which is a widely used dataset for natural language context to Python code generation. To measure the correlation between each evaluation metric and human preference, we utilize the human annotations provided by Evtikhiev et al. (2022). Specifically, for each example in the dataset, experienced software developers were asked to grade the generated code snippets from five different models. The grading scale ranges from zero to four, with zero indicating that the generated code is irrelevant and unhelpful, and four indicating that the generated code solves the problem accurately. The dataset comprises a total of 2,860 annotated

Metric	Example			Corpus		
	τ	r_p	r_s	τ	r_p	r_s
BLEU	.439	.522	.488	.423	.572	.542
CodeBLEU	.292	.363	.331	.259	.397	.339
chrF	.458	.570	.515	.449	<u>.592</u>	.578
ROUGE-L	.447	.529	.499	.432	.581	.552
METEOR	.410	.507	.462	.415	.557	.534
RUBY	.331	.397	.371	.339	.493	.439
CodeBERTScore-F1 (w/o S.)	.499	.595	.558	.461	.579	.589
CodeBERTScore-F1 (w/ S.)	.500	<u>.609</u>	.556	<u>.464</u>	.579	<u>.595</u>
CodeBERTScore-F3 (w/o S.)	.485	.587	.542	.441	.556	.568
CodeBERTScore-F3 (w/ S.)	<u>.505</u>	<u>.609</u>	<u>.563</u>	.437	.549	.564
GPT-3.5 (w/o R.)	**.556**	.613	**.594**	**.546**	.649	**.635**
GPT-3.5 (w/ R.)	.554	**.617**	.591	.539	**.661**	.630

Table 1: Example-level and corpus-level Kendall-Tau (τ), Pearson (r_p) and Spearman (r_s) correlations with the human preferred usefulness on CoNaLa. **(w/o S.)**: without source inputs; **(w/ S.)**: with source inputs; **(w/o R.)**: without reference code inputs, or reference-free; **(w/o R.)**: with reference code inputs, or reference-enhanced. The best performance is **bold**. The second-best performance is <u>underlined</u>.

code snippets (5 generations $\times$ 472 examples) with each snippet being graded by 4.5 annotators on average.

Execution-based Functional Correctness Experiments We conduct an evaluation of functional correctness using the HumanEval benchmark (Chen et al., 2021), which provides natural language goals, input-output test cases, and reference solutions written by humans for each example. The benchmark originally consists of 164 coding problems in Python, and has been extended by Cassano et al. (2023) to 18 other programming languages, including Java, C++, Python, and JavaScript. We chose to evaluate our models on these languages, as they are among the most popular programming languages. The translated examples also include the predictions of `code-davinci-002` and their corresponding functional correctness scores. Inspired by Zhou et al. (2023), we obtain them from the HumanEval-X dataset (Zheng et al., 2023). As each problem has nearly 200 generated code samples on average, it would be computationally expensive to evaluate them all using LLMs. Therefore, we randomly select 20 samples from each problem, and collect all samples from problems where no more than 20 versions of code were generated.

Correlation Metrics To measure the correlation between each metric's scores and the references, we follow best practices in natural language evaluation and used Kendall-Tau (τ), Pearson (r_p), and Spearman (r_s) coefficients.[4]. To systematically study the efficacy of each automatic evaluation metric, we compute both example-level and corpus-level correlations. The example-level correlation is the average correlation of each problem example, while the corpus-level correlation is the correlation of all aggregated examples in the task.

4 RESULTS

Human-based Usefulness Table 1 shows the correlation between different metrics with human preference. We compare two variants of our evaluation approach, reference-free and reference-enhanced evaluations, with 10 baseline metrics. We find that GPT-3.5-based evaluators outperforms these metrics by a significant margin, regarding both example- and corpus-level correlations. Our observation is consistent with the work of CodeBERScore, where the variants of CodeBERScore mostly outperform the strong baselines like chrF and ROUGE-L. For example, reference-free GPT-3.5 evaluator achieves 0.556 and 0.546 measured by Spearman correlation on example level and corpus level, respectively. In contrast, prior evaluation metrics barely reach the score of 0.5. In

[4]We use the implementations from `https://scipy.org/`

Metric	Java		C++		Python		JavaScript		Average	
	τ	r_s	τ	r_s	τ	r_s	τ	r_s	τ	r_s
BLEU	.337	.401	.146	.174	.251	.297	.168	.199	.225	.268
CodeBLEU	.355	.421	.157	.187	.272	.323	.226	.267	.253	.299
chrF	.346	.413	.166	.198	.262	.312	.186	.220	.240	.286
ROUGE-L	.327	.389	.143	.171	.240	.284	.151	.179	.215	.256
METEOR	.358	.425	.174	.208	.276	.327	.195	.231	.251	.298
RUBY	.340	.401	.139	.165	.216	.255	.138	.163	.208	.246
CodeBERTScore-F1 (w/o S.)	.333	.398	.146	.175	.237	.283	.148	.176	.216	.258
CodeBERTScore-F1 (w/ S.)	.314	.375	.148	.177	.231	.276	.145	.172	.209	.250
CodeBERTScore-F3 (w/o S.)	.359	.429	.169	.202	.265	.316	.180	.214	.243	.290
CodeBERTScore-F3 (w/ S.)	.356	.426	.166	.198	.262	.312	.189	.226	.243	.291
GPT-3.5 (w/o R.)	**.427**	**.442**	**.320**	**.326**	.279	.282	.316	.321	**.336**	**.343**
GPT-3.5 (w/ R.)	.388	.404	.274	.282	**.318**	**.325**	**.340**	**.348**	.330	.340

Table 2: Example-level Kendall-Tau (τ) and Spearman (r_s) correlations with the execution-based functional correctness on HumanEval . **(w/o S.)**: without source inputs; **(w/ S.)**: with source inputs; **(w/o R.)**: without reference code inputs, or reference-free; **(w/o R.)**: with reference code inputs, or reference-enhanced. The best performance is **bold**. The second-best performance is underlined.

Metric	Java		C++		Python		JavaScript		Average	
	τ	r_s	τ	r_s	τ	r_s	τ	r_s	τ	r_s
BLEU	.267	.326	.225	.276	.281	.344	.220	.270	.248	.304
CodeBLEU	.293	.359	.212	.260	.303	.371	.315	.385	.281	.343
chrF	.290	.355	.266	.325	.328	.402	.279	.342	.291	.356
ROUGE-L	.280	.342	.234	.286	.296	.363	.216	.264	.256	.314
METEOR	.318	.389	.260	.319	.349	.427	.311	.380	.309	.379
RUBY	.276	.337	.219	.268	.279	.341	.219	.268	.248	.303
CodeBERTScore-F1 (w/o S.)	.299	.367	.266	.326	.322	.394	.248	.303	.284	.348
CodeBERTScore-F1 (w/ S.)	.244	.298	.219	.268	.264	.324	.214	.262	.235	.288
CodeBERTScore-F3 (w/o S.)	.326	.399	.283	.347	.360	.441	.296	.363	.316	.387
CodeBERTScore-F3 (w/ S.)	.281	.344	.243	.297	.313	.384	.261	.320	.275	.336
GPT-3.5 (w/o R.)	.330	.345	.313	.321	.294	.298	.315	.323	.313	.322
GPT-3.5 (w/ R.)	**.412**	**.438**	**.367**	**.383**	**.425**	**.446**	**.432**	**.455**	**.409**	**.431**

Table 3: Corpus-level Kendall-Tau (τ) and Spearman (r_s) correlations with the execution-based functional correctness on HumanEval. **(w/o S.)**: without source inputs; **(w/ S.)**: with source inputs; **(w/o R.)**: without reference code inputs, or reference-free; **(w/o R.)**: with reference code inputs, or reference-enhanced. The best performance is **bold**. The second-best performance is underlined.

addition, we find that reference-enhanced evaluator does not significantly improve the performance, indicating the reference-code may not be optimized. Our further analysis on human rating of reference code of CoNaLa complies this implication, where the average score of reference code only achieves 3.4 out of 4, suggesting that not all human practitioners consider the reference fully useful.

Execution-based Functional Correctness Table 2 and Table 3 present the results of example- and corpus-level functional correctness, respectively. From Table 2, we observe that both reference-free and reference-enhanced evaluators consistently outperform the other baselines across all four programming languages on the example level. The reference-free GPT-3.5 evaluator even outperforms the reference-enhanced one, which suggests the potential bias in some reference code. Additionally, we find that METEOR and CodeBLEU receive better correlations than all variants of CodeBERTScore, indicating that they are still strong baselines compared to the recent neural-model-based evaluators in code generation. In Table 3, we observe that our reference-enhanced evaluator achieves the state-of-the-art performance among all evaluation metrics. When compared to other baselines, the reference-free GPT-3.5 evaluator still achieves comparable results to the source-free CodeBERTScore-F3.

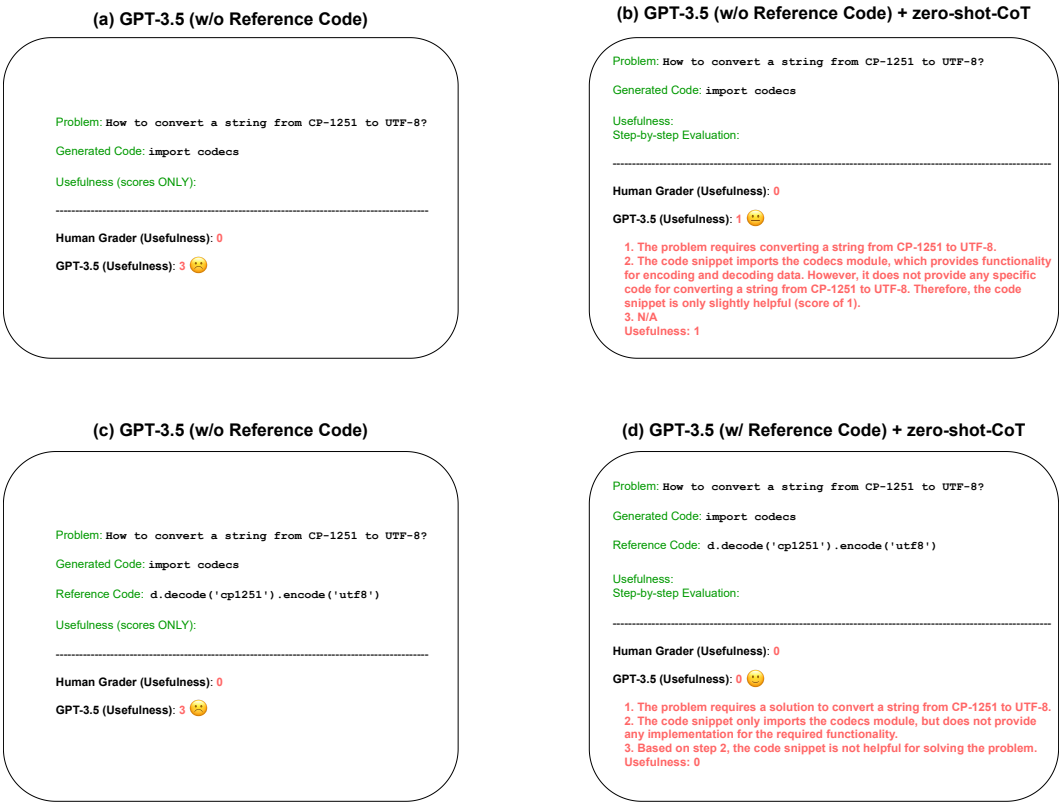

Figure 2: Example inputs and outputs with (a) reference-free GPT-3.5-based evaluator, (b) reference-free GPT-3.5-based evaluator with zero-shot-CoT evaluation, (a) reference-enhanced GPT-3.5-based evaluator, (b) reference-enhanced GPT-3.5-based evaluator with zero-shot-CoT evaluation. With the step-by-step evaluation, the output of our evaluator is more aligned with the human preference.

5 Ablation Study

Improving the Reliability of LLM-based Code Generation Evaluators Prior work (Wei et al.; Kojima et al.) has demonstrated that the performance of LLMs can be significantly improved via Chain-of-Thought (CoT) and zero-shot-Chain-of-Thought (zero-shot-CoT), where the prompts instruct LLMs to perform the task in a step-by-step manner. Here, we explore the zero-shot reasoning ability of LLMs in the evaluation of code generation. Specifically, we instruct GPT-3.5 to perform CoT-evaluation by adding "Step-by-step Evaluation:" at the end of the prompt. An example of the zero-shot-CoT prompt is shown in Figure 2. Instead of using LLMs to extract the evaluation score from the reasoning steps, like the original framework of zero-shot-CoT via multiple queries, we design a rule-based parser to extract scores. Due to limited resources, we only evaluate on CoNaLa in Table 4. Our results show that zero-shot-CoT can significantly improve the reliability of code generation evaluation. Additionally, we find that reference-enhanced evaluators can achieve better results than reference-free ones via zero-shot-CoT, even though their performances are similar without CoT processing. This suggests that LLMs can exploit the use of reference code through reasoning.

6 Discussion

Data Contamination Evaluations on recent closed-source LLMs have been criticized for the possibility of data contamination (Aiyappa et al., 2023), where the model may have already seen the evaluation datasets during training, due to the opaque training details of these models. For instance, Kocmi & Federmann (2023) conducted an empirical study on a few closed-source LLMs, including GPT-3.5, and suggested that LLMs are the state-of-the-art evaluators of translation quality, based on the evaluation of the WMT22 Metric Shared Task (Freitag et al., 2022). However, as most of the

Metric	Example			Corpus		
	τ	r_p	r_s	τ	r_p	r_s
GPT-3.5 (w/o R.)	.556	.613	.594	.546	.649	.635
+ zero-shot-CoT	**.561**	**.628**	**.600**	**.579**	**.703**	**.665**
GPT-3.5 (w/ R.)	.554	.617	.591	.539	.661	.630
+ zero-shot-CoT	**.571**	**.639**	**.607**	**.583**	**.712**	**.667**

Table 4: Example-level and corpus-level Kendall-Tau (τ), Pearson (r_p) and Spearman (r_s) correlations with the human preferred usefulness on CoNaLa. **(w/o R.)**: without reference code inputs, or reference-free; **(w/o R.)**: with reference code inputs, or reference-enhanced. The best performance is **bold**.

Dataset	Release Year	Likely to be contaminated?
CoNaLa	2018	✓
human-annotated CoNaLa w/ generated code	2023	✗
HumanEval (Python)	2021	✓
HumanEval-X (w/o Python)	2023	✗
human-annotated HumanEval-X w/ generated code	2023	✗

Table 5: Dataset, Release Year and the likelihood of data contamination for each dataset used in our study.

evaluated models were trained on data prior to 2022[5], it is highly likely that these models have been trained with some human-rated translation quality data. Similarly, G-EVLA (Liu et al., 2023) shows that GPT-3.5 and GPT-4 are the state-of-the-art evaluators of natural language generation (NLG) with the evaluation of three NLG datasets. However, as these human-annotated datasets were released before 2021, it is probable that they were included in the training data of GPT-3.5 and GPT-4. In contrast, our work is minimally impacted by data contamination, as we report the data release year in Table 5. Our analysis suggests that only CoNaL and HumanEval (Python) datasets may have been contaminated, and it is unlikely that GPT-3.5 has seen any human annotation or generated code during training.

LLM-based Evaluation Beyond Code Generation Recent studies have shown that large language models (LLMs) such as GPT-3.5 can achieve state-of-the-art performance in evaluating the functional correctness and usefulness of generated source code. However, the question remains as to whether LLMs can be utilized to evaluate downstream tasks related to source code beyond code generation. Allamanis et al. (2018) have identified several downstream applications such as code translation, commit message generation, and code summarization. While some studies have investigated the human evaluation of these tasks, none of them have released the annotation data or fully described the human evaluation criteria. This presents a challenge for extending the LLM-based evaluation framework to these tasks, although we believe it has great potential. For example, Hu et al. (2022) proposed a human evaluation metric for code documentation generation quality, which is specifically designed for code comment generation and commit message generation. Their metric includes three aspects: *Language-related*, *Content-related*, and *Effectiveness-related*, with detailed task descriptions and explanations of assigned scores. We propose that the information provided in their metric can be used to create prompts for LLM-based evaluation and enable human-aligned evaluation of code documentation generation.

7 Conclusion

In this paper, we propose a novel evaluation framework based on large language models trained on both text and code, which can better capture the complex syntax and semantics of code generation tasks. Our framework achieves superior correlations with subjective human preferences and objective functional correctness, both at the example and corpus levels, without reference and test suites.

[5] https://platform.openai.com/docs/model-index-for-researchers

We conduct an extensive evaluation on four programming languages (Java, Python, C, C++, and JavaScript) and demonstrate the effectiveness of our proposed method on human-based usefulness and execution-based functional correctness. We have publicly released our evaluation framework and datasets to encourage the development of more accurate and effective evaluation metrics for tasks involving source code.

REFERENCES

Rachith Aiyappa, Jisun An, Haewoon Kwak, and Yong-Yeol Ahn. Can we trust the evaluation on chatgpt? arXiv preprint arXiv:2303.12767, 2023.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! arXiv preprint arXiv:2301.03988, 2023.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR), 51(4):1–37, 2018.

Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization, pp. 65–72, 2005.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901, 2020.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. IEEE Transactions of Software Engineering (TSE), 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. arXiv preprint arXiv:2304.05128, 2023.

Aryaz Eghbali and Michael Pradel. Crystalbleu: precisely and efficiently measuring the similarity of code. In 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–12, 2022.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models?, 2022.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In Findings of the Association for Computational Linguistics: EMNLP 2020, pp. 1536–1547, 2020.

Markus Freitag, Ricardo Rei, Nitika Mathur, Chi-kiu Lo, Craig Stewart, Eleftherios Avramidis, Tom Kocmi, George Foster, Alon Lavie, and André FT Martins. Results of wmt22 metrics shared task: Stop using bleu–neural metrics are better and more robust. In Proceedings of the Seventh Conference on Machine Translation (WMT), pp. 46–68, 2022.

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. Gptscore: Evaluate as you desire. arXiv preprint arXiv:2302.04166, 2023.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. Communications of the ACM, 59(5):122–131, 2016.

Xing Hu, Qiuyuan Chen, Haoye Wang, Xin Xia, David Lo, and Thomas Zimmermann. Correlating automated and human evaluation of code documentation generation quality. ACM Transactions on Software Engineering and Methodology (TOSEM), 31(4):1–28, 2022.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve. arXiv preprint arXiv:2210.11610, 2022.

Tom Kocmi and Christian Federmann. Large language models are state-of-the-art evaluators of translation quality, 2023.

Tom Kocmi, Christian Federmann, Roman Grundkiewicz, Marcin Junczys-Dowmunt, Hitokazu Matsushita, and Arul Menezes. To ship or not to ship: An extensive evaluation of automatic metrics for machine translation. In Proceedings of the Sixth Conference on Machine Translation, pp. 478–494, Online, November 2021. Association for Computational Linguistics. URL https://aclanthology.org/2021.wmt-1.57.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In Advances in Neural Information Processing Systems.

Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In Text summarization branches out, pp. 74–81, 2004.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. Gpteval: Nlg evaluation using gpt-4 with better human alignment. arXiv preprint arXiv:2303.16634, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. arXiv preprint arXiv:2303.17651, 2023.

Inbal Magar and Roy Schwartz. Data contamination: From memorization to exploitation. arXiv preprint arXiv:2203.08242, 2022.

OpenAI. Gpt-4 technical report, 2023.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. Advances in Neural Information Processing Systems, 35: 27730–27744, 2022.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting of the Association for Computational Linguistics, pp. 311–318, 2002.

Maja Popović. chrf: character n-gram f-score for automatic mt evaluation. In Proceedings of the tenth workshop on statistical machine translation, pp. 392–395, 2015.

Matt Post. A call for clarity in reporting BLEU scores. In Proceedings of the Third Conference on Machine Translation: Research Papers, pp. 186–191, Brussels, Belgium, October 2018. Association for Computational Linguistics. doi: 10.18653/v1/W18-6319. URL https://aclanthology.org/W18-6319.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297, 2020.

Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. Does bleu score work for code migration? In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pp. 165–176. IEEE, 2019.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed H Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. In Advances in Neural Information Processing Systems.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In International Conference on Mining Software Repositories, MSR, pp. 476–486. ACM, 2018. doi: https://doi.org/10.1145/3196398.3196408.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. arXiv preprint arXiv:2303.17568, 2023.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code. arXiv preprint arXiv:2302.05527, 2023.

Terry Yue Zhuo, Yujin Huang, Chunyang Chen, and Zhenchang Xing. Exploring ai ethics of chatgpt: A diagnostic analysis. arXiv preprint arXiv:2301.12867, 2023.

A Automatic Evaluation Metric Baselines

Our implementations of the automatic evaluation metric baselines except for CodeBERTScore are based on `https://github.com/JetBrains-Research/codegen-metrics`. For CodeBERTScore, we adopt the official release at `https://github.com/neulab/code-bert-score`.

B Correlation Metrics

For all correlation metrics, we use the implementation from `https://scipy.org/` and call these APIs with the default settings.

C Rule-based Score Extraction from Zero-shot Chain Of Thought Evaluation

We demonstrate the general implementation of score extraction:

```python
import re
TASK_KEY_WORD = "usefulness" # or "functional"
def get_gpt_answer(raw_content):
    try:
        return int(raw_content)
    except:
        try:
            return process_raw_content(raw_content)
        except:
            return 0

def process_raw_content(content):
    # Clean up and split the content
    splits = content.lower().replace("(", "").replace(")", "").split("\n")

    # Extract relevant lines and clean them up
    ls = [ll.strip(".")
        .replace("out of ", "/")
        .replace("/4", "")
        for l in splits
        for ll in l.lstrip("0123456789. ").split(". ")
        if TASK_KEY_WORD in ll or "score" in ll]

    # Extract the scores
    ans = [ll for l in ls for ll in l.split() if ll.isnumeric()]

    # If there are multiple scores, take the most common one
    if len(set(ans)) != 1 and len(ans) > 1:
        return int(Counter(ans).most_common(1)[0][0])

    # If there are no scores or ambiguous scores, return 0 or 1
    if len(set(ans)) != 1:
        if "N/A" in content:
            return 0
        else:
            return 1

    # Otherwise, return the single score
    return int(ans[0])
```

Code Listing 1: Score Extractor Implementation

We note that our extraction process for the evaluation metrics is entirely rule-based and may not be optimized for the best results.

D EXAMPLES ON CONALA

D.1 PROBLEM-0

```
send a signal `signal.SIGUSR1` to the current process
```

Reference Code:

```
os.kill(os.getpid(), signal.SIGUSR1)
```

Code Snippet-0

```
os.system('<unk>.png',s = 300)
```

Human Grader (Usefulness 0-4): 0

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 0
GPT-3.5 (w/ R.) : 0
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

> *1. The problem requires sending a signal `signal.SIGUSR1` to the current process.*
> *2. The code snippet is completely unrelated to the problem. It is attempting to execute a system command with an unknown argument and a size parameter. There is no mention of signals or the current process.*
> *3. The code snippet is not helpful at all in solving the problem. It is completely irrelevant and does not provide any guidance or functionality related to sending signals.*
>
> *Usefulness Score: 0*

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

> *1. The problem requires sending a signal `signal.SIGUSR1` to the current process.*
> *2. The code snippet is completely unrelated to the problem. It is attempting to execute a system command with an unknown argument and a size parameter. There is no mention of signals or the current process.*
> *3. Based on step 2, the code snippet is not helpful at all in solving the problem. Therefore, the usefulness score is 0.*
>
> *Usefulness: 0*

Code Snippet-1

```
sys.signal(`signal.SIGUSR1`)
```

Human Grader (Usefulness 0-4): 4

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 3
GPT-3.5 (w/o R.) + zero-shot-CoT : 4

> *1. The problem requires sending a signal `signal.SIGUSR1` to the current process.*

2. The code snippet uses the ‘sys.signal()‘ function to send a signal ‘signal.SIGUSR1‘.
3. The code snippet covers the required functionality of the problem in a clear and concise manner.

Usefulness Score: 4

GPT-3.5 (w/ R.) + zero-shot-CoT : 4

1. The problem requires sending a signal ‘signal.SIGUSR1‘ to the current process.
2. The code snippet uses the ‘sys.signal()‘ function to send the signal ‘signal.SIGUSR1‘.
3. The code snippet covers the required functionality of the problem in a clear and concise manner.
4. Based on the Evaluation Criteria, the code snippet receives a score of 4 for usefulness as it solves the problem.

Usefulness: 4

Code Snippet-2

```
os.system(‘signal.SIGUSR1‘)
```

Human Grader (Usefulness 0-4): 1

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 1
GPT-3.5 (w/ R.) : 1
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

1. The problem requires sending a signal ‘signal.SIGUSR1‘ to the current process.
2. The code snippet uses the ‘os.system()‘ function to send the signal ‘signal.SIGUSR1‘. However, the argument passed to ‘os.system()‘ is incorrect. ‘signal.SIGUSR1‘ should be passed as an integer value, not a string. Additionally, the code snippet does not import the ‘signal‘ module, which is necessary to use ‘signal.SIGUSR1‘.
3. The code snippet is not helpful as it contains errors and does not provide a clear solution to the problem.
4. Usefulness score: 0.

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

1. The problem requires sending a signal ‘signal.SIGUSR1‘ to the current process.
2. The code snippet uses the ‘os.system()‘ function to send the signal ‘signal.SIGUSR1‘. However, the argument passed to ‘os.system()‘ is incorrect as it is enclosed in backticks instead of quotes. Additionally, the argument should be the command to send the signal, not the signal itself. Therefore, the code snippet is not useful for solving the problem.
3. Score: 0

Final Evaluation:
Usefulness: 0

Code Snippet-3

```
os.system('< unk > < unk > < unk >')
```

Human Grader (Usefulness 0-4): 0

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 2
GPT-3.5 (w/ R.) : 1
GPT-3.5 (w/o R.) + zero-shot-CoT : 1

> *1. The problem requires sending a signal 'signal.SIGUSR1' to the current process.*
> *2. The code snippet uses the 'os.system()' function, but the arguments are unknown.*
> *3. Based on the information provided, it is not clear if the code snippet covers all required functionalities of the problem, and if it presents them in a clear and logical order.*
>
> *Usefulness Score: 1*

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

> *1. The problem requires sending a signal 'signal.SIGUSR1' to the current process.*
> *2. The code snippet uses the 'os.system()' function, but the arguments are unknown ('< unk > < unk > < unk >'). It is unclear if the code snippet is relevant to the problem or not.*
> *3. Based on the lack of information in the code snippet, it is not possible to determine if it is useful or not. Therefore, the usefulness score is 0.*
>
> *Usefulness: 0*

Code Snippet-4

```
os.kill(os.getpid(), signal.SIGUSR1)
```

Human Grader (Usefulness 0-4): 3

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 4
GPT-3.5 (w/o R.) + zero-shot-CoT : 4

> *1. The problem requires sending a signal 'signal.SIGUSR1' to the current process.*
> *2. The code snippet uses the 'os' and 'signal' modules to send the required signal to the current process using the 'os.kill()' function.*
> *3. The code snippet covers all required functionalities of the problem and presents them in a clear and logical order.*
> *4. Based on the Evaluation Criteria, the code snippet is very helpful and solves the problem. Therefore, the usefulness score is 4.*
>
> *Usefulness: 4*

GPT-3.5 (w/ R.) + zero-shot-CoT : 4

1. The problem requires sending a signal 'signal.SIGUSR1' to the current process.
2. The code snippet uses the 'os' module to get the current process ID and then sends the 'signal.SIGUSR1' signal to it using the 'os.kill()' function. It covers all the required functionalities of the problem and presents them in a clear and logical order.
3. Based on the Evaluation Criteria, the code snippet is very helpful and solves the problem. Therefore, the usefulness score is 4.

Usefulness: 4

D.2 PROBLEM-1

```
decode a hex string '4a4b4c' to UTF-8.
```

Reference Code:

```
bytes.fromhex('4a4b4c').decode('utf-8')
```

Code Snippet-0

```
"""\\ascii.decode('utf-8').decode('utf-8')
```

Human Grader (Usefulness 0-4): 1

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 1
GPT-3.5 (w/ R.) : 0
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

1. The problem requires decoding a hex string to UTF-8.
2. The code snippet seems to be attempting to decode an ASCII string to UTF-8 twice, which is not relevant to the problem.
3. Based on the above evaluation, the code snippet is not helpful in solving the problem.

Usefulness Score: 0

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

1. The problem requires decoding a hex string to UTF-8.
2. The code snippet seems to be attempting to decode an ASCII string to UTF-8, but it is not clear how it relates to the problem. It also contains a syntax error (missing string before the method call).
3. Score: 0. The code snippet is not helpful for solving the problem and is irrelevant to the problem description.

Code Snippet-1

```
print(4a4b4c.decode(`utf-8`).decode(`utf-8`))
```

Human Grader (Usefulness 0-4): 1

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 0

GPT-3.5 (w/ R.) : 0
GPT-3.5 (w/o R.) + zero-shot-CoT : 2

> *1. The problem requires decoding a hex string to UTF-8.*
> *2. The code snippet attempts to decode a hex string to UTF-8, but it contains syntax errors and unnecessary decoding.*
> *3. Based on step 2, the code snippet needs significant changes to solve the problem.*
> *4. Therefore, the usefulness score for this code snippet is 2.*

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

> *1. The problem requires decoding a hex string to UTF-8.*
> *2. The code snippet attempts to decode the hex string using the ‘decode‘ method twice, which is not necessary and will result in a ‘SyntaxError‘ due to the use of invalid syntax in the ‘print‘ statement.*
> *3. Score: 0. The code snippet is not helpful as it contains invalid syntax and does not correctly decode the hex string to UTF-8.*

Code Snippet-2

```
print(4a4b4c.decode(‘4a4b4c‘,‘ignore‘))
```

Human Grader (Usefulness 0-4): 1

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 1
GPT-3.5 (w/ R.) : 0
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

> *1. The problem requires decoding a hex string to UTF-8.*
> *2. The code snippet attempts to decode a hex string using the ‘decode()‘ method, but it is not valid Python code as ‘4a4b4c‘ is not a valid string or bytes object. Additionally, the second argument of ‘ignore‘ is not a valid encoding type.*
> *3. Based on the above evaluation, the code snippet is not helpful in solving the problem and receives a usefulness score of 0.*
>
> *Usefulness: 0*

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

> *1. The problem requires decoding a hex string to UTF-8.*
> *2. The code snippet attempts to decode the hex string using the decode() method, but it contains syntax errors. It is unclear what the first argument (‘4a4b4c‘) represents and the second argument (‘ignore‘) is not a valid encoding. Therefore, the code snippet does not cover all required functionalities of the problem and does not present them in a clear and logical order.*
> *3. Score: 0. The code snippet is not at all helpful and is irrelevant to the problem.*

Code Snippet-3

```
print(4a4b4c.decode(‘4a4b4c‘))
```

Human Grader (Usefulness 0-4): 1

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 0
GPT-3.5 (w/ R.) : 0
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

> *1. The problem requires decoding a hex string to UTF-8.*
> *2. The code snippet attempts to decode a hex string, but it is not valid Python code. The ‘print‘ statement is also unnecessary and does not contribute to the decoding process.*
> *3. The code snippet is not helpful in solving the problem as it contains syntax errors and does not provide a valid solution.*
>
> *Usefulness Score: 0*

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

> *1. The problem requires decoding a hex string to UTF-8.*
> *2. The code snippet attempts to decode the hex string using the ‘decode()‘ method, but it is not clear what the argument ‘4a4b4c‘ represents. It is also not clear what the backticks around ‘4a4b4c‘ are for. Overall, the code snippet is incomplete and unclear.*
> *3. Based on the above evaluation, the usefulness score for this code snippet is 0.*

Code Snippet-4

```
'4a4b4c'.decode('utf-8')
```

Human Grader (Usefulness 0-4): 4

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 4
GPT-3.5 (w/o R.) + zero-shot-CoT : 4

> *1. The problem requires decoding a hex string to UTF-8.*
> *2. The code snippet uses the decode() method with 'utf-8' as the argument to decode the hex string.*
> *3. The code snippet covers the required functionality of the problem in a clear and concise manner.*
> *4. Therefore, the code snippet is very helpful and receives a usefulness score of 4.*
>
> *Usefulness: 4*

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

> *1. The problem requires decoding a hex string to UTF-8.*
> *2. The code snippet uses the decode() method to decode a string to UTF-8. However, it does not convert the hex string to a byte string first, which is necessary for decoding. Therefore, the code snippet is not useful for solving the problem.*
> *3. Usefulness score: 0.*

D.3 PROBLEM-2

```
check if all elements in list `myList` are identical
```

Reference Code:

```
all(x == myList[0] for x in myList)
```

Code Snippet-0

```
[int(i)for i in range(100)]
```

Human Grader (Usefulness 0-4): 0

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 1
GPT-3.5 (w/ R.) : 1
GPT-3.5 (w/o R.) + zero-shot-CoT : 1

> *1. The problem requires checking if all elements in a list are identical.*
> *2. The code snippet generates a list of integers from 0 to 99 using a list comprehension. It does not check if all elements in the list are identical.*
> *3. Usefulness score: 1. The code snippet is slightly helpful as it generates a list, but it does not address the problem of checking if all elements in a list are identical.*

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

> *1. The problem requires checking if all elements in a list are identical.*
> *2. The code snippet generates a list of integers from 0 to 99 using a list comprehension. It does not check if all elements in a list are identical, nor does it provide any functionality related to the problem.*
> *3. Usefulness score: 0. The code snippet is not helpful in solving the problem.*

Code Snippet-1

```
all(isinstance(list,my List)for x in my List)
```

Human Grader (Usefulness 0-4): 0

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 1
GPT-3.5 (w/ R.) : 1
GPT-3.5 (w/o R.) + zero-shot-CoT : 1

> *1. The problem requires checking if all elements in a list are identical.*
> *2. The code snippet uses the 'isinstance()' function to check if all elements in the list are of the same type as the 'myList' variable. However, it does not check if all elements are identical. Additionally, there is a syntax error in the code ('my List' should be 'myList').*
> *3. Based on the above evaluation, the usefulness score for this code snippet is 1. It contains information relevant to the problem, but it is easier to write the solution from scratch.*

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

> *1. The problem requires checking if all elements in a list are identical.*
> *2. The code snippet uses the 'isinstance()' function to check if all elements in the list are of the same type as the list itself. This is not the same as checking if all elements are identical. Additionally, there is a syntax error in the code ('my List'*

should be 'myList'.
3. Based on the above evaluation, the usefulness score for this code snippet is 0. It is not helpful for solving the problem.

Code Snippet-2

```
[i for i in my List if i = = i]
```

Human Grader (Usefulness 0-4): 1

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 3
GPT-3.5 (w/o R.) + zero-shot-CoT : 1

1. The problem requires checking if all elements in a list are identical.
2. The code snippet uses list comprehension to iterate through the list and check if each element is equal to itself. However, it does not return any output or provide a clear indication of whether all elements are identical or not.
3. Based on the above evaluation, the code snippet is only slightly helpful (score of 1) as it contains some relevant information but does not provide a clear solution to the problem.

Usefulness: 1

GPT-3.5 (w/ R.) + zero-shot-CoT : 4

1. The problem requires checking if all elements in a list are identical.
2. The code snippet uses a list comprehension to iterate through the elements in 'myList' and returns a new list with only the elements that are equal to themselves. This essentially checks if all elements in 'myList' are identical.
3. Based on the above evaluation, the code snippet is very helpful and solves the problem. Therefore, the usefulness score is 4.

Usefulness: 4

Code Snippet-3

```
[i for i in my List if i = = 2]
```

Human Grader (Usefulness 0-4): 3

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 1
GPT-3.5 (w/ R.) : 1
GPT-3.5 (w/o R.) + zero-shot-CoT : 1

1. The problem requires checking if all elements in a list are identical.
2. The code snippet filters the list 'myList' to only include elements that are equal to 2. This is not directly related to the problem, as it only checks for a specific value (2) and not if all elements are identical.
3. Based on the above evaluation, the usefulness score for this code snippet is 1. It contains information relevant to the problem, but it is easier to write the solution from scratch.

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

1. The problem requires checking if all elements in a list are identical.
2. The code snippet is incomplete and does not address the problem. It only filters elements in the list that are equal to 2.
3. Based on step 2, the code snippet is not useful for solving the problem. Therefore, the usefulness score is 0.

Usefulness: 0

Code Snippet-4

```
myList == [1, 2, 3, 4, 5]
```

Human Grader (Usefulness 0-4): 0

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 1
GPT-3.5 (w/ R.) : 1
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

1. The problem requires checking if all elements in a list are identical.
2. The code snippet does not address the problem at all. It simply assigns a list of integers to the variable ‘myList‘.
3. Based on step 2, the code snippet is not useful for solving the problem. Therefore, the usefulness score is 0.

Usefulness: 0

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

1. The problem requires checking if all elements in a list are identical.
2. The code snippet compares the list ‘myList‘ to a specific list ‘[1, 2, 3, 4, 5]‘. It does not check if all elements in ‘myList‘ are identical.
3. Score for usefulness: 0. The code snippet is not helpful for solving the problem.

D.4 PROBLEM-3

```
format number of spaces between strings ‘Python‘, ‘:‘ and ‘Very Good‘ to
    be ‘20‘
```

Reference Code:

```
print('%*s : %*s' % (20, 'Python', 20, 'Very Good'))
```

Code Snippet-0

```
cursor.execute('INSERT#SPACE#INTO#SPACE#INTO#SPACE#<unk>#SPACE#<unk>#
    SPACE#<unk>#SPACE#<unk>')
```

Human Grader (Usefulness 0-4): 0

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 0
GPT-3.5 (w/ R.) : 0
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

1. The problem requires formatting the number of spaces between strings to be a specific value.
2. The code snippet does not provide any information on how to format the number of spaces between strings. It only contains a SQL query with placeholders for values.
3. Usefulness score: 0. The code snippet is not helpful for solving the problem.

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

1. The problem requires formatting a specific number of spaces between strings.
2. The code snippet is completely irrelevant to the problem. It is a SQL query that inserts data into a table and contains no information on formatting spaces.
3. Usefulness score: 0. The code snippet is not helpful at all for solving the problem.

Code Snippet-1

```
print(`\ n`.join([str(int(x)) for x in Python]))
```

Human Grader (Usefulness 0-4): 0

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 0
GPT-3.5 (w/ R.) : 0
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

1. The problem requires formatting a string with a specific number of spaces between three substrings.
2. The code snippet does not address the problem at all. It converts a list of strings to a list of integers and joins them with a newline character.
3. Usefulness score: 0. The code snippet is not helpful for solving the problem.

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

1. The problem requires formatting the number of spaces between strings to a specific value.
2. The code snippet does not address the problem at all. It converts a list of strings to a list of integers and joins them with a newline character.
3. Usefulness score: 0. The code snippet is not helpful in solving the problem.

Code Snippet-2

```
``````.join(str(x)for x in str(Python))
```

**Human Grader (Usefulness 0-4)**: 0

**LLM Evaluator (Usefulness 0-4)**
GPT-3.5 (w/o R.) : 1
GPT-3.5 (w/ R.) : 1
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

*1. The problem requires formatting a string with a specific number of spaces between substrings.*
*2. The code snippet does not address the problem at all. It simply converts the*
```

string "Python" to a list of characters and then back to a string.
3. The code snippet is not helpful for solving the problem. It requires significant changes to even begin addressing the problem.
4. Usefulness score: 0.

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

1. The problem requires formatting a specific number of spaces between strings.
2. The code snippet does not address the problem at all. It simply converts the string "Python" into a list of its individual characters and then joins them back together with no spaces or formatting.
3. Usefulness score: 0. The code snippet is not helpful in solving the problem.

Code Snippet-3

```
``````.join(str(x)for x in str(Python))
```

**Human Grader (Usefulness 0-4)**: 0

**LLM Evaluator (Usefulness 0-4)**
GPT-3.5 (w/o R.) : 1
GPT-3.5 (w/ R.) : 1
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

*1. The problem requires formatting a string with a specific number of spaces between substrings.*
*2. The code snippet does not address the problem at all. It simply converts the string "Python" to a list of characters and then back to a string.*
*3. The code snippet is not helpful for solving the problem. It requires significant changes to even begin addressing the problem.*
*4. Usefulness score: 0.*

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

*1. The problem requires formatting a specific number of spaces between strings.*
*2. The code snippet does not address the problem at all. It simply converts the string "Python" into a list of its individual characters and then joins them back together with no spaces or formatting.*
*3. Usefulness score: 0. The code snippet is not helpful in solving the problem.*

**Code Snippet-4**

```
'Python : Very Good' .ljust(20)
```

Human Grader (Usefulness 0-4): 0

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 4
GPT-3.5 (w/o R.) + zero-shot-CoT : 2

1. The problem requires formatting a string to have a specific number of spaces between certain substrings.
2. The code snippet uses the 'ljust()' method to left justify the string and add
```

spaces to the right until it reaches a total length of 20 characters.
3. The code snippet covers the required functionality of adding spaces between the substrings, but it assumes a fixed length of 20 characters. If the required number of spaces changes, the code would need to be modified. However, for the specific problem as stated, the code snippet is clear and logical.
4. Based on the above evaluation, the code snippet is somewhat helpful and requires some modification to be fully useful. Therefore, the usefulness score is 2.

GPT-3.5 (w/ R.) + zero-shot-CoT : 4

1. The problem requires formatting a string with a specific number of spaces between certain substrings.
2. The code snippet uses the 'ljust()' method to left-justify the string and add spaces to the right until it reaches the specified length of 20 characters.
3. The code snippet covers the required functionality of the problem and presents it in a clear and logical order.
4. Based on the Evaluation Criteria, the code snippet is very helpful and solves the problem. Therefore, the usefulness score is 4.

D.5 PROBLEM-4

```
How to convert a string from CP-1251 to UTF-8?
```

Reference Code:

```
d.decode('cp1251').encode('utf8')
```

Code Snippet-0

```
print('<unk>#SPACE#<unk>#SPACE#<unk>#SPACE#<unk>')
```

Human Grader (Usefulness 0-4): 0

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 2
GPT-3.5 (w/ R.) : 1
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

1. The problem requires converting a string from CP-1251 to UTF-8.
2. The code snippet does not contain any code related to string conversion. It simply prints a string with some placeholders.
3. Therefore, the code snippet is not helpful in solving the problem.

Usefulness Score: 0

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

1. The problem requires converting a string from CP-1251 to UTF-8.
2. The code snippet does not contain any code related to string conversion. It simply prints a string with some placeholders. Therefore, it does not cover the required functionality of the problem.
3. Usefulness score: 0. The code snippet is not helpful in solving the problem.

Code Snippet-1

```
``````.join(chr(int(i)) for i in 10).encode(`utf8`)
```
```

Human Grader (Usefulness 0-4): 1

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 1
GPT-3.5 (w/ R.) : 1
GPT-3.5 (w/o R.) + zero-shot-CoT : 0

> *1. The problem requires converting a string from CP-1251 to UTF-8.*
> *2. The code snippet is not helpful as it does not address the problem requirements. It appears to be attempting to join a list of characters created from integer values and then encoding it in UTF-8, but it is not clear what the purpose of this is or how it relates to the problem.*
> *3. Score: 0*
>
> *Overall, the code snippet is not useful for solving the problem.*

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

> *1. The problem requires converting a string from CP-1251 to UTF-8.*
> *2. The code snippet is incomplete and does not provide a clear solution to the problem. It only joins a list of characters converted from integers and encodes it in UTF-8. It does not address the conversion from CP-1251 to UTF-8.*
> *3. Score: 0. The code snippet is not helpful in solving the problem.*

Code Snippet-2

```
unicodedata.normalize('NFKD', string).encode('utf8')
```

Human Grader (Usefulness 0-4): 1

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 3
GPT-3.5 (w/o R.) + zero-shot-CoT : 4

> *1. The problem requires converting a string from CP-1251 to UTF-8.*
> *2. The code snippet uses the 'unicodedata.normalize()' function with the 'NFKD' argument to normalize the string, and then encodes it to UTF-8 using the 'encode()' function with the 'utf8' argument.*
> *3. The code snippet covers the required functionality of converting a string from CP-1251 to UTF-8 in a clear and concise manner. Therefore, the usefulness score is 4.*
>
> *Usefulness: 4*

GPT-3.5 (w/ R.) + zero-shot-CoT : 4

> *1. The problem requires converting a string from CP-1251 to UTF-8.*
> *2. The code snippet uses the 'unicodedata.normalize' function to normalize the string to NFKD form and then encodes it to UTF-8 using the 'encode' function. It covers the required functionality in a clear and concise manner.*
> *3. The code snippet is very helpful and solves the problem.*
>
> *Usefulness: 4*

Code Snippet-3

```
unicodedata.normalize('NFKD', string).encode('utf8')
```

Human Grader (Usefulness 0-4): 0

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 3
GPT-3.5 (w/o R.) + zero-shot-CoT : 4

> *1. The problem requires converting a string from CP-1251 to UTF-8.*
> *2. The code snippet uses the 'unicodedata.normalize()' function with the 'NFKD' argument to normalize the string, and then encodes it to UTF-8 using the 'encode()' function with the 'utf8' argument.*
> *3. The code snippet covers the required functionality of the problem in a clear and concise manner. It is a one-liner that can easily be used to convert a string from CP-1251 to UTF-8.*
>
> *Usefulness: 4*

GPT-3.5 (w/ R.) + zero-shot-CoT : 4

> *1. The problem requires converting a string from CP-1251 to UTF-8.*
> *2. The code snippet uses the 'unicodedata.normalize' function to normalize the string to NFKD form and then encodes it to UTF-8 using the 'encode' function.*
> *3. The code snippet covers all required functionalities of the problem and presents them in a clear and logical order.*
> *4. Based on the Evaluation Criteria, the code snippet is very helpful and solves the problem. Therefore, the usefulness score is 4.*

Code Snippet-4

```
import codecs
```

Human Grader (Usefulness 0-4): 0

LLM Evaluator (Usefulness 0-4)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 3
GPT-3.5 (w/o R.) + zero-shot-CoT : 1

> *1. The problem requires converting a string from CP-1251 to UTF-8.*
> *2. The code snippet imports the codecs module, which provides functionality for encoding and decoding data. However, it does not provide any specific code for converting a string from CP-1251 to UTF-8. Therefore, the code snippet is only slightly helpful (score of 1).*
> *3. N/A*
>
> *Usefulness: 1*

GPT-3.5 (w/ R.) + zero-shot-CoT : 0

> *1. The problem requires a solution to convert a string from CP-1251 to UTF-8.*
> *2. The code snippet only imports the codecs module, but does not provide any*

implementation for the required functionality.
3. Based on step 2, the code snippet is not helpful for solving the problem.

Usefulness: 0

E EXAMPLES ON HUMANEVAL-X

E.1 HUMANEVAL-JAVA

E.1.1 PROBLEM-0

```java
import java.util.*;
import java.lang.reflect.*;
import org.javatuples.*;
import java.security.*;
import java.math.*;
import java.io.*;
import java.util.stream.*;
class Problem {
    // Create a function that takes a value (string) representing a number
    // and returns the closest integer to it. If the number is equidistant
    // from two integers, round it away from zero.
    // Examples
    // >>> closest_integer("10")
    // 10
    // >>> closest_integer("15.3")
    // 15
    // Note:
    // Rounding away from zero means that if the given number is equidistant
    // from two integers, the one you should return is the one that is the
    // farthest from zero. For example closest_integer("14.5") should
    // return 15 and closest_integer("-14.5") should return -15.
    public static int closestInteger(String value) {
```

Reference Code:

```java
        if (value.contains(".")) {
            while (value.charAt(value.length() - 1) == '0') {
                value = value.substring(0, value.length() - 1);
            }
        }
        double num = Double.parseDouble(value);
        int res = 0;
        if (value.substring(Math.max(value.length() - 2, 0)).equals(".5") ) {
            if (num > 0) {
                res = (int) Math.ceil(num);
            } else {
                res = (int) Math.floor(num);
            }
        } else if(value.length() > 0) {
            res = (int) Math.round(num);
        }
        return res;
    }
}
```

Code Snippet-0

```
double num = Double.parseDouble(value);
        return (int) Math.round(num);
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 3

Code Snippet-1

```
// Your code goes here
        try {
            // Attempt to parse the string into a decimal.
            double decimal = Double.parseDouble(value);
            // If the parse was successful, then we now have a decimal
    value.
            // However, we want to round it and get an integer.
            int rounded = (int) Math.round(decimal);
            // Finally, we want to return the integer value.
            return rounded;
        } catch (NumberFormatException e) {
            // If the parse failed, we should return an error message.
            return -1;
        }
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 3

Code Snippet-2

```
return BigDecimal.valueOf(Double.parseDouble(value)).setScale(0,
    RoundingMode.HALF_UP).intValue();
```

Execution (Functional Correctness 0-1): 1
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 3

E.1.2 PROBLEM-1

```
import java.util.*;
import java.lang.reflect.*;
import org.javatuples.*;
import java.security.*;
import java.math.*;
import java.io.*;
import java.util.stream.*;
class Problem {
    // Given a string text, replace all spaces in it with underscores,
    // and if a string has more than 2 consecutive spaces,
    // then replace all consecutive spaces with -
    // fix_spaces("Example") == "Example"
    // fix_spaces("Example 1") == "Example_1"
    // fix_spaces(" Example 2") == "_Example_2"
    // fix_spaces(" Example   3") == "_Example-3"
    public static String fixSpaces(String text) {
```

Reference Code:

```
        StringBuilder sb = new StringBuilder();
        int start = 0, end = 0;
        for (int i = 0; i < text.length(); i++) {
            if (text.charAt(i) == ' ') {
                end += 1;
            } else {
                if (end - start > 2) {
                    sb.append('-');
                } else if (end - start > 0) {
                    sb.append("_".repeat(end - start));
                }
                sb.append(text.charAt(i));
                start = i + 1;
                end = i + 1;
            }
        }
        if (end - start > 2) {
            sb.append('-');
        } else if (end - start > 0) {
            sb.append("_".repeat(end - start));
        }
        return sb.toString();
    }
}
```

Code Snippet-0

```
String result = text.replace(" ", "_");
        String result2 = result.replace("__", "-");
        return result2;
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 2

Code Snippet-1

```
// Your code here.
        if (text.length() == 0) {
            return text;
        }
        text = text.replaceAll("\\s{2,}", "-");
        return text.replaceAll("\\s", "_");
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 3

Code Snippet-2

```
return text.replaceAll("\\s+", "_");
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 3

E.2 HUMANEVAL-C++

E.2.1 PROBLEM-0

```
#include<assert.h>
#include<bits/stdc++.h>
// Create a function that takes a value (string) representing a number
// and returns the closest integer to it. If the number is equidistant
// from two integers, round it away from zero.
// Examples
// >>> closest_integer("10")
// 10
// >>> closest_integer("15.3")
// 15
// Note:
// Rounding away from zero means that if the given number is equidistant
// from two integers, the one you should return is the one that is the
// farthest from zero. For example closest_integer("14.5") should
// return 15 and closest_integer("-14.5") should return -15.
long closest_integer(std::string value) {
```

Reference Code:

```
    double w;
    w=atof(value.c_str());
    return round(w);
}
```

Code Snippet-0

```
float f = stof(value);
  int intf = (int)f;
  int diff_neg = abs(intf - f);
  int diff_pos = (int)f + 1 - f;
  if (diff_neg <= diff_pos)
    return intf;
  else
    return intf + 1;
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 3

Code Snippet-1

```
double num = std::stod(value);
  return num > 0 ? std::ceil(num) : std::floor(num);
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 3

Code Snippet-2

```
long result;
    // TODO: Write code to implement this function
    float fl = stof(value);
    float delta = fl - floor(fl);
    if (delta < 0.5) {
        result = floor(fl);
    } else {
```

```
        result = ceil(fl);
    }
    return result;
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 3

E.2.2 PROBLEM-1

```
#include<assert.h>
#include<bits/stdc++.h>
// Given a string text, replace all spaces in it with underscores,
// and if a string has more than 2 consecutive spaces,
// then replace all consecutive spaces with -
// fix_spaces("Example") == "Example"
// fix_spaces("Example 1") == "Example_1"
// fix_spaces(" Example 2") == "_Example_2"
// fix_spaces(" Example   3") == "_Example-3"
std::string fix_spaces(std::string text) {
```

Reference Code:

```
    string out="";
    int spacelen=0;
    for (int i=0;i<text.length();i++)
    if (text[i]==' ') spacelen+=1;
    else
    {
        if (spacelen==1) out=out+'_';
        if (spacelen==2) out=out+"__";
        if (spacelen>2) out=out+'-';
        spacelen=0;
        out=out+text[i];
    }
    if (spacelen==1) out=out+'_';
    if (spacelen==2) out=out+"__";
    if (spacelen>2) out=out+'-';
    return out;
}
```

Code Snippet-0

```
int l = text.length();
    std::string str;
    int f = 0;
    int c = 0;
    for(int i = 0; i < l; i++) {
        if(text[i] == ' ') {
            c++;
            if(c >= 2) {
                str += '-';
                c = 0;
            }
        }
        else {
            if(text[i] != ' ') {
                str += text[i];
                c = 0;
            }
        }
```

```
    }
    return str;
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 3

Code Snippet-1

```
std::vector<char> text_in_chars{};
  for (const auto &char_ : text) {
    text_in_chars.push_back(char_);
  }
  for (auto &char_ : text_in_chars) {
    if (char_ == ' ') {
      char_ = '_';
    }
  }
  std::vector<int> count_spaces_in_row{};
  int count_spaces = 0;
  for (int i = 0; i < text_in_chars.size(); ++i) {
    if (text_in_chars[i] == '_') {
      count_spaces++;
    } else {
      count_spaces_in_row.push_back(count_spaces);
      count_spaces = 0;
    }
  }
  std::vector<int> indexes_for_replacement{};
  for (int i = 0; i < count_spaces_in_row.size(); ++i) {
    if (count_spaces_in_row[i] > 2) {
      indexes_for_replacement.push_back(i);
    }
  }
  for (const auto &index : indexes_for_replacement) {
    text_in_chars[index] = '-';
  }
  std::string result{};
  for (const auto &char_ : text_in_chars) {
    result += char_;
  }
  return result;
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 2

Code Snippet-2

```
int count = 0;
    for (int i = 0; i < text.length(); i++) {
        if (text[i] == ' ') {
            count++;
        }
        else {
            count = 0;
        }
        if (count == 2) {
            text[i - 1] = '-';
            count = 1;
        }
        else if (count > 2) {
```

```
            text[i] = '-';
        }
    }
    return text;
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 3

E.3 HUMANEVAL-PYTHON

E.3.1 PROBLEM-0

```
def closest_integer(value: str) -> int:
    """
    Create a function that takes a value (string) representing a number
    and returns the closest integer to it. If the number is equidistant
    from two integers, round it away from zero.

    Examples
    >>> closest_integer("10")
    10
    >>> closest_integer("15.3")
    15

    Note:
    Rounding away from zero means that if the given number is equidistant
    from two integers, the one you should return is the one that is the
    farthest from zero. For example closest_integer("14.5") should
    return 15 and closest_integer("-14.5") should return -15.
    """
```

Reference Code:

```
    from math import floor, ceil

    if value.count('.') == 1:
        # remove trailing zeros
        while (value[-1] == '0'):
            value = value[:-1]

    num = float(value)
    if value[-2:] == '.5':
        if num > 0:
            res = ceil(num)
        else:
            res = floor(num)
    elif len(value) > 0:
        res = int(round(num))
    else:
        res = 0

    return res
```

Code Snippet-0

```
pass
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)

GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 2

Code Snippet-1

```
if value[-1] == "5":
        return round(float(value))
    else:
        return int(round(float(value)))
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 3

Code Snippet-2

```
return int(round(float(value)))
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 3

E.3.2 PROBLEM-1

```
def fix_spaces(text: str) -> str:
    """
    Given a string text, replace all spaces in it with underscores,
    and if a string has more than 2 consecutive spaces,
    then replace all consecutive spaces with -

    fix_spaces("Example") == "Example"
    fix_spaces("Example 1") == "Example_1"
    fix_spaces(" Example 2") == "_Example_2"
    fix_spaces(" Example   3") == "_Example-3"
    """
```

Reference Code:

```
    new_text = ""
    i = 0
    start, end = 0, 0
    while i < len(text):
        if text[i] == " ":
            end += 1
        else:
            if end - start > 2:
                new_text += "-"+text[i]
            elif end - start > 0:
                new_text += "_"*(end - start)+text[i]
            else:
                new_text += text[i]
            start, end = i+1, i+1
        i+=1
    if end - start > 2:
        new_text += "-"
    elif end - start > 0:
        new_text += "_"
    return new_text
```

Code Snippet-0

```
num_of_spaces = text.count(" ")
    if num_of_spaces == 0:
        return text
    elif num_of_spaces == 1:
        return text.replace(" ", "_")
    else:
        return re.sub(r' +', '-', text.replace(" ", "_"))
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 2

Code Snippet-1

```
text = text.replace(" ", "_")
    text = text.replace("_ ", "_")
    text = text.replace("_ _", "_")
    text = text.replace("  ", " ")
    text = text.replace(" ", "-")
    return text
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 2

Code Snippet-2

```
return re.sub(r'\s\s+', '-', re.sub(r'\s', '_', text))
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 3

E.4 HUMANEVAL-JAVASCRIPT

E.4.1 PROBLEM-0

```
//Create a function that takes a value (string) representing a number
// and returns the closest integer to it. If the number is equidistant
// from two integers, round it away from zero.
// Examples
// >>> closest_integer("10")
// 10
// >>> closest_integer("15.3")
// 15
// Note:
// Rounding away from zero means that if the given number is equidistant
// from two integers, the one you should return is the one that is the
// farthest from zero. For example closest_integer("14.5") should
// return 15 and closest_integer("-14.5") should return -15.
function closest_integer(value){
```

Reference Code:

```
  value = Number(value)
  let t = value % 1
```

```
  if (t < 0.5 && t > -0.5) { value -= t }
  else { value += t }
  return value
}
```

Code Snippet-0

```
let val = Number(value);
    if(val > 0){
        return Math.ceil(val);
    }else{
        return Math.floor(val);
    }
}
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 2

Code Snippet-1

```
value = parseFloat(value)
  return Math.round(value)
}
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 2

Code Snippet-2

```
value = +value;
  if (isNaN(value)) {
    return false;
  }
  return Math.round(value);
}
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 2

E.4.2 PROBLEM-1

```
//Given a string text, replace all spaces in it with underscores,
// and if a string has more than 2 consecutive spaces,
// then replace all consecutive spaces with -
// fix_spaces("Example") == "Example"
// fix_spaces("Example 1") == "Example_1"
// fix_spaces(" Example 2") == "_Example_2"
// fix_spaces(" Example   3") == "_Example-3"
function fix_spaces(text){
```

Reference Code:

```
  let t = ''
  let c = 0
```

```
  for (let i = 0; i < text.length; i++) {
    if (text[i] == ' ') { c++ }
    else if (c > 0) {
      if (c == 1) { t += '_' }
      if (c == 2) { t += '__' }
      if (c > 2) { t += '-' }
      t += text[i]
      c = 0;
    } else {
      t += text[i]
    }
  }
  if (c == 1) { t += '_' }
  if (c == 2) { t += '__' }
  if (c > 2) { t += '-' }
  return t
}
```

Code Snippet-0

```
var t = text.replace(/\s+/g,'-');
    return t.replace(/\s/g, '_');
}
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 3

Code Snippet-1

```
var arr = text.split('');
    var result = '';
    var count = 0;
    for(var i = 0; i < arr.length; i++){
        if(arr[i] == ' '){
            count++;
            if(count > 2){
                result += '-';
                count = 0;
            }
            else{
                result += '_';
            }
        }
        else{
            result += arr[i];
            count = 0;
        }
    }
    return result;
}
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 4
GPT-3.5 (w/ R.) : 3

Code Snippet-2

```
return text.replace(/\s{2,}/g, "-").replace(/\s/g, "_");
}
```

Execution (Functional Correctness 0-1): 0
LLM Evaluator (Functional Correctness 0-4, 4: fully correct, 0: totally incorrect)
GPT-3.5 (w/o R.) : 3
GPT-3.5 (w/ R.) : 3