INTERVENOR: Prompting the Coding Ability of Large Language Models with the Interactive Chain of Repair

Anonymous ACL submission

Abstract

This paper introduces **INTERVENOR** (INTERactiVE chaiN Of Repair), a system designed to emulate the interactive code repair processes observed in humans, encompassing both code diagnosis and code repair. INTERVENOR prompts Large Language Models (LLMs) to play distinct roles during the code repair process, functioning as both a `Code Learner` and a `Code Teacher`. Specifically, the `Code Learner` is tasked with adhering to instructions to generate or repair code, while the `Code Teacher` is responsible for crafting a Chain-of-Repair (CoR) to serve as guidance for the `Code Learner`. During generating the CoR, the `Code Learner` needs to check the generated codes from `Code Learner` and reassess how to address code bugs based on error feedback received from compilers. Experimental results demonstrate that INTERVENOR surpasses baseline models, exhibiting improvements of approximately 18% and 4.3% over GPT-3.5 in code generation and code translation tasks, respectively. Our further analyses show that CoR is effective to illuminate the reasons behind bugs and outline solution plans in natural language. With the feedback of code compilers, INTERVENOR can accurately identify syntax errors and assertion errors and provide precise instructions to repair codes. All data and codes will be released via GitHub.

1 Introduction

Large Language Models (LLMs), such as ChatGPT (OpenAI, 2022), have shown remarkable performance on code related tasks (OpenAI, 2023; Roziere et al., 2023; Wang et al., 2023b). This has significantly enhanced the efficiency and productivity in coding and software development (Qian et al., 2023a). Current approaches for code-based models involve pretraining language models on code corpora (Muennighoff et al., 2023; Luo et al., 2023; Li et al., 2023b; Zheng et al., 2023) and employing Chain-of-Thought (CoT) to prompt the coding proficiency of LLMs (Wei et al., 2022; Huang et al., 2023; Li et al., 2023a). However, compelling LLMs to directly generate entirely correct code proves to be exceptionally challenging, even for proficient programmers in real-world scenarios (Chen et al., 2023b).

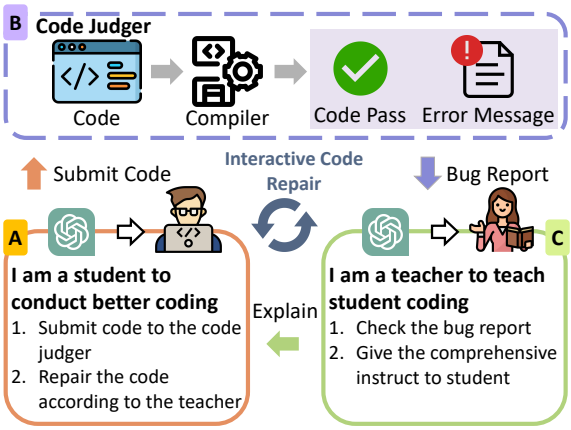


Figure 1: The Illustration of INTERVENOR. There are two agents in INTERVENOR, the teacher and student, who collaborate to repair the code. The error messages are utilized as a kind of INTERVENOR to alleviate the Degeneration-of-Thought (DoT) problem.

Recently, researchers focus on improving the code generation ability of LLMs through Self-Repair techniques (Olausson et al., 2023; Chen et al., 2023b). These methods leverage LLMs themselves to execute and repair codes, thereby enhancing the quality of generated code. Moreover, multi-agent collaborative coding approaches (Qian et al., 2023a; Dong et al., 2023) have also proven their effectiveness in handling difficult code tasks by prompting LLMs to play different roles, such as developers and testers. However, lots of bugs are difficult to find due to the cognitive inertia (McGuire, 1960)–overlooking the buggy codes that may not conform to their pre-existing coding thinking of LLMs. These agent-based code refinement methods (Dong et al., 2023; Qian et al.,

2023a) heavily rely on the self-evaluation capabilities of LLMs, potentially encountering the Degeneration-of-Thought (DoT) problem (Liang et al., 2023; Shinn et al., 2023).

This paper proposes **INTER**acti**VE** chai**N** **O**f **R**epair (INTERVENOR) to alleviate the DoT problem in code repair. The approach incorporates feedback from code compilers to enhance the code repair process. Following Dong et al. (2023), we develop two agents, namely, `Code Learner` and `Code Teacher`, to collaboratively repair codes interactively. As illustrated in Figure 1, the `Code Learner` is tasked with generating/repairing codes based on provided instructions. To craft specific code repair instructions to guide the `Code Learner` (Kaddour et al., 2023; Wang et al., 2023a), the `Code Teacher` generates the Chain-of-Repair (CoR) to illustrate the bug repair solutions for the `Code Learner`. Instead of Self-Debug (Chen et al., 2023b), `Code Teacher` incorporates the bug report from compilers to rethink the reasons of code errors and generate planning on how to repair the bugs. This interactive code repair process will continue until the code learner successfully fixes all code errors or reaches the predetermined maximum number of repair attempts.

Experimental results demonstrate the effectiveness of INTERVENOR by outperforming previous baseline models. Notably, INTERVENOR also achieves about 18% and 4.3% improvements over GPT-3.5 (OpenAI, 2022) in code generation and code translation tasks, showing its ability to improve the quality of generated codes through iterative code repair. Besides, we also build the CodeError dataset for evaluating the code repair ability of INTERVENOR by collecting the buggy code snippets from GPT-3.5 and real-world user-submitted codes. INTERVENOR further validates its efficacy by demonstrating a twofold increase in the number of successfully repaired codes.

Our further analyses illustrate that INTERVENOR is effective in leveraging the bug messages from code compilers, recognizing the reasons for code errors, and providing correction planning in natural language. Thanks to our CoR mechanism, INTERVENOR avoids thinking by LLMs themselves and can accurately diagnose the buggy codes and correct the assertion errors and name/syntax errors even in more difficult code generation scenarios. Our CoR mechanism enables LLMs to avoid designing complex code generation/repair prompts and achieve the best performance via only three-turn code repair. It also shows the potential to leverage the feedback from environments or rule systems to evolve LLMs (Olausson et al., 2023).

2 Related Work

Code generation tasks (Chen et al., 2021; Austin et al., 2021; Zheng et al., 2023) aim to generate correct and executable code based on the given natural language description, which has drawn lots of attention from researchers. LLMs such as ChatGPT (OpenAI, 2022) and GPT-4 (OpenAI, 2023) have shown strong effectiveness in generating code of high quality. To enhance the coding ability of LLMs, existing work focuses on code-specific pre-training and performs exceptionally well in code generation tasks (Roziere et al., 2023; Li et al., 2023b; Luo et al., 2023; Wang et al., 2023b).

Recently, some models focus on utilizing prompting techniques to enhance the coding capabilities of LLMs. CodeCoT (Huang et al., 2023) is inspired by Chain-of-Thought (Wei et al., 2022) and prompts the quality of generated codes using Code-CoT and Self-exam methods. LLMs are asked to craft the code and design a set of test cases to polish the codes. Structured Chain-of-Thought (SCoT) (Li et al., 2023a) further considers the program structure, such as sequences, branches, and loops, and prompts LLMs to generate intermediate reasoning steps with program structures. Nevertheless, forcing LLMs to directly generate the completely correct codes is challenging in the code generation task (Chen et al., 2023b).

To generate more accurate code, existing efforts primarily concentrate on Self-Refine (Olausson et al., 2023) and Self-Repair (Chen et al., 2023b) techniques. The Self-Refine models aspire to improve the quality of generated code by decoding multiple samples and subsequently selecting the most suitable one based on specific criteria. One strategy to formulate customized criteria involves executing the generated code and selecting the optimal one based on the resulting execution outcomes (Ni et al., 2023; Zhang et al., 2023b; Shi et al., 2022; Li et al., 2022). Another approach is to rerank multiple code solutions to determine the final code (Shi et al., 2022; Zhang et al., 2023b; Chen et al., 2023a; Inala et al., 2022). However, these methods necessitate significant computing resources for generating code candidates, rendering them inefficient (Zhang et al., 2023a).

Another research avenue involves employing an

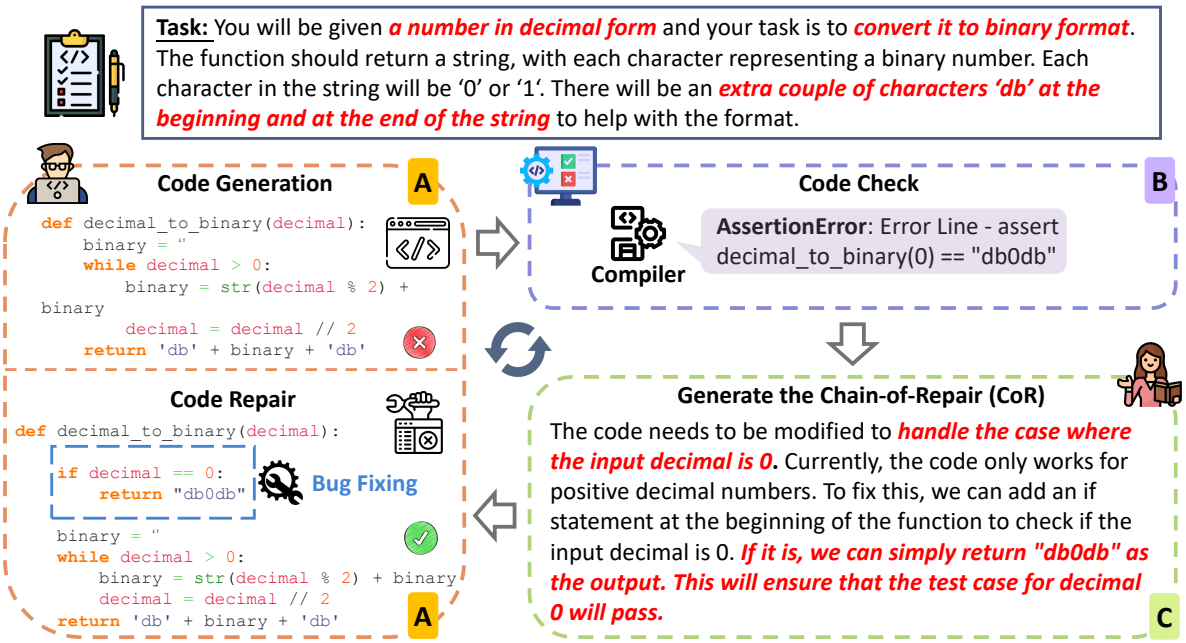


Figure 2: Illustration of Our Interactive Chain-of-Repair Model (INTERVENOR).

iterative code repair approach to enhance the quality of generated code (Zhang et al., 2023a; Welleck et al., 2023; Madaan et al., 2023; Shinn et al., 2023). Self-Debug (Chen et al., 2023b) utilizes explanations generated by LLMs to rectify self-generated code, while Self-Repair (Olausson et al., 2023) incorporates human-provided feedback for improvement. Self-Edit (Zhang et al., 2023a) employs error messages to refine generated code, but it necessitates the training of an additional fault-aware editor to generate a new program. Instead of directly integrating feedback for code repair, INTERVENOR designs an additional agent to reflect the reasons for coder errors and generate the Chain-of-Repair (CoR). This CoR is then employed to instruct the other agent in code repair through natural language.

Moreover, the work (Dong et al., 2023; Qian et al., 2023a) also designs a multi-agent collaborative approach to simulate the software development process and improve the efficiency of code generation. Nevertheless, these methods are highly dependent on the self-evaluation ability of LLMs and may face the Degeneration-of-Thought (DoT) problem (Liang et al., 2023; Shinn et al., 2023). Unlike them, INTERVENOR focuses on the bug-fixing process and proposes a simple but effective solution, which utilizes external tools, such as the Python interpreter, to execute the code (Xu et al., 2023; Qian et al., 2023b) and use the accurate bug report to facilitate the agent collaboration during interactive code repair.

3 Methodology

In this section, we introduce INTERVENOR, which conducts an interactive program repair process using LLM collaboration. We first describe the preliminary of code repair (Sec. 3.1) and then introduce our interactive Chain-of-Repair (CoR) mechanism (Sec. 3.2).

3.1 Preliminary of Code Repair

The code repair models mainly focus on Self-Repair (Olausson et al., 2023; Chen et al., 2023b). These models usually consist of three steps, including code generation, code execution, and code explanation. Self-Repair aims to use LLM itself to conduct the self-debug and self-execution processes and then iteratively repair codes. Nevertheless, programmers usually fail to recognize the code errors because of cognitive inertia (McGuire, 1960), making the code execution more difficult by LLM itself.

Different from these self-repair models, INTERVENOR follows previous work (Zhang et al., 2023a; Wang et al., 2022) to incorporate the feedback from compilers to prompt the code generation ability of LLMs. Instead of directly feeding the code bug message to LLMs, we design the *interactive chain of repair mechanism*, which builds two agents to rethink and repair the code errors. The

compiler serves as an INTERVENOR to avoid the Degeneration-of-Thought (DoT) problem during the interactive repair process.

3.2 Interactive Chain-of-Repair (CoR)

As shown in Figure 2, given the code generation tasks, INTERVENOR aims to mimic the human bug-repairing behavior by iteratively acquiring feedback from compilers, then generating bug reasons and solving plans in natural language, and finally fixing the program. Specifically, INTERVENOR employs two LLM based agents to play different roles in code repair (Sec. 3.2.1). Then we conduct an interactive code repair process using the agents (Sec. 3.2.2).

3.2.1 Agent Building

INTERVENOR involves the integration of two agents, `Code Learner` and `Code Teacher`, who work interactively to repair the generated codes. The role of the `Code Learner` follows the instructions to conduct code generation/repair, guided by the `Code Teacher`. The primary focus of the `Code Teacher` is to rethink and elucidate code errors for students. Additional details about agent construction can be found in Appendix A.5.

Code Learner. The `Code Learner` follows instructions and engages in two coding tasks, including initial code generation and code repair.

In the initial code generation phase, the `Code Learner` endeavors to generate the initial version of code according to the requirements of the given coding task. Subsequently, the agent's role is modified for code repair. We trigger the code repair proficiency of LLMs using the instruction "You are a student assistant with excellent code repair capabilities". Besides, `Code Learner` incorporates the Chain-of-Repair (CoR) as an instruction to guide the code repair.

Code Teacher. The `Code Teacher` conducts the CoR results during interactive code repair. It aims to generate the code error explanation and give bug-fixing planning for `Code Learner`.

Specifically, we use the prompt–"You are an experienced and insightful programming instructor"–to instruct LLMs to function as proficient code debuggers and serve to activate their bug-tracing and code-diagnosis abilities. `Code Teacher` integrates feedback from code compilers to produce extensive repair suggestions and guidance. This assists `Code Learner` in gaining a deeper understanding and effectively addressing errors within their code.

Benchmark	Language	Problems	#Tests
HumanEval	Python	164	7.8
MBPP	Python	500	3.1
HumanEval-X	C++	164	7.8
	Java	164	7.8
	JavaScript	164	7.8
CodeError	Python	4,463	9.0

Table 1: Data Statistics. #Tests represents the average number of test cases.

3.2.2 Interactive Code Repair Workflow

INTERVENOR conducts an interactive code-repair process, facilitating the collaboration among agents and the code compiler.

In the initial step (**Step A**^{0}, where 0 signifies the initial turn), we prompt the `Code Learner` to generate code for the given task. Subsequently, the `Code Learner` executes the generated code using the code compiler to assess its correctness (**Step B**). Following this, the `Code Teacher` generates code repair instructions (CoR) based on the bug report and the associated buggy code (**Step C**). These instructions elucidate the reason of the bug, such as "modified to handle the case where the input decimal is 0", and include code correction planning, for example, "we can simply return db0db". Such instructions are informative and enhance the guidance for the `Code Learner`. Ultimately, the `Code Learner` follows the *chain-of-repair (CoR)* to rectify the code and subsequently resubmits the corrected version to the compiler for execution in the subsequent turn ((**Step A**^{i}), where $i \geq 1$ denotes the code repair process). The A^i, B, and C steps are iterated sequentially until either the code meets the compiler's estimation or the maximum turn limit is reached.

4 Experimental Methodology

In this section, we describe the datasets, evaluation metrics, baselines, and implementation details.

Dataset. We evaluate the code generation and translation effectiveness on three datasets, including `HumanEval`, `MBPP` and `HumanEval-X`. Besides, we build a new benchmark `CodeError` to further test the code repair ability of LLMs. All data statistics are shown in Table 1.

`HumanEval` (Chen et al., 2021) serves as a benchmark for evaluating the functional correctness of synthesized programs generated from docstrings. It comprises 164 hand-written Python program-

ming problems, which consist of function signatures, docstrings, bodies, and multiple unit tests. MBPP (Austin et al., 2021) is a benchmark that includes 974 introductory-level Python programming problems. Each problem comprises a problem statement, a code solution, and three automated test cases, and the task IDs that range from 11 to 510 are used for evaluation. HumanEval-X (Zheng et al., 2023) is used to assess a model's multi-programming language generation and translation capability. It consists of 820 human-crafted data instances, covering C++, Java, JavaScript, and Go.

Then we build the CodeError benchmark to further evaluate the code repair effectiveness of INTERVENOR. The CodeError benchmark contains a total of 4,463 examples, evenly distributed across more than six different error types. It includes basic programming problems, data analysis problems, and programming competition problems. For each example, there are 9 test cases on average to evaluate the code's correctness. More details of CodeError are shown in Appendix A.3.

Evaluation Metrics. We use Pass@k (Chen et al., 2021) to evaluate the effectiveness of different models on both code generation task and code translation task, which is the same as the previous work (Chen et al., 2021; Zheng et al., 2023; Li et al., 2023a; Chen et al., 2023a; Nijkamp et al., 2023).

Baselines. We first compare INTERVENOR with several code-oriented large language pretrained models, such as Incoder (Fried et al., 2023), CodeGen (Nijkamp et al., 2023), CodeGeeX (Zheng et al., 2023), CodeT5 (Wang et al., 2023b), StarCoder (Li et al., 2023b), WizardCoder (Luo et al., 2023), and Llama based models (Touvron et al., 2023; Roziere et al., 2023). These models are pretrained on large-scale code corpora, demonstrating strong code generation capabilities. Additionally, we also compare INTERVENOR with some closed-source and high-performance large language models, *e.g.* Claude (Anthropic, 2023), GPT-3.5 (OpenAI, 2022), and GPT-4 (OpenAI, 2023), which show strong emergent abilities, especially for the code generation tasks. In our experiments, GPT-3.5 is our main baseline model. Besides, we also compare Self-Debug (Chen et al., 2023b) and the multi-agent collaborative method, Self-Collaboration (Dong et al., 2023) to show the code repair ability of INTERVENOR.

Implementation Details. In our experiments, we use GPT-3.5 as the foundation model to build different agents in our INTERVENOR model. We set the temperature to 0.2 and the maximum generation length to 512 tokens. The maximum number of interactive code repairs is set to 5. Additionally, we also use CodeLlama-7B/13B to implement the agents, Code Learner and Code Teacher, of our INTERVENOR model to explore the impact of using different LLMs.

5 Evaluation Results

In this section, we evaluate the overall performance of INTERVENOR. Then we conduct ablation studies and also show the effectiveness of Interactive CoR in different testing scenarios. Finally, case studies are presented.

5.1 Overall Performance

The overall performance of INTERVENOR in code generation and translation tasks is shown in Table 2.

Overall, INTERVENOR outperforms all baselines in all tasks by achieving more than 1% improvements, showing its effectiveness. Compared to our main baseline model GPT-3.5, INTERVENOR achieved about 18% and 4.3% improvements in code generation and code translation tasks, respectively. It illustrates that INTERVENOR has the ability to prompt the coding ability of LLMs by mimicking the human code repair behavior– *iteratively judging, rethinking, and repairing*. Notably, INTERVENOR also surpasses Self-Debug and Self-Collaboration models, demonstrating its ability to successfully intervene in the code generation/translation process and guide LLMs to better repair the codes using our chain of repair mechanism. All these experimental results highlight the generalization ability of INTERVENOR in improving LLMs' coding ability in different languages.

5.2 Ablation Studies

The ablation studies are conducted to show the effectiveness of the interactive CoR mechanism.

Firstly, we evaluate the code generation ability of LLMs using different code generation/repair prompting methods. As shown in Table 3, we compare CoT (Kojima et al., 2022), Few-Shot (Chen et al., 2021), and Few-Shot CoT (Wei et al., 2022) models in experiments, which prompt LLMs to better generate codes. These models try to generate natural language as the chain of coding thought (CoT) or provide some instances to demonstrate the coding task (Few-Shot). Then we compare different methods to generate the code repair instruction,

Model	Code Generation					Code Translation			
	HumanEval	MBPP	HumanEval-X			Target Language			
	Python	Python	C++	Java	JS	Python	C++	Java	JS
InCoder (Fried et al., 2023)	15.2	19.4	9.5	9.1	13.0	-	-	-	-
CodeGen (Nijkamp et al., 2023)	18.3	20.9	18.1	14.9	18.4	40.7	37.6	35.4	51.8
CodeGeeX (Zheng et al., 2023)	22.9	24.4	17.1	20.0	17.6	68.5	43.6	56.8	45.2
CodeT5+ (Wang et al., 2023b)	30.9	-	-	-	-	-	-	-	-
InstructCodeT5+ (Wang et al., 2023b)	35.0	-	-	-	-	-	-	-	-
PaLM-Coder (Chowdhery et al., 2023)	35.9	47.0	-	-	-	-	-	-	-
StarCoder (Li et al., 2023b)	40.8	49.5	-	-	-	-	-	-	-
WizardCoder (Luo et al., 2023)	57.3	51.8	-	-	-	-	-	-	-
LLama2 (Touvron et al., 2023)	30.5	45.4	-	-	-	-	-	-	-
CodeLLama (Roziere et al., 2023)	62.2	61.2	-	-	-	-	-	-	-
PanGu-Coder2 (Shen et al., 2023)	61.6	-	-	-	-	-	-	-	-
Claude (Anthropic, 2023)	47.6	-	-	-	-	-	-	-	-
GPT-4 (OpenAI, 2023)	67.0	-	-	-	-	-	-	-	-
Self-Debug (Chen et al., 2023b)	73.8	-	-	-	-	-	-	-	-
Self-Collaboration (Dong et al., 2023)	74.4	68.2	-	-	-	-	-	-	-
GPT-3.5 (OpenAI, 2022)	60.3	39.8	52.4	50.6	54.3	84.3	71.5	81.7	84.6
INTERVENOR	**75.6**	**69.8**	**67.1**	**68.3**	**67.1**	**89.8**	**75.6**	**85.4**	**88.3**

Table 2: Overall Performance of Different Models. We evaluate model effectiveness on code generation and code translation (HumanEval-X dataset) tasks using the Pass@1 evaluation metric. The baseline results are borrowed from corresponding papers. More evaluation results on the code translation task are shown in Appendix A.2.

Code Repair	Prompt Methods	HumanEval	MBPP	HumanEval-X			CodeError	Avg.
		Python	Python	C++	Java	JS	Python	
No Repair	Zero-Shot	60.3	39.8	52.4	50.6	54.3	-	-
	Zero-Shot CoT	51.8	35.2	48.2	45.7	45.4	-	-
	Few-Shot	62.2	45.4	53.1	62.2	43.3	-	-
	Few-Shot CoT	60.4	45.4	48.2	63.4	57.9	-	-
Single Turn	Zero-Shot	62.2	41.6	54.3	65.2	60.4	4.9	48.1
	Few-Shot	65.2	40.6	57.9	65.2	62.8	9.8	50.3
	CoT	66.5	48.8	56.7	62.2	60.4	10.3	50.8
	Self-Refine	65.2	48.8	57.3	64.0	60.4	5.2	50.2
	Error Msgs	67.1	51.8	57.3	59.8	62.8	9.8	51.4
	INTERVENOR (CoR)	69.5	51.0	59.8	65.2	60.4	15.9	53.6
Multi-Turns	INTERVENOR (CoR)	**75.6**	**69.8**	**67.1**	**68.3**	**67.1**	**21.7**	**61.6**

Table 3: Ablation Studies. We compare different prompting techniques to evaluate the effectiveness of our Chain-of-Repair (CoR) mechanism. All models are built based on GPT-3.5 and evaluated with Pass@1.

including Self-Refine (Madaan et al., 2023), Error Msgs, and Chain-of-Repair (CoR). Self-Refine asks the LLMs to rethink the errors by themselves, while Error Msgs and CoR incorporate the code error messages from compilers. Msgs directly uses error messages to guide the code repair process.

The experimental results show that code repair is more effective than directly prompting LLMs to generate codes. Even though we conduct different Few-shot and CoT methods to directly prompt LLMs, we only achieve 3.5% improvements, which shows that it is difficult for LLMs to generate correct codes without repairing. On the contrary, the code repair methods improve the quality of generated codes by achieving more than 9.7% improvements with only single-turn repair. Both Error Msgs and CoR thrive on the feedback from code compilers and achieve more than 1.2% improvements than Self-Refine, demonstrating that compilers can provide valuable signals to help LLMs better recognize the code bugs. Notably, CoR achieves the best performance among all code repair models, illustrating its effectiveness in guiding LLMs for code repair. CoR uses the error messages from code compilers to prompt LLMs, aiming to rethink the reasons for making errors and generate the instructions for repairing.

Then, as shown in Table 4, we investigate the impact of using different LLMs to build `Code Learner` and `Code Teacher`. When we use the identical LLM to build the `Code Learner`, the performance of INTERVENOR is improved with the enhanced capabilities of the `Code Teacher`, underscoring the efficacy of our CoR mechanism. This indicates that stronger models can conduct a more in-depth analysis of erroneous code and provide more accurate suggestions for code repair. Similarly, by keeping the same LLM to build the `Code`

Code Learner	Code Teacher	HumanEval Python	MBPP Python	HumanExal-X C++	HumanExal-X Java	HumanExal-X JS	CodeError Python
CodeLlama-7B	N/A	32.3	34.4	30.3	29.8	33.5	-
	CodeLlama-7B	32.9	35.1	31.0	31.0	34.1	4.6
	CodeLlama-13B	33.5	35.3	32.2	31.5	35.5	4.9
	GPT-3.5	**36.6**	**38.4**	**33.5**	**32.3**	**40.4**	**8.6**
CodeLlama-13B	N/A	39.6	36.4	36.6	33.5	39.6	-
	CodeLlama-7B	40.0	37.1	37.1	34.5	40.4	8.8
	CodeLlama-13B	42.1	37.7	38.6	36.6	42.8	9.4
	GPT-3.5	**43.6**	**40.4**	**42.2**	**37.5**	**46.1**	**12.6**
GPT-3.5	N/A	60.3	39.8	52.4	50.6	54.3	-
	CodeLlama-7B	65.2	42.8	55.4	55.1	56.1	11.8
	CodeLlama-13B	66.5	46.1	56.1	58.9	58.9	13.6
	GPT-3.5	**69.5**	**51.0**	**59.8**	**65.2**	**60.4**	**15.9**

Table 4: Model Performance of Different `Code Learner` and `Code Teacher` Model Pairings. We only conduct one turn code repair in experiments. "N/A" represents the initial output of `Code Learner` without any intervention from the `Code Teacher` for code repair, it reflects the initial generation results of `Code Learner`.

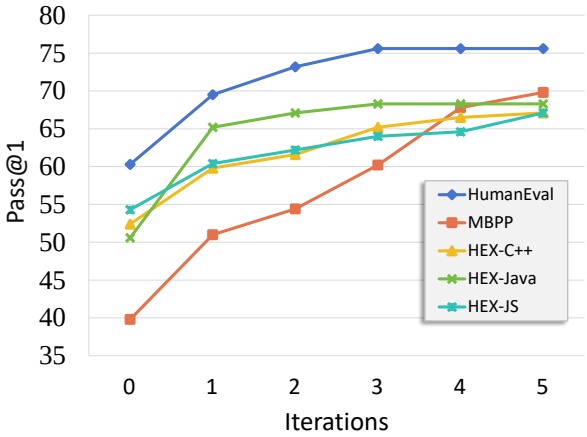
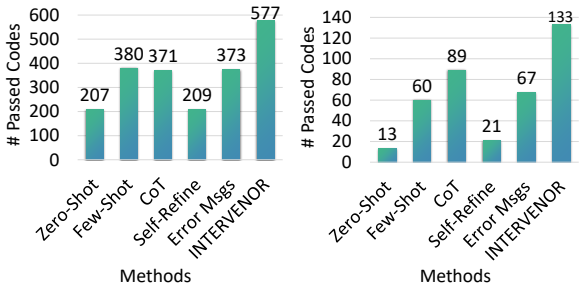


Figure 3: The Impact of Different Code Repair Turns. HumanEval, MBPP, and HumanEval-X (HEX) are used to evaluate our INTERVENOR model.



(a) AssertionError. (b) Other Error Types.

Figure 4: Code Repair Performance on the CodeError Dataset. We repair the error codes with one single turn. The codes are divided into two groups to evaluate the code repair effectiveness, including Assertion Errors and Others (AttributeError, NameError, RecursionError, SyntacError and TypeError).

`Teacher`, a stronger `Code Learner` also conducts better code repair results. This indicates that a more powerful `Code Learner` has a stronger instruction-following ability to better comprehend the CoR provided by the `Code Teacher`.

5.3 Effectiveness of INTERVENOR in Different Testing Scenarios

In this subsection, we delve deeper into exploring the effectiveness of INTERVENOR in two testing scenarios: 1) validating the impact of different code repair turns, and 2) evaluating the code repair effectiveness on different code error types.

As shown in Figure 3, the code generation performance is significantly improved during the iteratively repairing. After three turns, the INTERVENOR achieves almost the best performance on HumanEval and HumanEval-X datasets, showing the efficiency of our interactive chain-of-repair mechanism. For a more difficult dataset MBPP, INTERVENOR achieves 30% improvements and still has some room to achieve further improvements, demonstrating its advantages in dealing with more difficult and realistic coding tasks.

Then we show the code repair effectiveness on different error types in Figure 4. We use different prompt methods to stimulate LLMs to repair code errors and show their effectiveness on different types of code errors. Overall, INTERVENOR doubles the number of corrected code examples of baseline models, showing its effectiveness in repairing code errors. On the one hand, INTERVENOR significantly outperforms other methods on the AssertionError repairing task. It illustrates that INTERVENOR can provide more precise guidance and identify errors with the help of the bug report derived from the failed testing case. On the other hand, INTERVENOR also shows a strong ability to correct other code errors. Our Chain-of-Repair

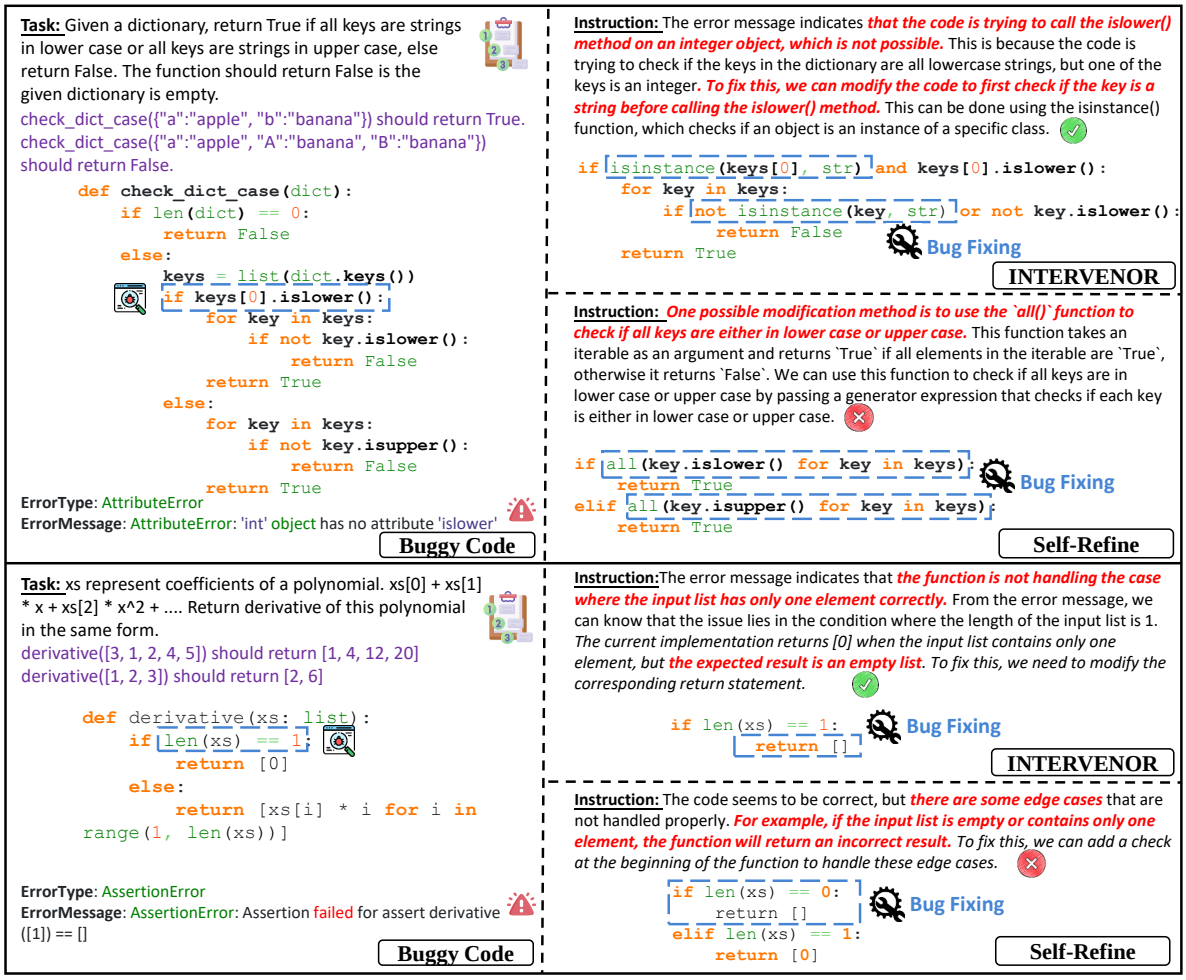


Figure 5: Case Studies. We provide two cases that showcase the effectiveness of the Chain-of-Repair (CoR) generated by INTERVENOR when fixing AttributeError and AssertionError, respectively.

(CoR) method also thrives on the error messages of the compiler, breaks the cognitive inertia of LLMs, and identifies the specific code error for repairing. All these phenomena show that the quality of code execution feedback is critical in repairing codes.

5.4 Case Studies

Finally, we show two cases in Figure 5 to demonstrate the effectiveness of INTERVENOR. We compare INTERVENOR with Self-Refine (Madaan et al., 2023), which prompts LLMs themselves to execute codes, recognize bugs, and repair codes.

Overall, the feedback from compilers indeed helps to improve the accuracy of repaired code by providing more valuable instructions. In the first case, Self-Refine fails to fix the AttributeError and adds the "all()" function in codes. On the contrary, INTERVENOR successfully fixes the code, showing its effectiveness. It accurately analyzes the reason of bugs "call the islower() method on an integer object, which is not possible" and also provides a solution by suggesting to "check if the key is a string before calling the islower() method". In the second case, the instruction provided by Self-Refine thinks the code seems to be correct and does not offer definite solutions, showing the cognitive inertia in debug–*It is hard to debug the code written by ourselves*. INTERVENOR shows its effectiveness in directly generating the reason for the bug: "the function is not handling the case where the input list has only one element correctly". More cases are shown in Appendix A.4.

6 Conclusion

This paper proposes INTERVENOR, which regards the feedback of compilers as the INTERVENOR to facilitate the agent collaboration of code error diagnose and repair. Our experiments show that INTERVENOR outperforms previous code generation/repair models by generating accurate code repair solutions and instructions.

Limitations

The effectiveness of INTERVENOR heavily depends on the capabilities of the foundation model, necessitating its proficiency in code error analysis and repair. The efficacy of INTERVENOR may be constrained when using less proficient LLMs to build the agents. Furthermore, in cases where codes exhibit solely logical errors, the compiler relies on test cases to furnish feedback. The quality and coverage of test cases are crucial in this scenario. Inadequate or poorly designed test cases may not be sufficient to detect or help rectify these logical errors effectively. Hence, the efficacy of INTERVENOR is intricately linked to the robustness and comprehensiveness of the test cases.

Ethics Statement

In conducting this research and presenting the findings in this paper, we have upheld rigorous ethical standards throughout the entirety of the process. Our experiments are conducted on commonly used datasets from previous works. Additionally, for our newly constructed dataset, we have also reached out to relevant platforms, and they have granted us permission to use the data for scientific research purposes. Besides, the code submitted to the Atcoder website has undergone stringent ethical review, with very few instances where ethical concerns arise. Furthermore, despite the risk of large language models (LLMs) generating toxic data, by providing clear role-playing instructions and only allowing LLMs to complete code-related tasks, we believe the model's output will not produce harmful content.

References

Anthropic. 2023. Model card and evaluations for claude models.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023a. Codet: Code generation with generated tests. In *Proceedings of ICLR*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023b. Teaching large language models to self-debug. *ArXiv preprint*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, pages 240:1–240:113.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration code generation via chatgpt.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *Proceedings of ICLR*.

Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2023. Fixeval: Execution-based evaluation of program fixes for programming problems. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 11–18. IEEE.

Dong Huang, Qingwen Bu, and Heming Cui. 2023. Codecot and beyond: Learning to program and test like a developer. *ArXiv preprint*.

Faria Huq, Masum Hasan, Md Mahim Anjum Haque, Sazan Mahbub, Anindya Iqbal, and Toufique Ahmed. 2022. Review4repair: Code review aided automatic program repairing. *Information and Software Technology*, page 106765.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu K Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Proceedings of NeurIPS*.

Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. *ArXiv preprint*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Proceedings of NeurIPS*, pages 22199–22213.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023a. Structured chain-of-thought prompting for code generation.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023b. Starcoder: may the source be with you! *ArXiv preprint*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Ré mi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science*, (6624):1092–1097.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Zhaopeng Tu, and Shuming Shi. 2023. Encouraging divergent thinking in large language models through multi-agent debate. *ArXiv preprint*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *ArXiv preprint*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback.

William J McGuire. 1960. Cognitive consistency and attitude change. *The Journal of Abnormal and Social Psychology*, (3):345.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *Proceedings of ICML*, pages 26106–26128.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *Proceedings of ICLR*.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation?

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue.

OpenAI. 2023. Gpt-4 technical report.

Chen Qian, Xin Cong, Wei Liu, Cheng Yang, Weize Chen, Yusheng Su, Yufan Dang, Jiahao Li, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023a. Communicative agents for software development.

Cheng Qian, Chi Han, Yi R. Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023b. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Proceedings of EMNLP Findings*, pages 6922–6939.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *ArXiv preprint*.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Proceedings of EMNLP*, pages 3533–3546.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Proceedings of NeurIPS*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models.

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable neural code generation with compiler feedback. In *Proceedings of ACL Findings*, pages 9–19.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023a. Self-instruct: Aligning language model with self generated instructions. In *Proceedings of ACL*, pages 13484–13508.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. pages 1069–1088.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of NeurIPS*, pages 24824–24837.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2023. Generating sequences by learning to self-correct. In *Proceedings of ICLR*.

Yiheng Xu, Hongjin Su, Chen Xing, Boyu Mi, Qian Liu, Weijia Shi, Binyuan Hui, Fan Zhou, Yitao Liu, Tianbao Xie, Zhoujun Cheng, Siheng Zhao, Lingpeng Kong, Bailin Wang, Caiming Xiong, and Tao Yu. 2023. Lemur: Harmonizing natural language and code for language agents.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *Proceedings of ICML*, pages 11941–11952.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023a. Self-edit: Fault-aware code editor for code generation. In *Proceedings of ACL*, pages 769–787.

Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen tau Yih, Daniel Fried, and Sida I. Wang. 2023b. Coder reviewer reranking for code generation. In *Proceedings of ICML*, pages 41832–41846.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x.

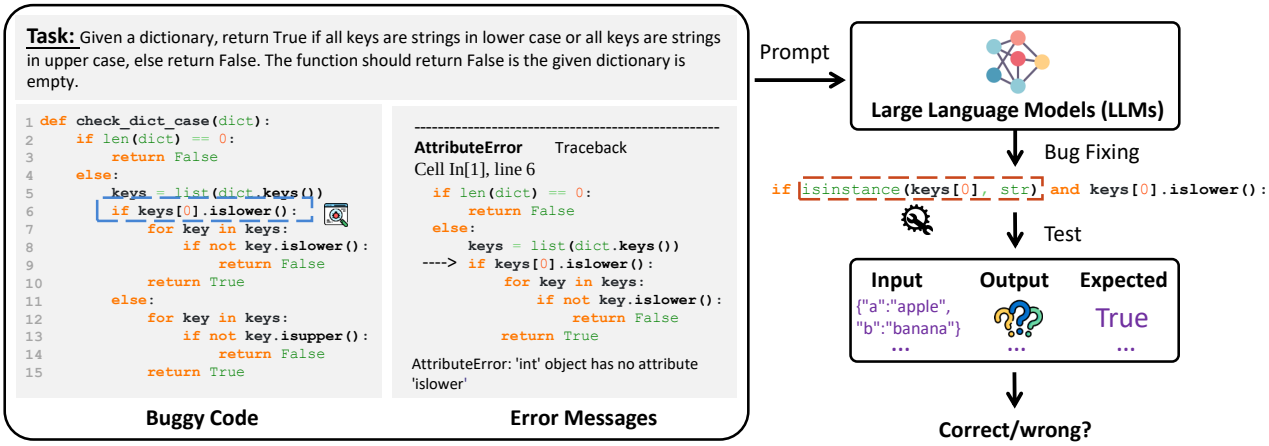


Figure 6: An Example of the CodeError Dataset. The CodeError benchmark asks LLMs to fix buggy codes and then evaluate whether the fixed code meets the requirements specified in the task description and passes all test cases. During the process of code repair, LLMs can utilize error messages to repair codes.

A Appendix

A.1 License

For all datasets in our experiments, HumanEval uses the MIT License, MBPP uses the CC-BY-4.0 License, and HumanEval-X uses the Apache License 2.0. All of these licenses allow their data for academic use.

A.2 Code Translation Results

As shown in Table 5, we present more detailed evaluation results on the code translation task.

In general, INTERVENOR shows the best performance across all twelve cross-language code translation tasks, demonstrating its ability to understand and translate codes. Compared with GPT-3.5, INTERVENOR improves its performance on all code translation tasks, especially on the tasks that translate other languages into Python. This indicates that INTERVENOR is more effective in generating an effective code repair chain according to the bug reports from the Python interpreter. Such a phenomenon aligns with human intuition during the debugging process, namely that bugs in Python code are more easily resolved.

A.3 More Details of the CodeError Benchmark

In our experiments, we build the CodeError benchmark to evaluate the code repair capabilities of LLMs, which facilitates the research on the code repair task. The CodeError dataset is collected from basic programming problems, data analysis problems, and programming competition problems. In this section, we show more detailed data information about the CodeError benchmark.

Source	Model	Target			
		Python	C++	Java	JS
Python	CodeGen	-	35.9	29.3	43.4
	CodeGeeX	-	34.2	42.0	34.8
	GPT-3.5	-	62.8	70.7	82.3
	INTERVENOR	-	**67.7**	**75.1**	**87.8**
C++	CodeGen	33.8	-	43.2	54.5
	CodeGeeX	62.8	-	71.7	50.8
	GPT-3.5	81.1	-	89.6	82.3
	INTERVENOR	**90.2**	-	**92.1**	**86.1**
Java	CodeGen	52.7	41.4	-	57.7
	CodeGeeX	75.0	49.7	-	50.0
	GPT-3.5	89.6	75.6	-	89.1
	INTERVENOR	**92.1**	**79.3**	-	**90.9**
JS	CodeGen	35.5	35.4	33.8	-
	CodeGeeX	67.7	46.9	56.6	-
	GPT-3.5	82.3	76.2	84.8	-
	INTERVENOR	**87.2**	**79.9**	**89.1**	-

Table 5: Code Translation Performance on Humaneval-X. We evaluate the code translation effectiveness among different program languages, including Python, C++, Java, and JavaScript (JS). We report the results of INTERVENOR, which only repairs codes with a single turn. All evaluation results are evaluated with Pass@1.

CodeError Examples. As shown in Figure 6, each example in CodeError consists of a programming task description, a buggy code snippet, error messages, and test cases to evaluate the code correctness of the repaired codes. If the fixed/generated code meets the requirements specified in the task description and passes all test cases, it proves that the repaired code is correct. During the process of code repair, LLMs need to fix the buggy code and are also able to use the error messages from the buggy code for assistance. We provide more examples that are sampled from the

Dataset	Language	Size	Avg.TC	Error Type	Error Msg	Category
DeepFix (Yasunaga and Liang, 2021)	C	6,971	✗	CE Only	✗	Basic
Review4Repair (Huq et al., 2022)	Java	2,961	✗	All	✗	Basic
Bug2Fix (Lu et al., 2021)	Java	5,835	✗	All	✗	Basic
Github-Python (Yasunaga and Liang, 2021)	Python	15k	✗	CE Only	✗	Basic
FixEval (Haque et al., 2023)	Java/Python	43k/243k	25.0	All	✗	Competition
CodeError (Ours)	Python	4,463	9.0	All	✓	Basic Data Analysis Competition

Table 6: A Comparison between CodeError and Other Code Repair Benchmarks. Size only represents the size of the test set. Avg.TC indicates the average number of test cases per problem. CE indicates compilation errors (e.g., SyntaxError). Error Msg indicates whether the buggy code contains detailed error information, such as the line of incorrect code, the reasons for the error, etc. Category represents the scope covered by the buggy code, and CodeError covers basic programming problems, data analysis problems, and programming competition problems.

	Basic	Comp	DA	Total/Avg.
Problem	326	3,888	249	4,463
AssertionError	236	2,949	52	3,237
NameError	22	62	32	116
TypeError	39	91	53	183
IndexError	1	92	8	101
ValueError	2	229	44	275
SyntaxError	11	375	4	390
Other Errors	15	90	57	162
Avg. Problem Words	10	47	140	49
Avg. Buggy Code	21	34	2	31
Avg. Test Cases	4	10	1.6	9

Table 7: Data Statistics of CodeError. We calculate the average word count per problem, the average number of lines in buggy code, and the average number of test cases per problem. Basic, Comp, and DA represent basic programming problems, programming competition problems, and data analysis problems, respectively.

basic programming problems, programming competition problems, and data analysis problems in Figure 7, Figure 8, and Figure 9, respectively.

Data Collection. CodeError is collected from various coding problems, making the dataset diverse and reliable.

To ensure the diversity of CodeError, we collect basic programming problems, data analysis problems, and programming competition problems from HumanEval, MBPP, DS-1000, APPS, and the programming contest site AtCoder. We conduct code generation experiments using GPT-3.5 on the first four datasets, preserving the generated codes that contain errors. For the programming contest site AtCoder, we crawl real-world user-submitted buggy codes to ensure the reliability of CodeError. The diverse data sources allow us to build a comprehensive and robust dataset to estimate the code repair ability of LLMs.

Data Statistics. The data statistics of CodeError are shown in Table 7.

The CodeError benchmark contains a total of 4,463 examples, evenly distributed across more than six different error types. These errors range from simple syntax errors to complex logic errors. And we provide detailed descriptions for common error types in Table 8. The predominant error type is the AssertionError (Logic Error). This phenomenon is quite normal in the real-world coding scenario since the current integrated development environment (IDE) can assist developers in avoiding simple errors such as SyntaxError and NameError but may not help to identify logic errors within the code. For each example, there are 9 test cases

AssertionError is an exception that is raised when an assert statement fails. In this paper, an AssertionError indicates that the code can run correctly but fails to pass certain test cases, suggesting the presence of potential logic errors that require further investigation and resolution.

NameError is an exception that is raised when a local or global name is not found. This error occurs when you try to access a variable or a function that is not defined or is not in the current scope.

TypeError is an exception that is raised when an operation or function is applied to an object of an inappropriate type. This typically occurs when you try to perform an operation that is not supported for the type of data you are working with.

IndexError is an exception that is raised when you try to access an index that does not exist in a list, tuple, or any other sequence. This typically happens when you attempt to access an index that is outside the range of the sequence.

ValueError is an exception that is raised when a built-in operation or function receives an argument that has the right type but an inappropriate value. Essentially, this error occurs when a function receives an argument of the correct type, but the value of the argument is not appropriate for the operation.

SyntaxError is an exception that is raised when there is an error in the syntax of your code. This can happen due to various reasons, such as missing parentheses, invalid keywords, or incorrect indentation.

Table 8: Common Code Errors and Their Descriptions.

on average to evaluate the correctness of the repaired codes.

Dataset Comparison. Finally, we compare the CodeError dataset with other code repair benchmarks. The differences are shown in Table 6.

Similar to FixEval (Haque et al., 2023), we utilize test cases to assess the functional correctness of the repaired code. Additionally, in comparison to other benchmarks (Yasunaga and Liang, 2021; Huq et al., 2022; Lu et al., 2021; Haque et al., 2023), we have two notable features:

- We provide detailed error information for erroneous codes, including error code localization, the cause of the error, error type, and more.
- CodeError covers a variety of problem types, including basic programming problems, data analysis problems, and programming competition problems, enabling a more comprehensive benchmark to evaluate the code repair capabilities of LLMs.

A.4 Additional Case Studies

In this subsection, we sample some cases from CodeError to demonstrate the effectiveness of the Chain-of-Repair (CoR) mechanism.

As shown in Figure 10, in the first case, there is a simple NameError in the buggy code, which indicates that 'hashlib' is not defined. We can see that the INTERVENOR recognizes hashlib as a Python package and provides a solution: "import the 'hashlib' module at the beginning of the code". In the second case, the error in the code is more subtle, involving an operation that should check the data structures of 'int' and 'list'. The CoR explicitly states that this is a "valid operation" and provides a solution: "check if the current element in the list is a list or not before adding it to the sum". With the help of accurate instruction, INTERVENOR successfully repairs the code.

Additionally, in Figure 11, we demonstrate that INTERVENOR can effectively address boundary issues, which are often quite tricky. INTERVENOR accurately identifies the code error in the loop condition and suggests a modification method: "modify the loop to iterate up to $n - 1$ instead of n". In Figure 12, we demonstrate that INTERVENOR can fix multiple error types in the code by iteratively repairing. Over two iterations, INTERVENOR identifies and accurately modifies the ValueError and TypeError in the code.

All these cases demonstrate the effectiveness of the Chain-of-Repair (CoR) mechanism and highlight the effectiveness of CoR in identifying errors and providing appropriate solutions for code repair.

A.5 Prompts for Agent Building

In this subsection, we show the prompts used in our experiments.

We first show the role-playing instructions used by `Code Learner` and `Code Teacher` in Figure 13. We utilize specific instructions to ensure that the `Code Teacher` and `Code Learner` within INTERVENOR play the correct roles and carry out code inspection and code repair.

Then, we present the prompts used in code generation and code repair. From Figures 14 to 15, we show the Zero-Shot, Zero-Shot CoT, Few-Shot, and Few-Shot CoT prompts for code generation tasks. From Figures 16 to 19, we show the prompts used in some code repair baseline models, including Zero-Shot, Few-Shot, CoT, Self-Refine, and Error Msgs. Self-Refine asks the LLMs to rethink the errors by themselves, while Error Msgs incorporate the code error messages from compilers and directly use them to guide the code repair process.

Task: Implement a function that takes an non-negative integer and returns an array of the first n integers that are prime numbers and less than n.

Buggy Code

```
def count_up_to(n):
    """Implement a function that takes an non-
negative integer and returns an array of the
first n integers that are prime numbers and less
than n.
    for example:
    count_up_to(5) => [2,3]
    count_up_to(11) => [2,3,5,7]
    count_up_to(0) => []
    count_up_to(20) => [2,3,5,7,11,13,17,19]
    count_up_to(1) => []
    count_up_to(18) => [2,3,5,7,11,13,17] """
    if n <= 1:
        return []
    primes = []
    for i in range(2, n):
        if is_prime(i):
            primes.append(i)
    return primes
```

Error Messages

```
---------------------------
---------------------------
---------------------------
NameError Traceback (most
recent call last)
Cell In[], line 17, in
count_up_to(n)

    15 primes = []
    16 for i in range(2, n):
--->17 if is_prime(i):
    18 primes.append(i)
    19 return primes

NameError: name 'is_prime'
is not defined
```

Figure 7: An Example from the Basic Programming Problems.

Task: Vasya has a string s of length n consisting only of digits 0 and 1. Also he has an array a of length n.
Vasya performs the following operation until the string becomes empty: choose some consecutive substring of equal characters, erase it from the string and glue together the remaining parts (any of them can be empty). For example, if he erases substring 111 from string 111110 he will get the string 110. Vasya gets a_x points for erasing substring of length x. Vasya wants to maximize his total points, so help him with this!

-----Input-----

The first line contains one integer n ($1 \le n \le 100$) — the length of string s.
The second line contains string s, consisting only of digits 0 and 1.
The third line contains n integers $a_1, a_2, \dots a_n$ ($1 \le a_i \le 10^9$), where a_i is the number of points for erasing the substring of length i.

-----Output-----

Print one integer — the maximum total points Vasya can get.

-----Examples-----

Input
7
1101001
3 4 9 100 1 2 3
Output
109

Input
5
10101
3 10 15 15 15
Output
23

-----Note-----

In the first example the optimal sequence of erasings is: 1101001 $\rightarrow$ 111001 $\rightarrow$ 11101 $\rightarrow$ 1111 $\rightarrow$ $\varnothing$.

In the second example the optimal sequence of erasings is: 10101 $\rightarrow$ 1001 $\rightarrow$ 11 $\rightarrow$ $\varnothing$.

Buggy Code

```
n = int(input())
s = input()
a = list(map(int,
input().split()))

max_points = 0
for i in range(n):
    if s[i] == s[i+1]:
        max_points += a[i]
    else:
        max_points += a[i+1]

print(max_points)
```

Error Messages

```
-----------------------------
-----------------------------
-----------------------------

NameError Traceback (most
recent call last)

    6 max_points = 0
    7 for i in range(n):
---->8 if s[i] == s[i+1]:
    9 max_points += a[i]
    10 else:
    11 max_points += a[i+1]

IndexError: string index out
of range
```

Failed Test Case

```
"input": "
          7
          1101001
          3 4 9 100 1 2 3",
"output": "",
"expected": "109"
```

Figure 8: An Example from the Programming Competition Problems. The error information includes the line of code where the error occurs and the test case that failed.

Task: I have a data frame like below

```
   A_Name  B_Detail  Value_B  Value_C   Value_D ......
0  AA      X1        1.2      0.5      -1.3     ......
1  BB      Y1        0.76     -0.7      0.8     ......
2  CC      Z1        0.7      -1.3      2.5     ......
......
```

This is just a sample of data frame, I can have n number of columns like (Value_A, Value_B, Value_C, ... Value_N)
Now i want to filter all rows where absolute value of all columns (Value_A, Value_B, Value_C, ...) is less than 1.
If you have limited number of columns, you can filter the data by simply putting 'and' condition on columns in dataframe, but I am not able to figure out what to do in this case.
I don't know what would be number of such columns, the only thing I know that such columns would be prefixed with 'Value'.
In above case output should be like

```
   A_Name  B_Detail  Value_B  Value_C   Value_D ......
1  BB      Y1        0.76     -0.7      0.8     ......
3  DD      L1        0.9      -0.5      0.4     ......
5  FF      N1        0.7      -0.8      0.9     ......
```

A:
<code>
import pandas as pd

```
df = pd.DataFrame({'A_Name': ['AA', 'BB', 'CC', 'DD', 'EE', 'FF', 'GG'],
            'B_Detail': ['X1', 'Y1', 'Z1', 'L1', 'M1', 'N1', 'K1'],
            'Value_B': [1.2, 0.76, 0.7, 0.9, 1.3, 0.7, -2.4],
            'Value_C': [0.5, -0.7, -1.3, -0.5, 1.8, -0.8, -1.9],
            'Value_D': [-1.3, 0.8, 2.5, 0.4, -1.3, 0.9, 2.1]})
```

</code>
df = ... # put solution in this variable
BEGIN SOLUTION
<code>

Buggy Code

```
df = df[(df.abs() < 1).all(axis=1)]
```

Error Messages

```
Traceback (most recent call last):
File "program.py", line 13, in <module>
  df = df.loc[(df.abs() > 1).any(axis=1)]
File "<hidden path>/lib/python3.8/site-
packages/pandas/core/generic.py", line 9773, in abs
  return np.abs(self) # type: ignore[return-value]
File "<hidden path> /lib/python3.8/site-
packages/pandas/core/generic.py", line 2032, in __array_ufunc__
  return arraylike.array_ufunc(self, ufunc, method, *inputs, **kwargs)
File "<hidden path> /lib/python3.8/site-
packages/pandas/core/arraylike.py", line 372, in array_ufunc
  result = mgr.apply(getattr(ufunc, method))
File "<hidden path> /lib/python3.8/site-
packages/pandas/core/internals/managers.py", line 325, in apply
  applied = b.apply(f, **kwargs)
File "<hidden path> /lib/python3.8/site-
packages/pandas/core/internals/blocks.py", line 381, in apply
  result = func(self.values, **kwargs)
TypeError: bad operand type for abs(): 'str
```

Figure 9: An Example from the Data Analysis Problem. We use the completion mode in DS-1000.

NameError

---Task

Given a string 'text', return its md5 hash equivalent string. If 'text' is an empty string, return None.

---Buggy Code

```
def string_to_md5(text):
    if text == '' :
        return None
    else:
        return hashlib.md5(text.encode('utf-8')).hexdigest()
```

---Error Message

NameError: name 'hashlib' is not defined

---Chain of Repair

The error message indicates that ***the 'hashlib' module is not imported***. To fix this, we can ***import the 'hashlib' module at the beginning of the code*** using the 'import' keyword.

---Corrected Code

```
def string_to_md5(text):
    import hashlib   Bug Fixing
    if text == '' :
        return None
    else:
        return hashlib.md5(text.encode('utf-8')).hexdigest()
```

TypeError

---Task

Write a function of recursion list sum.

---Buggy Code

```
def recursive_list_sum(data_list):
    if len(data_list) == 1:
        return data_list[0]
    else:
        if isinstance(data_list, list):
            return data_list[0] + recursive_list_sum(data_list[1:])
        else:
            return data_list
```

---Error Message

TypeError: unsupported operand type(s) for +: 'int' and 'list'

---Chain of Repair

The error message indicates that the ***function is trying to add an integer and a list, which is not a valid operation.*** To fix this, we can ***check if the current element in the list is a list or not before adding it to the sum.*** If it is a list, we can recursively call the function on that list and add the result to the sum. If it is not a list, we can simply add it to the sum. This way, we can handle both cases where the input is a list or an integer.

---Corrected Code

```
def recursive_list_sum(data_list):
    if len(data_list) == 1:
        return data_list[0]
    else:
        if isinstance(data_list, list):
            sum = 0
            for element in data_list:
                if isinstance(element, list):   Bug Fixing
                    sum += recursive_list_sum(element)
                else:
                    sum += element return sum
        else:
            return data_list
```

Figure 10: Case Studies. We present two cases from CodeError. The Chain-of-Repair clearly identifies the NameError and TypeError in the buggy code and provides the correct modification methods, which are highlighted in **red**.

Handling Edge Cases

---Task

Title: Minimum Difference Subarray

Given an integer array nums, write a function min_diff_subarray that returns a subarray such that the absolute difference sum between its elements is minimized. In other words, find a subarray such that the sum of the absolute differences between its elements is minimal.

-----Input-----

nums: An integer array where 1 <= len(nums) <= 10^5, and -10^4 <= nums[i] <= 10^4.

-----Output-----

Return a list containing integers, representing the found minimum difference subarray.

Example Input:

nums = [1, 4, 7, 2, 9, 10, 12]

result = min_diff_subarray(nums)

print(result)

Example Output:

[9, 10, 12]

In this example, the subarray [9, 10, 12] has the minimum sum of absolute differences between its elements.

---Buggy Code

```
def min_diff_subarray(nums):
    n = len(nums)
    if n == 0:
        return []
    min_diff = float('inf' )
    result = []
    for i in range(n):
        current_diff = abs(nums[i] -
nums[i + 1])
        if current_diff < min_diff:
            min_diff = current_diff
            result = [nums[i], nums[i +
1]]
    return result
# Example call
nums = [1, 4, 7, 2, 9, 10, 12]
result = min_diff_subarray(nums)
print(result)
```

1

---Chain of Repair

The error message indicates an "IndexError: list index out of range" at line 10 of the min_diff_subarray function. ***This error occurs when the loop variable i reaches the last index of the array, and nums[i + 1] attempts to access an index beyond the bounds of the array.*** To fix this issue, you can adjust the loop range to ensure that the loop doesn't iterate up to the last index. You can ***modify the loop to iterate up to n - 1 instead of n***.

3

---Error Message

```
IndexError Traceback (most recent call last)

Cell In[1], line 19
    17 # Example call
    18 nums = [1, 4, 7, 2, 9, 10, 12]
---->19 result = min_diff_subarray(nums)
    20 print(result)

Cell In[1], line 10, in
min_diff_subarray(nums)
    7 result = []
    9 for i in range(n):
---->10 current_diff = abs(nums[i] - nums[i +
1])
    11 if current_diff < min_diff:
    12 min_diff = current_diff

IndexError: list index out of range
```

2

--- Corrected Code

```
def min_diff_subarray(nums):
    n = len(nums)
    if n == 0:
        return []
    min_diff = float('inf' )
    result = []
    for i in range(n - 1):
        current_diff = abs(nums[i] -
nums[i + 1])
        if current_diff < min_diff:
            min_diff = current_diff
            result = [nums[i], nums[i +
1]]
    return result
# Example call
nums = [1, 4, 7, 2, 9, 10, 12]
result = min_diff_subarray(nums)
print(result)
```

Bug Fixing

4

Figure 11: Handling Boundary Issues.

Handling Multiple Bugs

---Task

Suppose you have a DataFrame containing student information, including names, student IDs, subjects, and grades.

1) Select all student information with grades above 90 in all subjects.

2) For each subject, calculate the average grade and append the results to the original DataFrame.

Turn 1

---Buggy Code 1

```
import pandas as pd
data = {
     'Name ': . . ., 'StudentID' : . . .,
    'Subject' : . . ., 'Grade' : . . .}
df = pd.DataFrame(data)
selected_students = df[df['Grade'] > 90 &
df['Grade'] < 100]
print(selected_students)
average_scores =
df.groupby('Subject')['Grade'].mean()
df.merge(average_scores, left_on='Subject',
right_index=True, how='left', suffixes=('',
'_Average'), inplace=True)
```

---Chain of Repair 3

The error is occurring due to the ***incorrect use of the logical operator &*** in the code. To fix this error, you should ***use parentheses to clarify the order of operations.*** This modification ensures that the conditions are properly grouped, resolving the ambiguity mentioned in the error message.

---Error Message 2

```
---------------------------------------------
---------------------------------------------
ValueError Traceback (most recent call last)
<hidden path>/lib/python3.8/site-
packages/pandas/core/generic.py in ?(self)
    1535 @final 1536 def __nonzero__(self):
 -> 1537 raise ValueError(
    1538 f"The truth value of a
{type(self).__name__} is ambiguous. "
    1539 "Use a.empty, a.bool(), a.item(),
a.any() or a.all()."
    1540 )
ValueError: The truth value of a Series is
ambiguous. Use a.empty, a.bool(), a.item(),
a.any() or a.all().
```

--- Corrected Code 4

```
import pandas as pd
data = {
    'Name': ['John', 'Alice', 'Bob', 'Eva',
'David' ],
    'StudentID': [101, 102, 103, 104, 105],
    'Subject': ['Math', 'English', 'Math',
'Physics', 'English' ],
    'Grade': [92, 85, 95, 88, 91] }
df = pd.DataFrame(data)
selected_students = df[( df['Grade'] > 90 )
& ( df['Grade'] < 100)]
print(selected_students)
average_scores =
df.groupby('Subject')['Grade'].mean()
df.merge(average_scores, left_on='Subject',
right_index=True, how='left', suffixes=('',
'_Average'), inplace=True)
```

Bug Fixing

Turn 2

---Buggy Code 1

```
import pandas as pd
data = . . .
df = pd.DataFrame(data)
selected_students = df[( df['Grade'] > 90 )
& ( df['Grade'] < 100)]
print(selected_students)
average_scores =
df.groupby('Subject')['Grade'].mean()
df.merge(average_scores, left_on='Subject',
right_index=True, how='left', suffixes=('',
'_Average'), inplace=True)
```

---Chain of Repair 3

Based on the error message, it appears that the ***merge() function does not support the inplace parameter***. To address this issue, you can modify the code by ***removing the inplace=True parameter*** from the merge() function.

---Error Message 2

```
. . .
TypeError: merge() got an unexpected keyword
argument 'inplace'
```

--- Corrected Code 4

```
. . .
df.merge(average_scores, left_on='Subject',
right_index=True, how='left', suffixes=('',
'_Average'), ~~inplace=True~~)
```

Bug Fixing, remove inplace parameter

Figure 12: Handling Multiple Bugs.

INTERVENOR

```
You are an experienced and insightful programming instructor, and you need to identify the
bugs in the given code based on the error messages.

- buggy code:
{buggy code}

When testing the above code, errors occurred: {error_message}, some test cases did not pass!
Please check the implementation of the function and provide a method for modification based
on the error message. No need to provide the modified code.

Modification method:
```

Code Teacher

```
You are a student assistant with excellent code repair capabilities. You can attempt to fix
the bugs in the above code based on the provided error information and the method for
modification. Please make sure to carefully check every potentially problematic area and make
appropriate adjustments and corrections.

- buggy code:
{buggy code}

When testing the above code, errors occurred: {error_message} , some test cases did not pass!
Please check the implementation of the function and fix the code based on the modification
method.

- modification method:
{modification method}

Correct the code:
```

Code Learner

Figure 13: Role Instructions in INTERVENOR. Within INTERVENOR, there are two LLM-based agents `Code Teacher` and `Code Learner`. We utilize specific instructions to ensure that they play the correct roles and carry out the intended tasks.

Code Generation

```
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

Zero-Shot

```
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    Let's write the code step by step.
    """
```

Zero-Shot CoT

```
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    . . .
    """
    {canonical_solution}

from typing import List
def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple groups of nested parentheses.
Your goal is to
    separate those group into separate strings and return the list of those.
    Separate groups are balanced (each open brace is properly closed) and not nested within
each other
    Ignore any spaces in the input string.
    . . .
    """
    {canonical_solution}

def truncate_number(number: float) -> float:
    """ Given a positive floating point number, it can be decomposed into
    and integer part (largest integer smaller than given number) and decimals
    (leftover part always smaller than 1).
    . . .
    """
    {canonical_solution}

{task description}
```

Few-Shot

Figure 14: Zero-Shot, Zero-Shot CoT, and Few-Shot Prompts for Code Generation Tasks.

Code Generation

```
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    Let's complete the following code step by step.
    """
    # Step 1: Create a variable to store the result
    result = False
    # Step 2: Loop through the list of numbers
    for i in range(len(numbers)):
        # Step 3: Check if the current number is within the threshold of any other number in
the list
        for j in range(i+1, len(numbers)):
            if abs(numbers[i] - numbers[j]) <= threshold:
                # Step 4: If the condition is met, set the result to True and break out of
the loop
                result = True
                break
        # Step 5: If the result is already True, break out of the loop
        if result:
            break

    # Step 6: Return the result
    return result

from typing import List
def rescale_to_unit(numbers: List[float]) -> List[float]:
    """ Given list of numbers (of at least two elements), apply a linear transform to that
list,
    such that the smallest number will become 0 and the largest will become 1
    >>> rescale_to_unit([1.0, 2.0, 3.0, 4.0, 5.0])
    [0.0, 0.25, 0.5, 0.75, 1.0]
    Let's complete the following code step by step.
    """
    # Step 1: Find the smallest and largest numbers in the list
    smallest = min(numbers)
    largest = max(numbers)
    # Step 2: Calculate the difference between the largest and smallest numbers
    difference = largest - smallest
    # Step 3: Create a new list to store the rescaled numbers
    rescaled_numbers = []
    # Step 4: Loop through each number in the original list
    for number in numbers:
        # Step 5: Apply the linear transform to each number
        rescaled_number = (number - smallest) / difference
        # Step 6: Add the rescaled number to the new list
        rescaled_numbers.append(rescaled_number)
    # Step 7: Return the new list
    return rescaled_numbers

def strlen(string: str) -> int:
    """ Return length of given string
    >>> strlen('')
    0
    >>> strlen('abc')
    3
    Let's complete the following code step by step.
    """
    # 1. Initialize a variable to store the length of the string
    length = 0
    # 2. Use a for loop to iterate through each character in the string
    for char in string:
        # 3. Increment the length variable by 1 for each character
        length += 1
    # 4. Return the length variable
    return length

{task description}
```

Few-Shot CoT

Figure 15: Few-Shot CoT Prompts for Code Generation Tasks.

Code Repair

```
You are a master at debugging code. Please correct the following buggy code.
-buggy code:
{buggy_code}

-correct code:
{task_description}
```

Zero-Shot

```
You are a master at debugging code. Please correct the following buggy code.

-buggy code:
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
        for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx == idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False

-correct code:
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
        for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False

. . .

-buggy code:
{buggy_code}

-correct code:
{task_description}
```

Few-Shot

Figure 16: Zero-Shot and Few-Shot Prompts for Code Repair.

Code Repair

```
You are a master at debugging code. Please correct the following buggy code.

<buggy_code>
from typing import List
def separate_paren_groups(paren_string: str) -> List[str]:
    """Input to this function is a string containing multiple groups of nested parentheses.
Your goal is to separate those group into separate strings and return the list of those.
Separate groups are balanced (each open brace is properly closed) and not nested within each
other.Ignore any spaces in the input string.
    """
    result = []
    current_string = []
    current_depth = 0
    for c in paren_string:
        if c == ')':
            current_depth += 1
            current_string.append(c)
        elif c == '(':
            current_depth -= 1
            current_string.append(c)

            if current_depth == 0:
                result.append(''.join(current_string))
                current_string.clear()

    return result
</buggy_code>

<repair_method>
The error in the original separate_paren_groups function lies in the handling of parentheses.
The function incorrectly increments current_depth when encountering a closing parenthesis and
decrements it when encountering an opening parenthesis. This leads to an incorrect count of
the depth of parentheses.To fix the issue, we should increment current_depth when an opening
parenthesis is encountered and decrement it when a closing parenthesis is encountered. This
ensures that the depth is properly tracked, and we append characters to current_string based
on the correct conditions.
</repair_method>

<correct_code>
from typing import List
def separate_paren_groups(paren_string: str) -> List[str]:
    """Input to this function is a string containing multiple groups of nested parentheses.
Your goal is to separate those group into separate strings and return the list of those.
Separate groups are balanced (each open brace is properly closed) and not nested within each
other.Ignore any spaces in the input string.
    """
    result = []
    current_string = []
    current_depth = 0
    for c in paren_string:
        if c == '(':
            current_depth += 1
            current_string.append(c)
        elif c == ')':
            current_depth -= 1
            current_string.append(c)

            if current_depth == 0:
                result.append(''.join(current_string))
                current_string.clear()

    return result
</correct_code>

<buggy_code>
{buggy_code}
</buggy_code>

<repair_method>
```

Few-Shot CoT

Figure 17: Few-Shot CoT Prompts for Code Repair.

Self-Refine

```
{buggy_code}

The above code may contain errors.
Please check the implementation of the function and provide a method for modification based
on your knowledge. No need to provide the modified code.

Modification method:
```

Repair Method Generation

```
{buggy_code}

The above code contains errors.
Please check the implementation of the function and fix the code based on the modification
method.

modification method:{motification_method}

Correct the code:
```

Code Repair

Figure 18: Prompts Used in Self-Refine Model. The instructions for generating code repair plannings and code repair are shown.

Error Msgs

```
You are an student assistant with excellent code repair capabilities. You can attempt to fix
the bugs in the code based on the error messages. Please make sure to carefully check every
potentially problematic area and make appropriate adjustments and corrections.

- buggy code:
{buggy code}

When testing the above code, errors occurred: {error_message} , some test cases did not pass!
Please check the implementation of the function and fix the code based on your knowledge.

Correct the code:
```

Figure 19: Prompts Used in Error Msgs Model. It conduct code repair by directly using the code error messages.