Eliciting Instruction-tuned Code Language Models' Capabilities to Utilize Auxiliary Function for Code Generation

Seonghyeon Lee[1], Suyeon Kim[1], Joonwon Jang[2], Heejae Chon[3], Dongha Lee[3], Hwanjo Yu[1]*
Department of Computer Science and Engineering, POSTECH, Pohang, South Korea[1]
Department of Artificial Intelligence, POSTECH, Pohang, South Korea[2]
Department of Artificial Intelligence, Yonsei University, Seoul, South Korea[3]
sh0416@postech.ac.kr

Abstract

We study the code generation behavior of instruction-tuned models built on top of code pre-trained language models when they could access an auxiliary function to implement a function. We design several ways to provide auxiliary functions to the models by adding them to the query or providing a response prefix to incorporate the ability to utilize auxiliary functions with the instruction-following capability. Our experimental results show the effectiveness of combining the base models' auxiliary function utilization ability with the instruction following ability. In particular, the performance of adopting our approaches with the open-sourced language models surpasses that of the recent powerful proprietary language models, i.e., `gpt-4o`.

1 Introduction

Generating codes based on natural language requirements, i.e., code generation, becomes an appealing application for natural language processing community due to the recent advance of code pre-trained language models (Singh et al., 2023; Zhou et al., 2023; Wang et al., 2023; Zhang et al., 2023). Pre-training on large-scale code corpora enables a language model to implement correct functions based on their requirements written in the docstrings. Also, tuning a code pre-trained language model to follow instructions has been released due to the effectiveness of instruction-tuned language models on natural language tasks (Luo et al., 2024; Wei et al., 2023; Song et al., 2024; Lei et al., 2024).[1] These instruction-tuned models boost up the code generation ability.

In code generation tasks, leveraging an auxiliary function reduces the implementation difficulty of a target function compared to that of implementing them from scratch. The auxiliary function is a function that helps implement a target function by inspiring novel mechanism or handling complicated subroutines for the target function through function calls (Lee et al., 2024). Therefore, properly utilizing the given auxiliary function becomes important for the instruction-tuned models.

However, limited research has been conducted on providing auxiliary functions to make the instruction-tuned models utilize the auxiliary functions effectively. Lee et al. (2024) initially included the auxiliary function in the prompt, but it showed inferior results compared to just prompting the corresponding base pre-trained models. Also, the instruction-tuned models' ability to incorporate the code content with their natural language text has not been fully explored, except that the model providers showcase some qualitative examples in their appendix (Rozière et al., 2024).

In this work, we comprehensively explore the instruction-tuned models' code generation behavior when they can access an auxiliary function. To do this, we design several prompts that are likely to elicit the ability to utilize auxiliary functions by leveraging the query-response structure employed in the instruction-tuned models. To be specific, we provide detailed information about the auxiliary function in the query and provide an incomplete codeblock to the prefix in the response to complete the remaining response. Then, we evaluate their effectiveness across several competitive instruction-tuned models. Our evaluation results show that our proposed prompts perform efficaciously on the instruction-tuned models compared to the corresponding base models, and even surpass `gpt-4o`, which is purportedly known as the most powerful language model. Finally, we perform an in-depth analysis to demonstrate that incorporating auxiliary function utilization ability already encoded in their base model with instruction-following capability

* Corresponding author

[1] From now on, we call an instruction-tuned code pre-trained model an instruction-tuned model for brevity.

Findings of the Association for Computational Linguistics: EMNLP 2024, pages 1840–1846

```
[INST] Write a Python function `find_outlier(numbers: List[float])
-> List[float]` to solve the following problem:
For a given list of input numbers, find the outlier. Outliers are
defined as data whose distance from the mean is greater than the
mean absolute deviation.
>>> find_outlier([1.0, 2.0, 3.0, 4.0])
[1.0, 4.0]

You can use a helper function `mean_absolute_deviation(numbers:
List[float]) -> float` to solve the problem:
For a given list of input numbers, calculate Mean Absolute
Deviation around the mean of this dataset.
>>> mean_absolute_deviation([1.0, 2.0, 3.0, 4.0])
1.0

[PYTHON]
def mean_absolute_deviation(numbers: List[float]) -> float:
    mean = sum(numbers) / len(numbers)
    return sum((abs(x - mean) for x in numbers)) / len(numbers)
[/PYTHON]

Your code should start with a [PYTHON] tag and end with a
[/PYTHON] tag.
[/INST] [PYTHON]
from typing import List

def mean_absolute_deviation(numbers: List[float]) -> float:
    """For a given list of input numbers, calculate Mean Absolute
Deviation around the mean of this dataset.
    >>> mean_absolute_deviation([1.0, 2.0, 3.0, 4.0])
    1.0"""
    mean = sum(numbers) / len(numbers)
    return sum((abs(x - mean) for x in numbers)) / len(numbers)

def find_outlier(numbers: List[float]) -> List[float]:
    """For a given list of input numbers, find the outlier.
Outliers are defined as data whose distance from the mean is
greater than the mean absolute deviation.
    >>> find_outlier([1.0, 2.0, 3.0, 4.0])
    [1.0, 4.0]"""
```

Figure 1: An example of our proposed prompts compatible with `CodeLlama-Inst` format. The green and purple part indicate inserting information about auxiliary function into the query and response, respectively.

through the response prefix is the main cause of the superior performance.

2 Related work

Several language models (Singh et al., 2023; Zhou et al., 2023; Wang et al., 2023; Zhang et al., 2023) are pretrained on massive code corpora and further finetuned to follow user instructions using a pre-defined chat format. They show progress on various representative coding benchmarks with a prompt template (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Li et al., 2022; Lee et al., 2024). The widely used template during the evaluation forms a function signature with the docstring containing the requirement. However, considering that the instruction-following models are trained with chat format, their effectiveness could be overlooked when we evaluating them without considering their formats. Lee et al. (2024) initially design a prompt template for instruction-following models, but the effectiveness on their models are marginal under their evaluation process. Along with the prompt engineering research line (Sondos Mahmoud Bsharat, 2023) demonstrating that even a simple prompt adjustment brings enormous gain by eliciting the power of language models, this work explores a better prompt that effectively combines the power of base model and the instruction following capabilities.

3 Methods

We design several prompts for pre-trained models and instruction-tuned models to show their ability to generate code with and without auxiliary functions. To do this, we build up two different approaches to naturally fuse the information inside the auxiliary functions into the instruction format used during instruction tuning.

3.1 Preliminary

We briefly explain how the existing work uses instruction-tuned models to implement codes. In general, the instruction-tuned models comply with a query-response format. Within this format, the models are trained to respond to that query. Therefore, we simply write a query with the given requirements to induce them to implement the code. The query consists of (1) the objective statement, (2) a description of a function, and (3) a formatting guideline. The objective statement commands them to implement a function, and a description of the function is followed to explain their functionality with some examples. Finally, the output formatting guideline is provided to easily parse the codeblock from their response. The constructed query is inserted into the pre-defined format for the model prompts. Then, the models generate a response that contains a codeblock with the implementation.

3.2 Incorporating the function utilization ability with instruction-following ability

On top of the existing approach, we propose simple and effective ways to enhance the instruction-tuned models' ability to utilize other functions.

Approach 1. Inserting auxiliary function information in the query-side The first approach (Figure 1, Green) is to insert the information about auxiliary function to the query. Assuming that the instruction-tuned models understand the codes in their query, we can expect that information about the auxiliary function, e.g., declaration, docstring, and their implementation, provided in the query would be comprehended by the models and leveraged when generating their response.

Approach 2. Inserting auxiliary function definition in the response-side Another approach (Figure 1, Purple) is to attach an informative starting text to the response to guide the models to naturally complete the remaining response. One similar approach that has been conducted in general tasks is to elicit language models' reasoning capability by attaching a starting text, e.g., let's think step by step, to the response (Kojima et al., 2022). Motivated by this work, we attach the incomplete codeblock that models should generate within their response. Specifically, we open a codeblock with an appropriate tag and function declaration requested to be implemented in the query with proper import statements. Using this approach also guarantees that the codeblock is properly opened, so the task for the models becomes easier as they just properly finalize the codeblock. In addition, we insert an auxiliary function into the codeblock for the models to leverage the auxiliary function during the generation.

4 Experiments

We explore the effectiveness of our proposed strategy with recent competitive instruction-tuned models and analyze their behavior from various angles.

4.1 Experimental setup

Basic setup We list the recent competitive instruction-tuned models and their corresponding base models in Table 1. We adopt the Humanextension benchmark consist of 151 relevant function pairs specially designed for measuring the language models' ability to utilize other functions (Lee et al., 2024). We follow the widely used decoding strategy for generating code: 0.2 for temperature and 0.95 for top p, and generate at most 512 tokens per prompt (Ben Allal et al., 2022). We evaluate the generated implementations using functional correctness by measuring the proportion of correct implementations that pass whole test cases among 20 generations, which is known as pass@1 score.

Measuring the base models' auxiliary function utilization capability We measure the pre-trained base models' ability to utilize other functions using the Humanextension benchmark with the prompt where the detailed prompt for the base models can be found in the Appendix. In doing so, we disentangle the strength of the instruction-tuned models from our experimental results by considering the improvement already observed in the base models. We compare this score to evaluate whether the proposed approaches with the instruction-tuned models could surpass this simple baseline.

4.2 Results

Overall results We demonstrate that our approaches successfully elicit the instruction-tuned models' ability to utilize auxiliary functions. We report the pass@1 score with various prompting approaches in Table 2. First, both adding information about auxiliary functions in the query and appending a codeblock with an auxiliary function definition in the response show a clear improvement compared to their closest counterparts. Comparing the first and third columns or the second and fourth columns demonstrates the effectiveness of providing information about auxiliary functions to the query. Also, comparing the second and fifth columns verifies the effectiveness of attaching the auxiliary function definition as a prefix for their response. Furthermore, our proposed prompts also work effectively in the bigger sized model such as `Deepseek-coder-33b-Inst`. We additionally report the performance of the powerful proprietary instruction-tuned models such as `gpt-3.5-turbo` and `gpt-4o` and verify that the open-sourced models can easily surpass them with our prompting approach.

Model analysis When we look at each model, `MagicoderS-CL-7b` shows superior performance compared to `CodeLlama-Inst-7b`, representing the importance of diverse instruction-tuned datasets and this trend is also observed in `Deepseek-coder-6.7b`. Assuming that `Deepseek-coder-Inst-6.7b` and `Deepseek-coder-Inst-7b` are trained on the same instruction-tuned dataset, enhancing the base models is also a plausible direction to improve their code generation capability with auxiliary functions. Comparing `CodeGemma-Inst-7b` and `CodeGemma-Inst-1.1-7b`, the pass@1 score is improved when they implement the given problem without an auxiliary function (from 0.3881 to 0.4626), but this improvement is not transferred when they access the auxiliary function (from 0.6228 to 0.6219). `Llama3-Inst-8b` shows different patterns compared to other models in that their score mostly increases when the auxiliary function is in the query. We speculate the reason for this results as `Llama3-8b` is pre-trained on the mixture of code and text corpora while other

Base model	No Aux	Aux	Instruction-tuned variants
CodeLlama-7b (Rozière et al., 2024)	0.1973	0.5284	CodeLlama-Inst-7b, MagicoderS-CL-7b
Deepseek-coder-6.7b (Guo et al., 2024)	0.2688	0.6741	Deepseek-coder-Inst-6.7b, MagicoderS-DS-6.7b
Deepseek-coder-7b (Guo et al., 2024)	0.3066	0.6152	Deepseek-coder-Inst-7b
CodeGemma-7b (CodeGemmaTeam, 2024)	0.2623	0.6043	CodeGemma-Inst-7b, CodeGemma-Inst-1.1-7b
Llama3-8b (AI@Meta, 2024)	0.1828	0.4914	Llama3-Inst-8b
Starcoder2-15b (Lozhkov et al., 2024)	0.3066	0.6682	Starcoder2-Inst-15b
CodeLlama-34b (Rozière et al., 2024)	0.2709	0.6411	CodeLlama-Inst-34b
Deepseek-coder-33b (Guo et al., 2024)	0.3599	0.7248	Deepseek-coder-Inst-33b

Table 1: Base models and their instruction-tuned variants. We measure the pass@1 score of the base models on Humanextension to correctly identify whether our approaches can acquire better performance compared to that of prompting the base model.

Instruction-tuned model	w/o auxiliary function		w/ auxiliary function			
1. Insert auxiliary function info to query			✓	✓		✓
2.1. Insert incomplete codeblock to response		✓		✓	✓	✓
2.2. Insert auxiliary function to incomplete codeblock					✓	✓
CodeLlama-Inst-7b (Rozière et al., 2024)	0.2907	0.2825	0.4844	0.5503	**0.5583**	0.5477
MagicoderS-CL-7b (Wei et al., 2023)	0.3977	0.4848	0.4656	0.6440	**0.6550**	0.6437
Deepseek-coder-Inst-6.7b (Guo et al., 2024)	0.3990	0.5497	0.6613	0.6623	**0.6894**	0.6828
MagicoderS-DS-6.7b (Wei et al., 2023)	0.4934	0.5507	0.6325	0.7050	0.6828	**0.7265**
Deepseek-coder-Inst-7b (Guo et al., 2024)	0.5348	0.6079	0.6414	0.6970	0.7166	**0.7348**
CodeGemma-Inst-7b (CodeGemmaTeam, 2024)	0.3086	0.3881	0.5324	0.6083	**0.6228**	0.6060
CodeGemma-Inst-1.1-7b (CodeGemmaTeam, 2024)	0.3354	0.4626	0.4563	0.5970	**0.6219**	0.6182
Llama3-Inst-8b (AI@Meta, 2024)	0.3632	0.3950	0.5801	**0.5868**	0.4970	0.5772
Starcoder2-Inst-15b (Lozhkov et al., 2024)	0.4182	0.5116	0.6325	**0.7079**	0.6834	0.6934
CodeLlama-34b-Inst (Rozière et al., 2024)	0.3599	0.3550	0.6219	**0.6421**	0.6146	0.6364
Deepseek-coder-33b-Inst (Guo et al., 2024)	0.4904	0.5957	0.6546	0.7503	0.7510	**0.7639**
gpt-3.5-turbo-0125 (Achiam et al., 2023)	0.4868		0.5901			
gpt-4o-2024-05-13 (Achiam et al., 2023)	0.6358		0.6987			

Table 2: Humanextension pass@1 score for instruction-tuned models with the proposed prompts. We mark bold on the most effective score in each model and underline the scores that surpass their base models. For gpt models, we could not report some scores as a user is not technically allowed to add a prefix to the response side.

models focus only on code corpora.

Comparison with the base models We further examine the performance by comparing that of their corresponding base models. We underscore the performance that surpasses the score that could be acquired by simply prompting the base models. The scores in the last three columns mostly outperform their base models, demonstrating that applying both approaches at the same time successfully elicits the instruction-tuned models' auxiliary function utilization ability. Based on our experimental results, the models generally surpass their base models when they gain knowledge about auxiliary functions with an incomplete codeblock in the response.

Response prefix	CodeLlama	CodeGemma
Add codeblock	0.5503	0.6083
Remove import statements	0.5593	0.6159
Remove docstring	0.5060	0.5758
Without codeblock	0.4844	0.5324

Table 3: Pass@1 score with various response prefix content. CodeLlama shorts for CodeLlama-Inst-7b and CodeGemma shorts for CodeGemma-Inst-7b

4.3 In-depth analysis for response codeblock

We perform an in-depth analysis on the effectiveness of appending an incomplete codeblock to the response by dissecting the code-block into several components and observing the performance change as we remove them sequentially. We report the performance in Table 3. For `CodeLlama`, removing the docstring for the target function in the codeblock significantly drops the performance. We conclude that `CodeLlama` understands the given requirements in the query with the docstring through the response codeblock. On the other hands, `CodeGemma` preserves the performance after removing the docstring to some extent. In this case, providing a function signature is much more crucial for `CodeGemma`. From this results, we find that the instruction-tuned models focus on different code components and we further investigate this phenomena in future work.

5 Conclusion

In this work, we study the instruction-tuned models' behavior when they can access the auxiliary function. Through various prompting approaches, i.e., providing an auxiliary function in the query or response, we discover effective prompting approaches that enhance the probability of implementing correct codes using open-sourced models and surpass the recent powerful proprietary models, i.e., `gpt-4o`. Our further investigation identifies that providing docstring or function signature to the response code-block is the major reason to boost performance. We believe that incorporating the ability to utilize other functions with the instruction-following capability is indispensable for generating complex code, and our work becomes a cornerstone towards this research direction.

6 Limitation

There are a few limitations that have not been fully addressed in this work. Due to the limited control of the proprietary models to the user, we could not report the score of `gpt-3.5-turbo` and `gpt-4o` when appending a prefix to the response which is verified as effective in the open-sourced models. Also, whether the improvement made by the proposed approaches could be transferred through fine-tuning does not explore in this work, which will be our main future work.

Acknowledgements

This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00584, (SW starlab) Development of Decision Support System Software based on Next-Generation Machine Learning) and No.2019-0-01906, Artificial Intelligence Graduate School Program (POSTECH)), the National Research Foundation of Korea (NRF) grant funded by the MSIT (South Korea, No.2020R1A2B5B03097210 and No. RS-2023-00217286), and the Digital Innovation Hub project supervised by the Daegu Digital Innovation Promotion Agency (DIP) grant funded by the Korean government (MSIT and Daegu Metropolitan City) in 2024 (No. DBSD1-07).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

AI@Meta. 2024. Llama 3 model card.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. 2022. A framework for the evaluation of code generation models. `https://github.com/bigcode-project/bigcode-evaluation-harness`.

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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

CodeGemmaTeam. 2024. Codegemma: Open code models based on gemma.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns,

Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1.

Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, volume 35, pages 22199–22213. Curran Associates, Inc.

Seonghyeon Lee, Sanghwan Jang, Seongbo Jang, Dongha Lee, and Hwanjo Yu. 2024. Exploring language model's code generation ability with auxiliary functions. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2836–2848, Mexico City, Mexico. Association for Computational Linguistics.

Bin Lei, Yuchen Li, and Qiuwu Chen. 2024. Autocoder: Enhancing code large language model with AIEV-INSTRUCT. *Preprint*, arXiv:2405.14906.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi 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*, 378(6624):1092–1097.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. 2023. CodeFusion: A pre-trained diffusion model for code generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 11697–11708, Singapore. Association for Computational Linguistics.

Zhiqiang Shen Sondos Mahmoud Bsharat, Aidar Myrzakhan. 2023. Principled instructions are all you need for questioning llama-1/2, gpt-3.5/4. *arXiv preprint arXiv:2312.16171*.

Zifan Song, Yudong Wang, Wenwei Zhang, Kuikun Liu, Chengqi Lyu, Demin Song, Qipeng Guo, Hang Yan, Dahua Lin, Kai Chen, and Cairong Zhao. 2024. Alchemistcoder: Harmonizing and eliciting code capability by hindsight tuning on multi-source data. *Preprint*, arXiv:2405.19265.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore. Association for Computational Linguistics.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *Preprint*, arXiv:2312.02120.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating code generation with pretrained models of code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13921–13937, Singapore. Association for Computational Linguistics.

A Prompts for base models with auxiliary functions

We provide an illustrative prompt for evaluating the base models' ability to utilize auxiliary function are provided as follow.

```
from typing import List

def mean_absolute_deviation(numbers):
  """{auxiliary docstring}"""
  {auxiliary implementation}

def find_outliers(numbers):
  """{target docstring}"""
```

We replace the content of auxiliary and target docstring and implementation for auxiliary function as placeholders for readability. In this example, the base models implement `find_outliers` and they could use `mean_absolute_deviation` as the function is given in the prompt.