# Python is Not Always the Best Choice: Embracing Multilingual Program of Thoughts

**Xianzhen Luo[1], Qingfu Zhu[1*], Zhiming Zhang[1], Libo Qin[2], Xuanyu Zhang[3], Qing Yang[3], Dongliang Xu[3], Wanxiang Che[1]**
[1]Harbin Institute of Technology, Harbin, China
[2]Central South University, Changsha, China
[3]Du Xiaoman (Beijing) Science Technology Co., Ltd.
{xzluo, qfzhu, zmzhang, car}@ir.hit.edu.cn
lbqin@csu.edu.cn
{zhangxuanyu, yangqing, xudongliang}@duxiaoman.com

## Abstract

Program of Thoughts (PoT) is an approach characterized by its executable intermediate steps, which ensure the accuracy of the logical calculations in the reasoning process. Currently, PoT primarily uses Python. However, relying solely on a single language may result in suboptimal solutions and overlook the potential benefits of other programming languages. In this paper, we conduct comprehensive experiments on the programming languages used in PoT and find that no single language consistently delivers optimal performance across all tasks and models. The effectiveness of each language varies depending on the specific scenarios. Inspired by this, we propose a task and model agnostic approach called MultiPoT, which harnesses strength and diversity from various languages. Experimental results reveal that it significantly outperforms Python Self-Consistency. Furthermore, it achieves comparable or superior performance compared to the best monolingual PoT in almost all tasks across all models. In particular, MultiPoT achieves more than 4.6% improvement on average on ChatGPT (gpt-3.5-turbo-0701)[1].

Today is the last day of the first quarter of 2008. What is the date one year ago from today?

**PoT with Python**

```
from datetime import datetime, timedelta
today = datetime(2008, 3, 31)
one_year_ago = today - timedelta(days=365)
```

Answer:04/01/2007❌

**PoT with R**

```
library(lubridate)
today <- ymd("2008-03-31")
one_year_ago <- today - years(1)
```

Answer:03/31/2007✅

**PoT with JavaScript**

```
let date = new Date(2008, 2, 31);
// months are 0-indexed in JavaScript
date.setFullYear(date.getFullYear() - 1);
```

Answer:03/31/2007✅

Figure 1: Comparison of PoT with different PLs. Python's 'timedelta' lacks support for year computation, leading to a leap year (2008 has 366 days) error by subtracting 365 days. R and JavaScript directly compute the year and get the correct answer.

## 1 Introduction

Program of Thoughts (PoT) aims to prompt Code Large Language Models (Code LLMs) to decompose complex problems into successive executable codes (Gao et al., 2023; Chen et al., 2022). Through execution by an external interpreter, the final results are accurately obtained, decoupling the computational process from the LLMs. PoT significantly reduces computation errors and improves reasoning performance (Wang et al., 2023a). Subsequently, benefiting from its flexibility and scalability, it is gradually applied to a broader spectrum of fields like image reasoning (Surís et al., 2023; Gupta and Kembhavi, 2023), financial QA (Koncel-Kedziorski et al., 2023) and robotic control (Li et al., 2023a). Nowadays, PoT has become a key method for enabling intelligence in agents (Yang et al., 2024; Wang et al., 2024). The widespread applicability highlights its significance.

Despite significant progress, PoT has a notable limitation: to the best of our knowledge, **all research on PoT focuses on Python**. However, since Code LLMs are capable of multilingual generation,[2] and most of the reasoning tasks are language-independent, many other programming languages (PLs) can also be applied to PoT, especially when considering their unique strength and diversity. From the perspective of **tasks**, different PLs repre-

* Corresponding author

[1]Code and data are released at https://github.com/Luowaterbi/MultiPoT

[2]In this paper, our "multilingual" represents multiple programming languages, not natural languages.



*Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 7185–7212
November 12-16, 2024 ©2024 Association for Computational Linguistics

sent PoT in different forms. As shown in Figure 1, the representation and calculation of dates in R is more concise than that in Python, which can reduce the complexity when LLMs generate PoTs. From the perspective of **models**, their multilingual ability is inconsistent. For instance, C++ of Deepseek Coder outperforms Python on the code generation task (Guo et al., 2024). It is natural to wonder whether this phenomenon also occurs on reasoning tasks. Therefore, a crucial question is raised with these perspectives: *Is Python truly the optimal language for all tasks and models for PoT?* Relying on Python may lead to a local optimum. In Figure 1, Python's '`timedelta`' does not support 'year', resulting in a miscalculation for the leap year. In contrast, R and JavaScript yield the correct answer.

Motivated by this, we conduct comprehensive experiments for multilingual PoTs. Beyond Python, we select four PLs: three widely used general languages (JavaScript, Java, and C++) and a niche but comprehensive language (R). For a comprehensive comparison, we identify five distinct sub-tasks within reasoning tasks: math applications (Cobbe et al., 2021; Patel et al., 2021; Miao et al., 2020), math (Hendrycks et al., 2021), tabular, date, and spatial (Suzgun et al., 2022). We select four backbone LLMs: ChatGPT (gpt-3.5-turbo-0701) and three strongest Code LLMs (Starcoder (Li et al., 2023b), Code Llama (Roziere et al., 2023), and Deepseek Coder (Guo et al., 2024)). Under both greedy decoding and Self-Consistency (Wang et al., 2022) settings, we answer that "***Python is not always the optimal choice, as the best language depends on the specific task and model being used.***"

In addition to the analysis contribution, to **leverage the strength of multiple PLs**, we further introduce a simple yet effective approach, called **MultiPoT** (**Multi**lingual **P**rogram **of** **T**houghts). It is a task and model agnostic approach, which uses LLMs to synchronously generate PoTs with various PLs and subsequently integrates their results via a voting mechanism. The use of **multiple PLs also provides greater diversity** and reduces the probability of repeating the same errors compared to single-language sampling. Experimental results demonstrate that MultiPoT outperforms Python Self-Consistency significantly. Furthermore, MultiPoT effectively matches or even surpasses the top-performing languages across nearly all tasks and models, and outperforms on averages. Especially on both ChatGPT and Starcoder, MultiPoT performs the best on four out of five tasks, with only a slight underperformance on the remaining task, and shows an improvement of over 4.6% compared to the best monolingual PoT on average.

Our contributions are summarized below:

- We conduct comprehensive experiments of PoTs with different PLs across various reasoning tasks and models, revealing that the choice of PL is dependent on tasks and models.
- We introduce a task and model agnostic approach called MultiPoT, which integrates multilingual PoTs and leverages strength and diversity across various PLs.
- Experimental results show that MultiPoT outperforms Python Self-Consistency and matches or surpasses the best language of each scenario. On both the model and task averages, MultiPoT enhances performance.

## 2 Related Work

### 2.1 Program of Thoughts

CoT is a specific form of in-context learning (Wei et al., 2022; Brown et al., 2020; Chowdhery et al., 2023). Its demonstrations consist of intermediate steps imitating the human thought process. It significantly enhances model's reasoning capabilities (Yang et al., 2023) but suffers from errors associated with calculations (Madaan and Yazdanbakhsh, 2022). CoT always uses Self-Consistency (Wang et al., 2023c) to improve answer accuracy through sampling and voting.

PoT (Chen et al., 2022; Gao et al., 2023) is an extension of CoT to avoid incorrect calculation. It represents intermediate steps as comments and code and executes the entire program with an interpreter to obtain answers. PoT not only excels in reasoning tasks but has rapidly extended to practical applications, including chart understanding, image reasoning, financial QA and robotic control (Zhang et al., 2024; Surís et al., 2023; Gupta and Kembhavi, 2023; Koncel-Kedziorski et al., 2023; Li et al., 2023a). It has become a key method for agents to perform complex reasoning and tool invocation (Yang et al., 2024; Wang et al., 2024). It is important to note that all previous PoT work only use Python. For the first time, we are exploring PoTs that use multiple PLs.

### 2.2 Usage of Multiple PLs

The training datasets naturally include a variety of PLs, endowing Code LLMs with the ability to

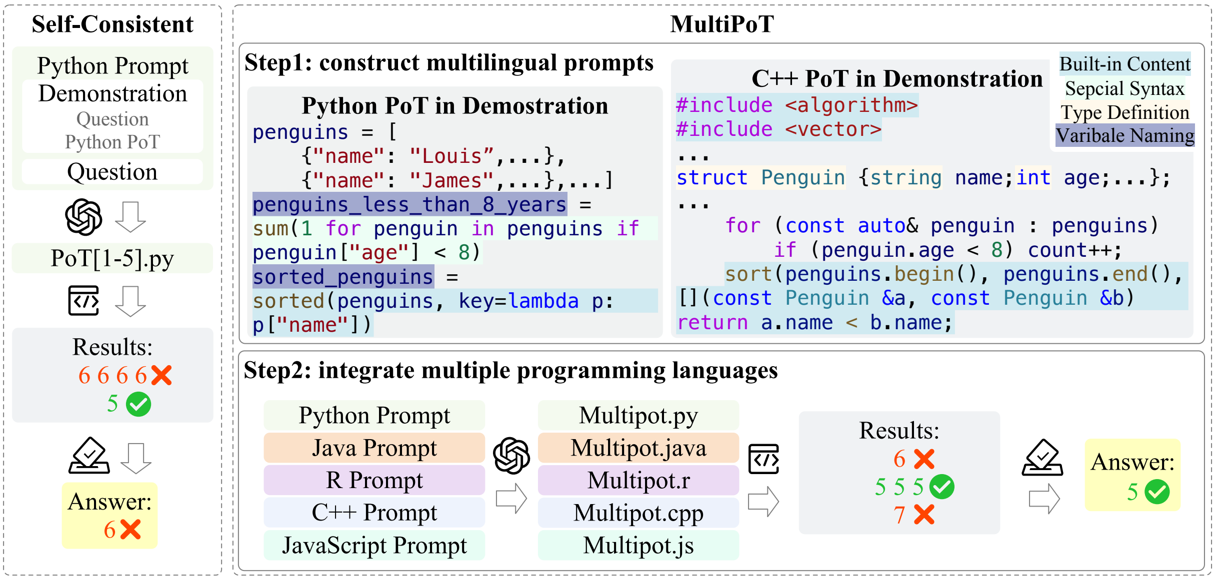


Figure 2: An overview of MultiPoT and Self-Consistency. MultiPoT first constructs prompts for each PL, ensuring a consistent reasoning process while also considering the distinct coding styles. It then integrates these PLs: generating multilingual PoTs based on the prompts, executing them to gather results, and finally voting for the answer. In contrast to Self-Consistency's single-language focus, MultiPoT leverages multiple PLs.

handle multilingual programming (Kocetkov et al., 2022; Nguyen et al., 2023; Gao et al., 2020; Nijkamp et al., 2023; Chen et al., 2021). This capability extends code tasks like generation, optimization, translation, and repair to other languages beyond Python (Gimeno et al., 2023; Shypula et al., 2023; Zhang et al., 2023; Wu et al., 2023). Despite the progress, current multilingual research (Jin et al., 2023; Joshi et al., 2023; Khare et al., 2023) mainly focuses on code-related tasks, neglecting the potential of PLs as tools to assist in other tasks. Additionally, these studies often treat each language separately without interaction. Our study pioneers the use of multiple PLs in reasoning tasks and introduces a novel integrated approach, leveraging the collective strength and diversity of various PLs to enhance overall performance.

# 3 Methodology

Figure 2 provides an overview of MultiPoT and Self-Consistency to highlight their differences. Concretely, MultiPoT consists of two main steps. First, a dedicated prompt is designed for each PL to sufficiently leverage the capability of the model with regard to the PL (Section 3.1). Second, PoTs in various PL are respectively generated by prompting the LLM with the prompts. The final answer is obtained by executing the PoTs and integrating their results via a voting mechanism (Section 3.2). Distinct from Self-Consistency, which relies on a single PL, MultiPoT integrates various PLs to utilize their strength and diversity.

## 3.1 Multilingual Prompts Construction

To instruct a LLM to generate PoT for a given question, a demonstration is included in the prompt. The demonstration consists of an example question and PoT. To ensure fairness, demonstrations of various PLs share the same example questions. Based on that, to efficiently leverage the capability of a LLM with regard to a PL, each PL is provided with a dedicated example PoT, taking into account its language-specific characteristics (Wang et al., 2023b). Note that language-agnostic features, such as algorithms and data structures, remain the same for example PoTs of all PLs, ensuring an identical reasoning process.

Concretely, the language-specific characteristics of each PL for constructing its dedicated example PoT includes **Built-in Content**, **Special Syntax**, **Type Definition**, and **Varibale Naming**. Figure 2 provides some examples of the characteristics. (1) while Python can directly employ the '`sort`' function, C++ has to load it from the '`algorithm`' library. Regarding variables, Python's '`list`' is more similar to C++'s '`vector`' than its array. (2) List comprehension like '`sum(1 for penguin in penguins if penguin["age"] < 8)`' is a standard syntax in Python. However, a straightforward for-loop is the common practice in other PLs. (3) Static PLs such as C++ require to define the variable type. We carefully define '`int`' and '`double`' variables to ensure computational accuracy and enhance flexibility by defining '`struct`'. (4) We keep the naming styles of each PL. For instance, Python uses Snake Case, whereas Java favors Camel Case ('`secondPenguin`'). Appendix A.5 shows the demonstrations. The above examples present the variations in example PoTs across different PLs. To accurately assess the model's capability in a specific PL, it is crucial to carefully consider its characteristics during the process of constructing.

Based on identical reasoning process, we successfully craft demonstrations of each PL exhibiting its characteristics. By adding the question after the demonstration, we get the prompt for each PL.

### 3.2 Integration

While Self-Consistency enhances performance by sampling to explore more reasoning paths, it can lead to repeated errors across different samples. In contrast, MultiPoT constructs multilingual prompts and generates PoTs in multiple PLs, significantly increasing the diversity of results.

Specifically, after constructing prompts for each PL, models generate corresponding PoTs, while tracking cumulative probabilities. These probabilities indicate the model's confidence in each answer, with higher probabilities denoting greater confidence. PoTs are then executed and results are collected. The final answer is determined by voting on these results. In cases of tied votes, answers with higher cumulative probabilities are favored. The integration of multiple PLs introduces more potential correct answers and reduces the probability of the same errors in candidate results.

## 4 Experiment Setup

### 4.1 Programming Languages

When selecting PLs to compare with Python, we focus on diversity. JavaScript is the most popular language on GitHub (GitHub, 2023) and has less overlap in application with Python, particularly in the ML/AI domains. R is a flexible and powerful language like Python but has much less data in pre-training data. The three PLs above are dynamic languages that do not require explicit variable type definitions. To incorporate the diversity of language types, we select the two most common static languages, Java and C++. The latter is closer to low-level programming and has fewer extension packages. We do not include C due to its high similarity with C++. These five languages offer a diverse range of application scenarios, data volumes, and language types compared to Python.

### 4.2 Tasks

We select representative and discriminating tasks. We initially select four tasks from Gao et al. (2023): **Math Application (Appl.)**, **Date**, **Tabular** and **Spatial**, and add the task **Math**. Appl. contains elementary-level mathematical application problems (GSM8K, SVAMP, Asdiv (Cobbe et al., 2021; Patel et al., 2021; Miao et al., 2020)). Date, Tabular, and Spatial are extracted from BBH-Hard (Suzgun et al., 2022) (Date Understanding, Penguins in a Table, Reasoning about Coloured Objects). These tasks assess understanding and reasoning about temporal sequences, structured text, and spatial positioning respectively. Math, consisting of the transformed MATH (Hendrycks et al., 2021) dataset. The difference between Math and Appl. lies in the level of difficulty. Math is more challenging and directly describes the math question without scenarios. The five tasks are distinct and representative of the evaluation of reasoning capabilities. They are language-agnostic, meaning that they can be performed in any PL, effectively demonstrating the model's reasoning ability across different languages. The additional details of the tasks are in the Appendix A.1.

### 4.3 Metric

Remaining consistent with previous work (Chen et al., 2022; Gao et al., 2023), the metric is accuracy. For tasks whose ground truth are real numbers (Appl./Math), the answer is considered correct if its difference from the ground truth is less than 1e-3; for tasks with string-type ground truth (Date/Tabular/Spatial), the answer is considered correct only if it is exactly the same as the ground truth.

### 4.4 Backbone LLMs

As the previously used code-davinci family is no longer accessible, we select four backbone LLMs, including the three strongest Code LLMs: **Starcoder** (15B), **Code Llama** (34B), and **Deepseek Coder** (33B). We select the base versions. The experiments of the Python version are discussed in Section 6.2, and the results are consistent with our conclusions and methodology. **ChatGPT** is also utilized as a representative of code-capable

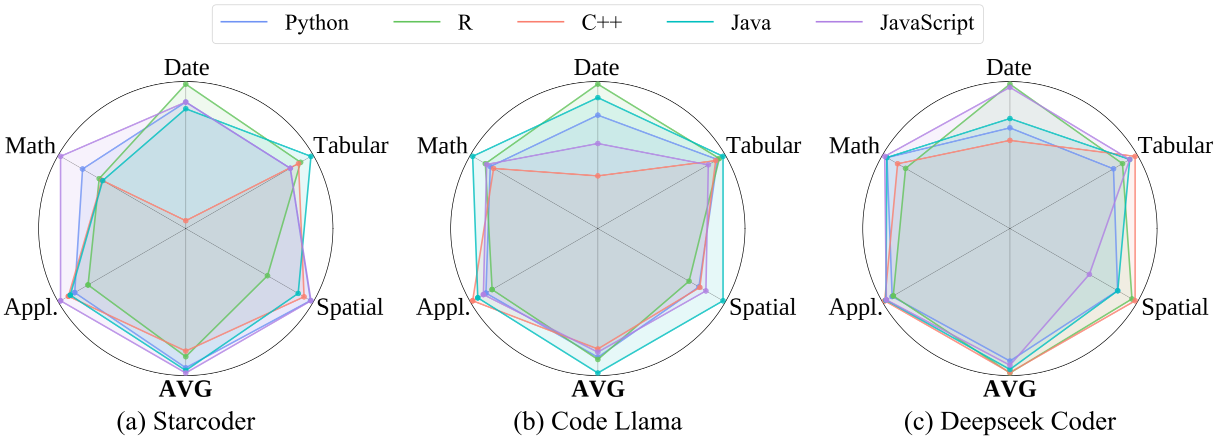


Figure 3: The greedy decoding performance of three models across five tasks in five different PLs. AVG denotes the average performance of a PL across all tasks. Each language performance is expressed as a ratio to the highest-performing language for that specific task. The center of the circle represents 50%. Detailed numerical data are provided in the Table 11 in Appendix A.2.

| Language | Code LLMs | | | | | | ChatGPT | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Appl. | Math | Date | Tabular | Spatial | AVG | Appl. | Math | Date | Tabular | Spatial | AVG |
| Python | 58.51 | 23.62 | 42.37 | 83.00 | 73.87 | 56.27 | 80.75 | 39.74 | 46.61 | **94.63** | 91.70 | 70.69 |
| R | 57.04 | 22.61 | **47.70** | 85.46 | 71.20 | 56.80 | 79.37 | 34.86 | **55.01** | 89.93 | **92.85** | 70.40 |
| C++ | **60.80** | 22.61 | 32.79 | 86.35 | **75.87** | 55.68 | 79.46 | 39.90 | 47.70 | 91.95 | 86.65 | 69.13 |
| Java | 60.11 | 23.75 | 43.81 | **87.92** | 75.82 | **58.28** | 80.63 | **42.65** | 51.22 | 87.92 | 86.70 | 69.82 |
| JavaScript | 60.14 | **24.35** | 42.82 | 83.89 | 71.58 | 56.56 | **81.25** | 36.07 | **55.01** | 92.62 | 90.15 | **71.02** |

Table 1: The performance of Code LLMs and ChatGPT for greedy decoding for five languages on five tasks. Code LLMs are the average results for Starcoder, Code Llama, and Deepseek Coder. **AVG** means the average performance of the language on five tasks. **Bold** denotes the highest performance on the task.

NL LLMs, invoking through the API of gpt-3.5-turbo-0701. By choosing these backbone LLMs with different sizes and characteristics, we can obtain more realistic and credible results.

### 4.5 Inference Details

We combine Chen et al. (2022) and Gao et al. (2023)'s prompt templates for few-shot inference. We fix the questions from the previous work and write code in the respective PLs. The number of questions in each task is shown in Appendix A.1. When sampling for Self-Consistency, we follow Chen et al. (2022) and set $t = 0.4, top\_p = 1$. For a fair comparison with MultiPoT which integrates five languages, we set $k = 5$.

## 5 Results

In this section, we first discover that Python is not the best language for all tasks and all models from the results of greedy decoding. There is no such perfect language. The performance of each PL varies greatly depending on the task and model (Section 5.1). After Self-Consistency, the performance discrepancy still exists. Finally, by integrating multiple languages, MultiPoT significantly outperforms Python. Furthermore, its performance matches or exceeds the best monolingual PoTs in almost all scenarios and achieves improvement on task and model averages (Section 5.2).

### 5.1 Comparison among PLs

**Python is not the optimal language choice.** Figure 3 shows the performance gap between each language and the best-performing language on each task of the three Code LLMs. It illustrates that Python does not achieve the best performance on any of the tasks for any of the Code LLMs. On Deepseek Coder, Python is even the worst on average. Table 1 shows the greedy decoding results of ChatGPT. Although Python performs best on Tabular, it falls short by 2.9% and 8.4% compared to the best PL on Math and Date respectively. The preference for Python among humans may be due to its simple syntax and high readability, but it is a subjective bias that PoT only needs it. Relying on Python leads to a suboptimal outcome.

| | ChatGPT | | | | | | Starcoder | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Appl. | Math | Date | Table | Spatial | **AVG** | Appl. | Math | Date | Table | Spatial | **AVG** |
| Python | 82.31 | 45.76 | 47.70 | 94.63 | 93.60 | 72.80 | 47.04 | 19.69 | 34.96 | 79.19 | 70.00 | 50.18 |
| R | 80.95 | 40.61 | **58.81** | 93.29 | 94.60 | 73.65 | 44.21 | 17.74 | 37.13 | 77.85 | 65.90 | 48.57 |
| C++ | 81.40 | 43.77 | 49.05 | 93.29 | 88.45 | 71.19 | 47.34 | 16.74 | 18.70 | 82.55 | 70.95 | 47.26 |
| Java | 81.79 | 45.33 | 53.39 | 92.62 | 88.80 | 72.39 | 47.97 | 16.76 | 35.23 | 78.52 | 69.50 | 49.60 |
| JavaScript | 82.58 | 40.64 | 56.10 | 96.64 | 93.30 | 73.85 | 48.40 | 19.15 | 36.31 | 80.54 | **72.95** | 51.47 |
| **MultiPoT** | **84.33** | **49.92** | 58.54 | **98.66** | **95.30** | **77.35** | **49.67** | **20.41** | **40.38** | **87.25** | 71.55 | **53.85** |
| | Code Llama | | | | | | Deepseek Coder | | | | | |
| Python | 68.63 | 27.95 | 50.68 | 92.62 | 77.55 | 63.48 | 70.65 | **37.64** | 44.72 | 93.96 | 89.80 | 67.35 |
| R | 66.80 | 26.65 | 58.27 | 93.29 | 79.05 | 64.81 | 69.22 | 33.59 | 53.12 | 93.29 | 92.60 | 68.36 |
| C++ | **71.33** | 24.99 | 43.36 | 93.29 | 80.45 | 62.68 | **72.32** | 33.94 | 39.57 | **95.30** | **93.40** | 66.91 |
| Java | 70.10 | 27.93 | 56.91 | **93.96** | **81.80** | 66.14 | 72.10 | 35.35 | **55.56** | 93.96 | 88.75 | 69.14 |
| JavaScript | 68.97 | 26.16 | 50.41 | 87.25 | 80.35 | 62.63 | 71.89 | 35.60 | 52.57 | 93.29 | 86.10 | 67.89 |
| **MultiPoT** | 71.17 | **27.97** | **58.54** | **93.96** | 79.60 | **66.24** | **72.32** | 37.55 | 54.47 | **95.30** | 91.70 | **70.27** |

Table 2: Self-Consistency and MultiPoT results of four LLMs on five tasks and **AVG**.

However, it is important to note that **there is no one-size-fits-all language**. The gap between PLs is significant when considering each task and model.

**The performance of each PL is task-dependent**. *AVG performance does not fully capture the disparity among languages.* Java and JavaScript performances of Starcoder differ by only 0.41% on AVG, but by 6.71% on Tabular. While the difference between the best and worst PLs of ChatGPT on AVG is less than 2% in Table 1, there are four tasks whose gap among languages exceeds 6%. *Different languages are suitable for different tasks.* Table 1 indicates that, except for C++, all PLs excel in at least one task on ChatGPT. Moreover, on ChatGPT, except for JavaScript, each language also ranks as the least effective in at least one task. *A language that performs exceptionally well in one task might underperform in another.* For instance, R demonstrates superior performance on Date for both Code LLMs and ChatGPT, yet it is the least effective on Appl. and Math.

**The performance of each PL is model-dependent.** *Code LLMs and ChatGPT differ significantly.* The results of three Code LLMs are averaged and compared with ChatGPT in Table 1. It shows that, on Appl., C++ performs best on Code LLMs but ranks second-to-last on ChatGPT; on Math, JavaScript excels on Code LLMs but similarly ranks second-to-last on ChatGPT; and on Spatial, Java ranks second-highest on Code LLMs (with only a 0.05% less than C++) but is second-to-last on ChatGPT. *Even within Code LLMs, disparities between models are evident.* Figure 3 shows that Code Llama has a clear preference for Java, which keeps the top two ranks across all tasks, yet is not observed on the remaining models. On Deepseek Coder, C++ leads on average, whereas it ranks last on the other models. R ranks second on Spatial on Deepseek Coder, but the worst on the other two Code LLMs.

These variations demonstrate that **different PLs exhibit unique strengths and diversity** due to complex factors such as task suitability and model preference. A further error analysis of the experimental results is shown in Appendix A.3.

### 5.2 Comparision between Self-Consistency and MultiPoT

**Self-Consistency does not eliminate performance disparities between PLs**, despite it significantly improving the performance. Table 2 presents the Self-Consistency results. *The inherent strength of different languages persist.* The optimal PL on each scenario is generally consistent with greedy decoding results, except Python emerges as the superior language on Math on all model. *The weaknesses of each language is further amplified.* For example, on Date of Deepseek Coder, C++ already had the lowest performance in greedy decoding, and Self-Consistency increases this gap even more. As a result, C++ shifts from the highest average performance in greedy decoding on Deepseek Coder to the lowest in Self-Consistency, despite remaining the best on Appl., Tabular, and Spatial. *A single language offers limited diversity.* When faced with tasks outside its strength, monolingual samples often make the same mistakes repeatedly, resulting in incorrect answers being chosen through voting.

Different from Self-Consistency relying on a

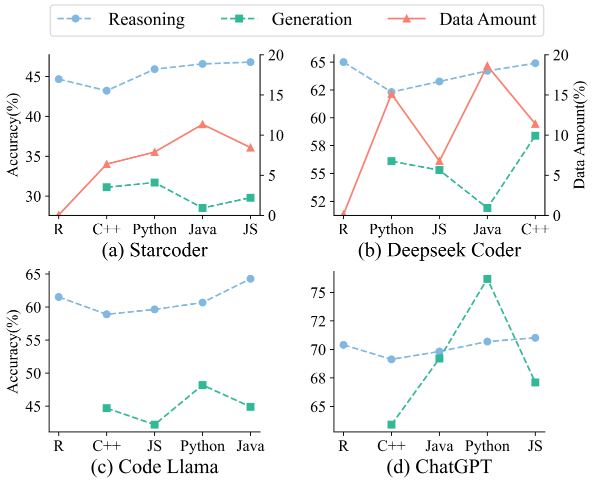


Figure 4: The reasoning ability, code generation ability, and percentage in pre-training data for different languages. Generation lacks data for R. The horizontal coordinates of each model are ranked according to the rise in reasoning performance (excluding R).

single PL, **MutliPoT** integrates multiple PLs. It not only **leverages the distinct strength** of each PL, but also **utilizes their greater diversity** to reduce the probability of repeating the same errors.

**MultiPoT significantly outperforms Python on almost all scenarios**. *It enhances performance in tasks or models where Python is weak.* Across the four models, MultiPoT improves upon Python's performance on Date by at least 15%, and in average (AVG) performance by 4.33% to 7.32%. Furthermore, *MultiPoT also capitalizes on Python's strength.* On Math, where Python excels, MultiPoT also achieves the best results, except in Deepseek Coder, where it slightly trails Python but remains significantly ahead of other languages.

**MultiPoT achieves comparable or superior performance to the best monolingual results across all tasks and models.** *It is task-agnostic.* It surpasses Self-Consistency on four tasks, ranking second on the remaining task, regardless of whether on Code LLMs average (Table 12) or ChatGPT. *MultiPoT is also model-agnostic.* It is the top performer across all LLMs on Tabular. On AVG, MultiPoT outperforms the best monolingual result on all four models. Particularly on ChatGPT and Starcoder, it exhibits an improvement of over 4.6%.

The performance of PLs depends on the task and model. Analyzing the interplay of PL, task, and model in practical applications is challenging. Therefore, MultiPoT is a great choice which has consistently high performance across scenarios.

# 6 Discussion

## 6.1 Reasoning Ability of Different Languages

In Section 5.1, we note that the ranking of the average performance of PL varies on each model. The language distribution in the pre-training data of Starcoder and Deepseek Coder offers insights into whether data amount, defined as the percentage of each language in the pre-training corpus, impacts reasoning capabilities. Moreover, we are interested in examining whether code generation and reasoning of multilingual ability are aligned. The difference between the two tasks is elucidated in Appendix A.4. To assess code generation ability, we utilize the results of each model on the Multilingual HumanEval benchmark, focusing on the four available languages, excluding R due to a lack of evaluation dataset.

**Data distribution influences but does not completely determine reasoning ability.** Figure 4 shows the relative relationships among reasoning performance of C++, Python, and Java are consistent with data distribution on Starcoder. However, R demonstrates unexpectedly strong performance, which has an extremely low percentage in both models. C++ has less data amount than Java on Deepseek Coder, but better reasoning performance. This suggests that there are other factors affecting performance besides data distribution.

**Code generation abilities do not always align with reasoning abilities.** We compare the four languages excluding R in Figure 4. On ChatGPT, the reasoning and code generation abilities of C++, Java, and Python align perfectly. However, an opposite trend is observed in Deepseek Coder's Python, JavaScript, and Java, where the two abilities diverge significantly. It highlights the necessity of testing the reasoning abilities of different PLs.

**Zero-shot reasoning ability shows considerable inconsistency when compared to 3-shot reasoning ability.** Table 3 presents the results of zero-shot and 3-shot experiments using Code Llama 34B on Appl., where AC denotes accuracy, and incorrect outcomes are further classified into Runtime Errors (RE) and Wrong Answers (WA). The results reveal particularly steep declines in R, C++, and JavaScript, largely driven by a significant increase in RE. This suggests that different PLs exhibit varying levels of sensitivity to shot settings. Two prominent error patterns emerge from the zero-shot outputs: (1) LLM frequently generates repetitive comments until reaching the maximum sequence length,

| #Shots | 0 | | | 3 | | |
|---|---|---|---|---|---|---|
| | RE | WA | AC | RE | WA | AC |
| Python | 17.33 | 29.97 | **52.71** | **2.72** | 32.14 | 65.14 |
| R | 55.81 | 15.45 | 28.74 | 2.08 | **34.47** | 63.44 |
| C++ | 59.61 | 14.90 | 25.48 | 0.34 | 30.87 | **68.79** |
| Java | 14.86 | **33.25** | 51.89 | 0.54 | 32.07 | 67.38 |
| JavaScript | **69.26** | 13.77 | 16.96 | 1.74 | 32.41 | 65.84 |

Table 3: Comparison for different PLs under zero-shot (0) and 3-shot (3) demonstrations. **RE** represents Runtime Error. **WA** means Wrong Answer. **AC** represents ACcuracy.

| | StarC. | C. Llama | Deep.C. | GPT |
|---|---|---|---|---|
| Python | 61.03 | 73.23 | 75.80 | 77.62 |
| R | 58.86 | 75.11 | 76.02 | 79.00 |
| C++ | 59.75 | 72.82 | 75.80 | 77.82 |
| Java | 61.32 | 75.62 | 78.06 | 78.08 |
| JavaScript | 62.60 | 74.15 | 76.62 | 77.65 |
| MultiPoT | **64.52** | **75.71** | **78.41** | **83.94** |

Table 4: The average coverage rate on five tasks of Self-Consistency and MultiPoT on each model.

| **Stability Metric** | **Starcoder** | **Deepseek Coder** |
|---|---|---|
| Default | **53.85** | **70.27** |
| Length Short | 53.36 | 69.99 |
| Length Long | 53.16 | 69.76 |
| Random | 53.71 | 69.99 |
| Data Amount Little | 53.18 | 70.20 |
| Data Amount Large | 53.55 | 69.43 |
| $\Delta$ | 0.69 | 0.84 |

Table 5: The performance of MultiPoT with different sorting methods. Length Short/Long represents the ascending/descending order according to the length of PoTs, respectively. $\Delta$ denotes the range of change.

and (2) LLM generates CoT without corresponding executable code. These observations highlight the importance of high-quality, language-specific demonstrations; only with effective demonstrations can the model fully harness the reasoning capabilities of different PLs.

### 6.2 MultiPoT Analysis

**MutliPoT has the highest coverage rate**. Unlike the voting mechanism which requires a majority for the correct answer, the coverage rate is the percentage of questions that have at least one correct answer in five candidate answers in the dataset. For example, if the candidate answers to a question are "(6 5 5 5 7)" and the ground truth is "7", the question is covered. Coverage rate can be considered as an upper bound because this metric represents the proportion of all potentially solvable problems,

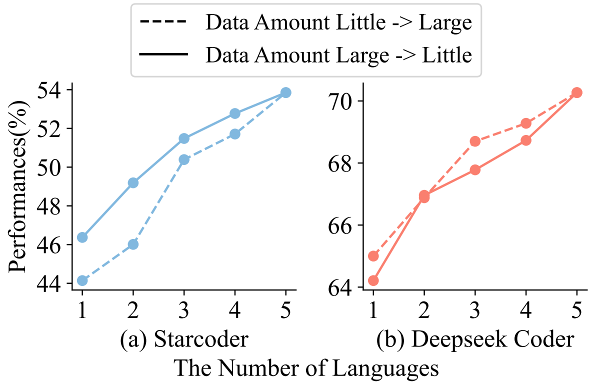


Figure 5: The impact of the number of integrating PLs. We test the different order of adding languages.

assuming there exists a selection mechanism better than the current voting mechanism. Table 4 demonstrates coverage rates on all four models and MultiPoT achieves the highest. The monolingual sampling covers less than the multilingual attempts, highlighting that the strength of different PLs exists. MultiPoT effectively utilizes the strength of different PLs and has the highest upper bound.

**MutliPoT has stable performance**. When results are tied, the top-ranked result is selected. Different sorting methods reflect the stability. Table 5 shows the performance fluctuation. MultiPoT is less than 1% across various sorting criteria, including PoT length, randomness, or data amount from pre-training, compared to the default cumulative probability sorting. This indicates that MultiPoT consistently selects the correct answer directly, with few instances of ties with incorrect answers. This also suggests a lower probability of different PoTs making the same errors.

**More PLs are better**. We investigate the impact of the number of PLs on MultiPoT. On both Starcoder and Deepseek Coder, we incrementally add languages in both ascending and descending order of data amount in Figure 5. The results show that MultiPoT's performance improves with more PLs, regardless of the order. This suggests that

| Model | Method | Appl. | Math | Date | Table | Spatial | AVG |
|---|---|---|---|---|---|---|---|
| Base | Python | 68.63 | 27.95 | 50.68 | 92.62 | 77.55 | 63.48 |
| | MultiPoT | **71.17** | 27.95 | **58.54** | **93.96** | **79.60** | **66.24** |
| Python | Python | 69.54 | **28.46** | 48.24 | 91.28 | 74.65 | 62.43 |
| | MultiPoT | 70.67 | 27.46 | 55.83 | 92.62 | 76.70 | 64.65 |

Table 6: The performance of Python Self-Consistency and MultiPoT on Code Llama Base and Code Llama Python.

| **Type** | **Starcoder** | **ChatGPT** |
|---|---|---|
| All Dynamic | 50.41 | 74.92 |
| Dynamic + Static | 51.87 | 75.77 |

Table 7: The impact of different language type combinations on MultiPoT. All Dynamic indicates that the three languages are all dynamic, and Dynamic+Static indicates a combination of dynamic and static languages.

| | Date | Table | Spatial |
|---|---|---|---|
| CoT | 82.38 | 91.28 | 97.70 |
| PoT | 79.40 | 97.28 | 97.95 |
| MultiPoT | 80.22 | 98.66 | 99.10 |

Table 8: Comparison between Self-Consistency of CoT, PoT (Python), and MultiPoT. CoT results are based on Deepseek LLM v2, while PoT and MultiPoT are based on Deepseek Coder v2.

MultiPoT is highly scalable and performance can be further enhanced by incorporating more PLs.

**More language types are better**. Python, R, and JavaScript are dynamic languages, while C++ and Java are static. To investigate whether a diverse set of language types enhances MultiPoT's performance, we focus on three PLs. On Starcoder and ChatGPT, JavaScript emerges as the highest-performing dynamic language, surpassing Java, which leads between the static languages. Consequently, we integrate JavaScript, Python, and R as All Dynamic and combine Java, Python, and R to represent Dynamic + Static. The results in Table 7 indicate that replacing the higher-performing JavaScript with the lower-performing Java improves performance. This suggests that more language types can provide more diversity to MultiPoT, thereby further enhancing performance.

**MultiPoT also works on Python model**. Our prior experiments with Code LLMs utilize the Base version. However, Code LLMs also have a Python-specific version trained with additional Python corpora. Evaluating MultiPoT on this Python version, as shown in Table 6, we find that Python Self-Consistency improves on Appl. and Math but declines on the other tasks compared to the Base model. Moreover, MultiPoT still outperforms Python Self-Consistency on all tasks except Math, highlighting the adaptability of MultiPoT. Notably, MultiPoT's performance on the Python model is lower across all tasks than on the Base model. This suggests that extensive training on monolingual corpora might diminish the Base model's multilingual abilities on reasoning tasks.

**MultiPoT is better than CoT Self-Consistency.** To compare the performance of CoT and PoT in scenarios where precise mathematical calculations are not required, we conduct experiments on Date, Table, and Spatial—using Deepseek LLM v2 (a 405B MoE LLM) for CoT and Deepseek Coder v2 (continually pretrained from Deepseek LLM v2) for PoT. The results, shown in Table 8, indicate that PoT achieves better Self-Consistency than CoT on Table and Spatial, with MultiPoT further improving performance. On Table, the improvement demonstrates the advantage of PoT in understanding and reasoning over structured data. On Date, however, PoT slightly underperforms CoT, which is primarily due to PoT interpreting the difference between two dates as exclusive, while natural language typically considers it inclusive. Nevertheless, the results suggest that PoT remains valuable in scenarios where precise calculations are unnecessary, and MultiPoT continues to be effective.

## 7 Conclusion

Regarding the reliance on Python in PoT, we conducted extensive experiments across various models and tasks using multiple PLs. Our findings show that Python is not always the best choice; the optimal language depends on the specific task and model. Building on this insight, we introduce MultiPoT, a simple yet effective multilingual integrated method that leverages the strengths and diversity of different PLs. MultiPoT significantly outperforms Python and achieves matches or exceeds performance to the best monolingual outcomes in nearly all scenarios. With its high stability, MultiPoT offers a promising avenue for future research.

## Limitations

Our study is comprehensive, but has certain limitations that we plan to address in future research. Due to computational resource constraints, we confine our experiments to a select number of commonly used programming languages (PLs). While these PLs are representative, they do not encompass the entire spectrum of languages used in programming. Future research could investigate the advantages of incorporating a broader range of programming languages. This may reveal further insights and improve the relevance of our findings.

## Ethical Considerations

Our research utilizes publicly available models and datasets with proper citations and adheres to the usage guidelines of ChatGPT, minimizing the risk of generating toxic content due to the widely-used, non-toxic nature of our datasets and prompts.

## Acknowledge

We gratefully acknowledge the support of the National Natural Science Foundation of China (NSFC) via grant 62236004, 62206078, 62441603 and 62476073 and the support of Du Xiaoman (Beijing) Science Technology Co., Ltd.

## References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

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.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *CoRR*, abs/2211.12588.

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*, 24(240):1–113.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The pile: An 800gb dataset of diverse text for language modeling.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Felix Gimeno, Florent Altché, and Rémi Leblond. 2023. Alphacode 2 technical report. Technical report, AlphaCode Team, Google DeepMind.

GitHub. 2023. The state of open source and ai.

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.

Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14953–14962.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms.

Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. 37:5131–5140.

Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the effectiveness of large language models in detecting security vulnerabilities.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

Rik Koncel-Kedziorski, Michael Krumdick, Viet Lai, Varshini Reddy, Charles Lovering, and Chris Tanner. 2023. Bizbench: A quantitative reasoning benchmark for business and finance. *arXiv preprint arXiv:2311.06602*.

Chengshu Li, Jacky Liang, Fei Xia, Andy Zeng, Sergey Levine, Dorsa Sadigh, Karol Hausman, Xinyun Chen, Li Fei-Fei, and brian ichter. 2023a. Chain of code: Reasoning with a language model-augmented code interpreter. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

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 arXiv:2305.06161*.

Aman Madaan and Amir Yazdanbakhsh. 2022. Text and patterns: For effective chain of thought, it takes two to tango. *arXiv preprint arXiv:2209.07686*.

Shen-Yun Miao, Chao-Chun Liang, and Keh-Yih Su. 2020. A diverse corpus for evaluating and developing english math word problem solvers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 975–984.

Dung Nguyen, Le Nam, Anh Dau, Anh Nguyen, Khanh Nghiem, Jin Guo, and Nghi Bui. 2023. The vault: A comprehensive multilingual dataset for advancing code understanding and generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 4763–4788, Singapore. Association for Computational Linguistics.

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.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are nlp models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2080–2094.

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 arXiv:2308.12950*.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*.

Dingzirui Wang, Longxu Dou, Wenbin Zhang, Junyu Zeng, and Wanxiang Che. 2023a. Exploring equation as a better intermediate meaning representation for numerical reasoning.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. In *ICML*.

Xingyao Wang, Sha Li, and Heng Ji. 2023b. Code4Struct: Code generation for few-shot event structure prediction. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3640–3663, Toronto, Canada. Association for Computational Linguistics.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023c. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '23. ACM.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2023. Large language models as optimizers.

Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi R. Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Yiquan Wang, Heng Ji, and Chengxiang Zhai. 2024. If llm is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents.

Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual code co-evolution using large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 695–707, New York, NY, USA. Association for Computing Machinery.

Liang Zhang, Anwen Hu, Haiyang Xu, Ming Yan, Yichen Xu, Qin Jin, Ji Zhang, and Fei Huang. 2024. Tinychart: Efficient chart understanding with visual token merging and program-of-thoughts learning. *arXiv preprint arXiv:2404.16635*.

# A Appendix

## A.1 Tasks

| Subset | #Original | #Filtered |
|---|---|---|
| Algebra | 1,187 | 1,068 |
| Counting & Probability | 474 | 474 |
| Geometry | 479 | 466 |
| Intermediate Algebra | 903 | 721 |
| Number Theory | 540 | 528 |
| Prealgebra | 871 | 842 |
| Precalculus | 546 | 370 |
| **SUM** | 5,000 | 4,469 |

Table 9: After filtering, the statistics of MATH dataset.

Appl. comprises the GSM8K (Cobbe et al., 2021), SVAMP (Patel et al., 2021), and Asdiv (Miao et al., 2020) datasets. These datasets contain elementary-level math problems set in specific application scenarios, focusing on mathematical abstraction and modeling skills, with relatively low difficulty. Since they are the same type of questions, we merge them into one task. Math, consisting of the transformed MATH (Hendrycks et al., 2021) dataset, whose answers to the problems are expressed using LaTeX. It's too hard to construct prompts in other languages that meet all the requirements, we select those that can be calculated to a single number, excluding problems with interval or formula-based answers. The filtered results are shown in Table 9.

| Task | #Data | #Shots |
|---|---|---|
| Appl. | 4,415 | 3 |
| Math | 4,469 | 3 |
| Date | 369 | 6 |
| Tabular | 149 | 3 |
| Spatial | 2,000 | 3 |

Table 10: Summarization of selected reasoning tasks.

Here are the details of our selected tasks, including the number of questions in each task (#Data) and the number of shots in demonstrations.

## A.2 Additional Data

Table 11 is the raw data of Figure 3, shows the greedy decoding results of each PL of each Code LLMs. Table 12 shows that on the average performance of three Code LLMs, MultiPoT surpasses all Self-Consistency on four tasks, and is only lower slightly than C++ on Spatial.

## A.3 Error Analysis

We further classify incorrect results into Wrong Answer (WA) and Runtime Error (RE), representing cases where the program runs but produces incorrect answers and where the program encounters errors during execution, respectively. Tables 13 to Table 16 show the results for the four models.

It is evident that there are significant differences in the proportion of runtime errors (RE) across different languages and models for each task. Even languages with similar performance exhibit different distributions of errors. For instance, on Appl. of Deepseek Coder, the accuracy difference between Java and JavaScript is less than 0.1%, yet JavaScript has an RE rate of 2.06%, while Java's is only 0.63%. It indicates that the types of errors vary significantly among languages.

A further categorization of the types of RE is conducted. We classify all REs into eight error types. **Redeclaration** represents duplicate naming of variables. **Division by Zero** represents the denominator in the division is zero. **Illegal Output** represents the answer can not be parsed or converted correctly. **Time Limit Error** represents the program runs out of time and sometimes it is due to stack space overflow. **Compile Error** often means there are some syntax error in the program. **Undefined Identifier** includes Undefined Variables and Undefined Functions, which means the variables or functions are not defined before they are used. **Variable Type Error** indicates that the types of variables are mismatched when they are involved in some operations, for example addition or division. Table 17 shows the proportion of different RE types for Deepseek Coder across five tasks and five languages. Table 18 presents the proportion of various RE types for four LLMs on Appl. across all languages. Deepseek Coder and the Appl. task are selected because the languages have the most similar performance on them. The results demonstrate that even in scenarios where languages exhibit similar performance, the proportions of RE differ significantly among languages. For instance, the RE rate on ChatGPT's Appl. of R and C++ differs by only 0.02%, yet Illegal Output account for 82.46% of C++ errors, in comparison to only 24.71% for R. Given that each prompt is accurate, the differing error distributions are attributable to the intrinsic characteristics of the languages, thereby demonstrating their diversity and the non-repetitive nature of their errors.

| | | Appl. | Math | Date | Tabular | Spatial | **AVG** |
|---|---|---|---|---|---|---|---|
| **Starcoder** | Python | 43.06 | 15.78 | 32.79 | 74.50 | 63.55 | 45.94 |
| | R | 40.63 | 14.63 | **34.96** | 77.85 | 52.60 | 44.13 |
| | C++ | 44.21 | 14.43 | 18.43 | 77.18 | 61.90 | 43.23 |
| | Java | 43.87 | 14.39 | 31.98 | **81.21** | 60.40 | 46.37 |
| | JavaScript | **45.64** | **17.30** | 32.79 | 74.50 | **63.65** | **46.78** |
| **Code Llama** | Python | 65.14 | 23.09 | 51.76 | 89.26 | 74.60 | 60.77 |
| | R | 63.44 | 23.58 | **57.99** | 89.93 | 71.35 | 61.26 |
| | C++ | **68.79** | 22.76 | 39.57 | 88.59 | 74.90 | 58.92 |
| | Java | 67.38 | **24.84** | 55.28 | **91.28** | **82.55** | **64.27** |
| | JavaScript | 65.84 | 23.45 | 46.07 | 85.91 | 76.90 | 59.63 |
| **Deepseek Coder** | Python | 67.34 | 32.00 | 42.55 | 85.23 | 84.45 | 62.31 |
| | R | 67.04 | 29.60 | **50.14** | 88.59 | 89.65 | **65.00** |
| | C++ | **69.40** | 30.63 | 40.38 | **93.29** | **90.80** | 64.90 |
| | Java | 69.08 | 32.02 | 44.17 | 91.28 | 84.50 | 64.21 |
| | JavaScript | 68.95 | **32.29** | 49.59 | 91.28 | 74.20 | 63.26 |

Table 11: The greedy decoding results of each PL of each Code LLMs. The detailed numerical data for Figure 3.

| | Appl. | Math | Date | Table | Spatial |
|---|---|---|---|---|---|
| Python | 62.11 | 28.43 | 43.45 | 88.59 | 79.12 |
| R | 60.08 | 25.99 | 49.50 | 88.14 | 79.18 |
| C++ | 63.66 | 25.23 | 33.88 | 90.38 | **81.60** |
| Java | 63.39 | 26.68 | 49.23 | 88.81 | 80.02 |
| JavaScript | 63.09 | 26.97 | 46.43 | 87.02 | 79.80 |
| MultiPoT | **64.39** | **28.64** | **51.13** | **92.17** | 80.95 |

Table 12: The average performance of three Code LLMs for Self-Consistency and MultiPoT in each task.

Upon further analysis of generated contents, several common failure patterns emerge:

- **Date Calculation**: PoTs often misinterpret the difference between two dates as exclusive, contrary to natural language conventions, where the interval is typically inclusive. Nevertheless, R demonstrates comparable performance to CoT on Date, indicating its potential for temporal reasoning tasks.

- **Output Content**: PoTs frequently respond to yes/no questions by outputting attributes instead of directly answering the question. For example, when asked, ‘On the desk, there is a teal pen and a yellow textbook. Is the textbook yellow?’, the correct answer is ‘yes’, but PoT might respond with ‘yellow’.

- **Demonstration Constraint**: Demonstrations are less restrictive for JavaScript, as it tends to output extra information beyond what is required, including descriptive sentences and variables, even when only the final answer is needed.

- **Syntax Preference**: C++ and Java tend to leverage language-specific constructs like for-loops, often reaching correct solutions in pattern recognition problems. In contrast, other languages that attempt step-by-step calculations may add or omit steps, leading to errors.

- **Resource Constraints**: Low-resource languages like R may cause the model to call non-existent packages, particularly in complex mathematical problems that rely on external tools.

### A.4 Difference Between Code Generation and PoT

Figure 4 illustrates that performance in code generation does not fully align with that in reasoning tasks.

Although both tasks involve generating code to solve problems, their objectives differ. The code generation task assesses the LLM’s ability to assist development in an engineering environment, covering real-world engineering issues. For example, consider the following problems: ’Given a positive floating point number, return its decimal part’ and ’Given a list of integers, return a tuple containing the sum and product of all the integers in the list.’ Although these problems require some reasoning,

| | Appl. | | | Math | | | Date | | | Tabular | | | Spatial | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AC | WA | RE | AC | WA | RE | AC | WA | RE | AC | WA | RE | AC | WA | RE |
| Python | 67.34 | 30.06 | 2.60 | 32.00 | 48.00 | 20.00 | 42.55 | 57.18 | 0.27 | 85.23 | 6.04 | 8.72 | 84.45 | 12.65 | 2.90 |
| R | 67.04 | 31.51 | 1.45 | 29.60 | 53.79 | 16.60 | 50.14 | 43.63 | 6.23 | 88.59 | 8.05 | 3.36 | 89.65 | 6.65 | 3.70 |
| C++ | 69.40 | 30.15 | 0.45 | 30.63 | 61.74 | 7.63 | 40.38 | 59.62 | 0.00 | 93.29 | 6.71 | 0.00 | 90.80 | 8.95 | 0.25 |
| Java | 69.08 | 30.28 | 0.63 | 32.02 | 60.80 | 7.18 | 44.17 | 47.15 | 8.67 | 91.28 | 7.38 | 1.34 | 84.50 | 14.65 | 0.85 |
| JavaScript | 68.95 | 28.99 | 2.06 | 32.29 | 55.36 | 12.35 | 49.59 | 49.86 | 0.54 | 91.28 | 7.38 | 1.34 | 74.20 | 19.45 | 6.35 |

Table 13: The execution result of programs generated from Deepseek Coder for five languages on five tasks. **AC** represents **Accept**, which means the program can generate a correct answer. **Wrong** means the answer is not right. **RE** represents **Runtime Error**, which means the program does not execute normally.

| | Appl. | | | Math | | | Date | | | Tabular | | | Spatial | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AC | WA | RE | AC | WA | RE | AC | WA | RE | AC | WA | RE | AC | WA | RE |
| Python | 43.06 | 53.70 | 3.24 | 15.78 | 60.66 | 23.56 | 32.79 | 63.41 | 3.79 | 74.50 | 14.09 | 11.41 | 63.55 | 29.65 | 6.80 |
| R | 40.63 | 57.94 | 1.43 | 14.63 | 66.08 | 19.29 | 34.96 | 55.83 | 9.21 | 77.85 | 19.46 | 2.68 | 52.60 | 28.65 | 18.75 |
| C++ | 44.21 | 54.74 | 1.04 | 14.43 | 71.81 | 13.76 | 18.43 | 81.57 | 0.00 | 77.18 | 18.12 | 4.70 | 61.90 | 37.75 | 0.35 |
| Java | 43.87 | 54.65 | 1.47 | 14.39 | 74.71 | 10.90 | 31.98 | 61.52 | 6.50 | 81.21 | 17.45 | 1.34 | 60.40 | 31.80 | 7.80 |
| JavaScript | 45.64 | 52.21 | 2.15 | 17.30 | 66.68 | 16.02 | 32.79 | 67.21 | 0.00 | 74.50 | 24.16 | 1.34 | 63.65 | 30.10 | 6.25 |

Table 14: The execution result of programs generated from Starcoder.

the focus is primarily on language comprehension and engineering skills.

In contrast, reasoning tasks aim to test the LLM's logical reasoning abilities. The generated code acts as a carrier of logic and facilitates the use of tools, such as more precise calculations, dictionaries for storing and retrieving attribute information, or calendars to aid in date reasoning. Reasoning tasks focus on a subset of a programming language's capabilities, rather than its entire spectrum in engineering practice.

Therefore, although there is some overlap between code generation and reasoning tasks, they are not entirely the same. This is why there is only partial consistency between the two tasks in Figure 4 and highlights the necessity of testing different programming languages in reasoning tasks..

| | Appl. | | | Math | | | Date | | | Tabular | | | Spatial | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AC | WA | RE | AC | WA | RE | AC | WA | RE | AC | WA | RE | AC | WA | RE |
| Python | 65.14 | 32.14 | 2.72 | 23.09 | 57.04 | 19.87 | 51.76 | 48.24 | 0.00 | 89.26 | 8.72 | 2.01 | 73.60 | 18.85 | 7.55 |
| R | 63.44 | 34.47 | 2.08 | 23.58 | 61.42 | 14.99 | 57.99 | 41.73 | 0.27 | 89.93 | 9.40 | 0.67 | 71.35 | 24.00 | 4.65 |
| C++ | 68.79 | 30.87 | 0.34 | 22.76 | 71.69 | 5.55 | 39.57 | 59.89 | 0.54 | 88.59 | 10.74 | 0.67 | 74.90 | 23.80 | 1.30 |
| Java | 67.38 | 32.07 | 0.54 | 24.84 | 68.20 | 6.96 | 55.28 | 38.75 | 5.96 | 91.28 | 6.71 | 2.01 | 82.55 | 17.05 | 0.40 |
| JavaScript | 65.84 | 32.41 | 1.74 | 23.45 | 67.69 | 8.86 | 46.07 | 53.39 | 0.54 | 85.91 | 12.75 | 1.34 | 76.90 | 21.50 | 1.60 |

Table 15: The execution result of programs generated from Code Llama.

| | Appl. | | | Math | | | Date | | | Tabular | | | Spatial | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AC | WA | RE | AC | WA | RE | AC | WA | RE | AC | WA | RE | AC | WA | RE |
| Python | 80.75 | 15.61 | 3.65 | 39.74 | 22.76 | 37.50 | 46.61 | 52.85 | 0.54 | 94.63 | 4.70 | 0.67 | 91.70 | 8.00 | 0.30 |
| R | 79.37 | 16.78 | 3.85 | 34.86 | 25.53 | 39.61 | 55.01 | 42.82 | 2.17 | 89.93 | 7.38 | 2.68 | 92.85 | 5.75 | 1.40 |
| C++ | 79.46 | 16.67 | 3.87 | 39.90 | 39.94 | 20.16 | 47.70 | 50.95 | 1.36 | 91.95 | 4.03 | 4.03 | 86.65 | 12.20 | 1.15 |
| Java | 80.63 | 16.44 | 2.92 | 42.65 | 41.96 | 15.39 | 51.22 | 40.92 | 7.86 | 87.92 | 6.71 | 5.37 | 86.30 | 11.00 | 2.70 |
| JavaScript | 81.25 | 15.24 | 3.51 | 36.07 | 24.23 | 39.70 | 55.01 | 44.17 | 0.81 | 92.62 | 4.70 | 2.68 | 90.15 | 9.70 | 0.15 |

Table 16: The execution result of programs generated from ChatGPT.

| Task | Language | Redecl aration | Division by Zero | Illegal Output | Time Limit Error | Compile Error | Undefined Identifier | Variable Type Error | Other Error |
|---|---|---|---|---|---|---|---|---|---|
| **Appl.** | Python | - | - | 61.74 | 2.61 | 9.57 | 23.48 | 2.61 | - |
| | R | - | - | 32.81 | 4.69 | 39.06 | 23.44 | - | - |
| | C++ | 60.00 | - | 15.00 | 10.00 | 5.00 | 5.00 | - | 5.00 |
| | Java | 46.43 | 7.14 | - | 3.57 | 14.29 | 3.57 | 25.00 | - |
| | JavaScript | 5.49 | - | 84.62 | 2.20 | 2.20 | 5.49 | - | - |
| **Math** | Python | - | 2.57 | 19.91 | 31.21 | 4.70 | 31.1 | 7.61 | 2.91 |
| | R | - | - | 20.22 | 10.65 | 10.24 | 38.27 | 1.35 | 19.27 |
| | C++ | 2.93 | - | 17.60 | 28.15 | 21.11 | 7.04 | 4.40 | 18.77 |
| | Java | 1.25 | 3.12 | 16.20 | 28.97 | 16.51 | 8.72 | 3.12 | 22.12 |
| | JavaScript | 1.09 | - | 23.01 | 22.64 | 6.34 | 32.43 | - | 14.49 |
| **Date** | Python | - | - | - | - | - | 100 | - | - |
| | R | - | - | - | 95.65 | 4.35 | - | - | - |
| | C++ | - | - | - | - | - | - | - | - |
| | Java | - | - | - | - | 3.12 | 56.25 | - | 40.62 |
| | JavaScript | 50.00 | - | - | - | - | 50.00 | - | - |
| **Tabular** | Python | - | - | - | - | 84.62 | - | 7.69 | 7.69 |
| | R | - | - | - | - | - | - | 20.00 | 80.00 |
| | C++ | - | - | - | - | - | - | - | - |
| | Java | - | - | - | - | 50.00 | - | - | 50.00 |
| | JavaScript | - | - | - | - | - | 100 | - | - |
| **Spatial** | Python | - | - | - | - | 1.72 | 1.72 | 96.55 | - |
| | R | - | - | - | - | - | - | 22.97 | 77.03 |
| | C++ | - | - | 20.00 | - | 80.00 | - | - | - |
| | Java | - | - | - | - | 5.88 | - | 17.65 | 76.47 |
| | JavaScript | - | - | - | - | 0.79 | 96.85 | - | 2.36 |

Table 17: Runtime Error concrete analysis for five languages on five tasks of Deepseek Coder.

| Model | Language | Redecl aration | Division by Zero | Illegal Output | Time Limit Error | Compile Error | Undefined Identifier | Variable Type Error | Other Error |
|---|---|---|---|---|---|---|---|---|---|
| **Starcoder** | Python | - | - | 69.93 | 1.40 | 3.50 | 17.48 | 1.40 | - |
| | R | - | - | 38.10 | 1.59 | 23.81 | 34.92 | 1.59 | - |
| | C++ | 28.26 | - | 8.70 | 17.39 | 17.39 | 8.70 | - | 19.57 |
| | Java | 6.15 | 3.08 | 3.08 | 3.08 | 24.62 | 3.08 | 56.92 | - |
| | JavaScript | 29.47 | - | 53.68 | 3.16 | 2.11 | 9.47 | - | 2.11 |
| **Code Llama** | Python | - | - | 49.17 | 1.67 | 6.67 | 39.17 | 1.67 | 1.67 |
| | R | - | - | 36.96 | 3.26 | 4.35 | 51.09 | 1.09 | 3.26 |
| | C++ | 13.33 | - | 6.67 | 6.67 | 20.00 | 33.33 | - | 20.00 |
| | Java | 8.33 | - | 4.17 | - | 12.50 | 8.33 | 58.33 | 8.33 |
| | JavaScript | 9.09 | - | 68.83 | 1.30 | 2.60 | 14.29 | - | 3.90 |
| **Deepseek Coder** | Python | - | - | 61.74 | 2.61 | 9.57 | 23.48 | 2.61 | - |
| | R | - | - | 32.81 | 4.69 | 39.06 | 23.44 | - | - |
| | C++ | 60.00 | - | 15.00 | 10.00 | 5.00 | 5.00 | - | 5.00 |
| | Java | 46.43 | 7.14 | - | 3.57 | 14.29 | 3.57 | 25.00 | - |
| | JavaScript | 5.49 | - | 84.62 | 2.20 | 2.20 | 5.49 | - | - |
| **ChatGPT** | Python | - | - | 51.55 | 0.62 | 10.56 | 35.40 | 1.24 | 0.62 |
| | R | - | - | 24.71 | 0.59 | 23.53 | 43.53 | 1.18 | 1.76 |
| | C++ | - | 0.58 | 82.46 | 4.68 | - | 8.19 | - | 4.09 |
| | Java | 0.78 | 2.33 | 67.44 | 2.33 | 1.55 | 9.3 | 13.95 | 2.33 |
| | JavaScript | 2.58 | - | 50.32 | 1.29 | 0.65 | 36.13 | - | 9.03 |

Table 18: Runtime Error concrete analysis for five languages on Appl. of four LLMs.

### A.5 Prompts

Here are our multilingual prompts. We show prompts of Tabular(3-shots) as an example and prompts for other tasks are in the released code.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
We now add a penguin to the table:
James, 12, 90, 12
How many penguins are less than 8 years old?

Use the Python program to solve the problem. The reasoning progress is clearly shown in the program.

# Define the table of penguins with their attributes
penguins = [
    {"name": "Louis", "age": 7, "height": 50, "weight": 11},
    {"name": "Bernard", "age": 5, "height": 80, "weight": 13},
    {"name": "Vincent", "age": 9, "height": 60, "weight": 11},
    {"name": "Gwen", "age": 8, "height": 70, "weight": 15},
    {"name": "James", "age": 12, "height": 90, "weight": 12}
]

# Count the number of penguins that are less than 8 years old
penguins_less_than_8_years = sum(1 for penguin in penguins if penguin["age"] < 8)

ans = penguins_less_than_8_years
```

Figure 6: Python Prompt of the first question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
Which is the youngest penguin?

Use the Python program to solve the problem. The reasoning progress is clearly shown in the program.

# Define the data for the penguins
penguins = [
    {"name": "Louis", "age": 7, "height": 50, "weight": 11},
    {"name": "Bernard", "age": 5, "height": 80, "weight": 13},
    {"name": "Vincent", "age": 9, "height": 60, "weight": 11},
    {"name": "Gwen", "age": 8, "height": 70, "weight": 15}
]

# Find the youngest penguin
youngest_penguin = min(penguins, key=lambda p: p["age"])
ans = youngest_penguin["name"]
```

Figure 7: Python Prompt of the second question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
What is the name of the second penguin sorted by alphabetic order?

Use the Python program to solve the problem. The reasoning progress is clearly shown in the program.

# Creating a list of dictionaries to represent each penguin and their attributes
penguins = [
    {"name": "Louis", "age": 7, "height": 50, "weight": 11},
    {"name": "Bernard", "age": 5, "height": 80, "weight": 13},
    {"name": "Vincent", "age": 9, "height": 60, "weight": 11},
    {"name": "Gwen", "age": 8, "height": 70, "weight": 15}
]

# Sorting the list of penguins by their names in alphabetical order
sorted_penguins = sorted(penguins, key=lambda p: p["name"])

# Extracting the name of the second penguin in the sorted list
ans = sorted_penguins[1]["name"]
```

Figure 8: Python Prompt of the third question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
We now add a penguin to the table:
James, 12, 90, 12
How many penguins are less than 8 years old?

Use the C++ program to solve the problem. The reasoning progress is clearly shown in the program.

#include <iostream>
#include <vector>

using namespace std;

// Define a structure for storing penguin data
struct Penguin {
    string name;
    int age;
    int height;
    int weight;
};

int main() {
    // Create a vector of Penguin structures
    vector<Penguin> penguins = {
        {"Louis", 7, 50, 11},
        {"Bernard", 5, 80, 13},
        {"Vincent", 9, 60, 11},
        {"Gwen", 8, 70, 15},
        {"James", 12, 90, 12}
    };

    // Count penguins less than 8 years old
    int count = 0;
    for (const auto& penguin : penguins) {
        if (penguin.age < 8) {
            count++;
        }
    }
    int ans = count;
    // Print the result
    cout << ans << endl;

    return 0;
}
```

Figure 9: C++ Prompt of the first question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
Which is the youngest penguin?

Use the C++ program to solve the problem. The reasoning progress is clearly shown in the program.

#include <iostream>
#include <string>
#include <vector>

using namespace std;

struct Penguin {
    string name;
    int age;
    int height;
    int weight;
};

int main() {
    // Create a list of penguins
    vector<Penguin> penguins = {
        {"Louis", 7, 50, 11},
        {"Bernard", 5, 80, 13},
        {"Vincent", 9, 60, 11},
        {"Gwen", 8, 70, 15}
    };

    // Initialize variables to track the youngest penguin
    Penguin youngest = penguins[0];

    // Iterate through the list to find the youngest penguin
    for (const Penguin& penguin : penguins) {
        if (penguin.age < youngest.age) {
            youngest = penguin;
        }
    }

    string ans = youngest.name;
    // Output the name of the youngest penguin
    cout << ans << endl;

    return 0;
}
```

Figure 10: C++ Prompt of the second question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
What is the name of the second penguin sorted by alphabetic order?

Use the C++ program to solve the problem. The reasoning progress is clearly shown in the program.

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct Penguin {
    string name;
    int age;
    int height;
    int weight;
};

int main() {
    // Creating a list of penguins
    vector<Penguin> penguins = {
        {"Louis", 7, 50, 11},
        {"Bernard", 5, 80, 13},
        {"Vincent", 9, 60, 11},
        {"Gwen", 8, 70, 15}
    };

    // Sorting the penguins by name using a simpler lambda function
    sort(penguins.begin(), penguins.end(), [](const Penguin &a, const Penguin &b) {
        return a.name < b.name;
    });

    // Displaying the name of the second penguin after sorting
    string ans = penguins[1].name;
    cout << ans << endl;

    return 0;
}
```

Figure 11: C++ Prompt of the third question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
We now add a penguin to the table:
James, 12, 90, 12
How many penguins are less than 8 years old?

Use the Java program to solve the problem. The reasoning progress is clearly shown in the program.

import java.util.ArrayList;
import java.util.List;

public class PenguinAgeCounter {
    public static void main(String[] args) {
        List<Penguin> penguins = new ArrayList<>();

        // Add penguins to the list
        penguins.add(new Penguin("Louis", 7, 50, 11));
        penguins.add(new Penguin("Bernard", 5, 80, 13));
        penguins.add(new Penguin("Vincent", 9, 60, 11));
        penguins.add(new Penguin("Gwen", 8, 70, 15));
        penguins.add(new Penguin("James", 12, 90, 12));

        // Count the number of penguins less than 8 years old
        int count = 0;
        for (Penguin penguin : penguins) {
            if (penguin.age < 8) {
                count++;
            }
        }
        int ans = count;
        System.out.println(ans);
    }
}

class Penguin {
    String name;
    int age;
    int height; // in cm
    int weight; // in kg

    public Penguin(String name, int age, int height, int weight) {
        this.name = name;
        this.age = age;
        this.height = height;
        this.weight = weight;
    }
}
```

Figure 12: Java Prompt of the first question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
Which is the youngest penguin?

Use the Java program to solve the problem. The reasoning progress is clearly shown in the program.

import java.util.ArrayList;
import java.util.List;

public class PenguinAgeFinder {
    public static void main(String[] args) {
        // Create a list of Penguin objects
        List<Penguin> penguins = new ArrayList<>();
        penguins.add(new Penguin("Louis", 7, 50, 11));
        penguins.add(new Penguin("Bernard", 5, 80, 13));
        penguins.add(new Penguin("Vincent", 9, 60, 11));
        penguins.add(new Penguin("Gwen", 8, 70, 15));

        // Call the method to find the youngest penguin
        Penguin youngest = findYoungestPenguin(penguins);

        // Print the name of the youngest penguin
        String ans = youngest.name;
        System.out.println(ans);
    }

    // Method to find the youngest penguin
    public static Penguin findYoungestPenguin(List<Penguin> penguins) {
        Penguin youngest = penguins.get(0);
        for (Penguin penguin : penguins) {
            if (penguin.age < youngest.age) {
                youngest = penguin;
            }
        }
        return youngest;
    }
}

// Define the Penguin class with relevant attributes
class Penguin {
    String name;
    int age;
    int height; // in cm
    int weight; // in kg

    // Constructor to initialize a Penguin object
    Penguin(String name, int age, int height, int weight) {
        this.name = name;
        this.age = age;
        this.height = height;
        this.weight = weight;
    }
}
```

Figure 13: Java Prompt of the second question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
What is the name of the second penguin sorted by alphabetic order?

Use the Java program to solve the problem. The reasoning progress is clearly shown in the program.

import java.util.*;

public class Main {
    public static void main(String[] args) {
        // Step 2: Create Penguin objects
        Penguin louis = new Penguin("Louis", 7, 50, 11);
        Penguin bernard = new Penguin("Bernard", 5, 80, 13);
        Penguin vincent = new Penguin("Vincent", 9, 60, 11);
        Penguin gwen = new Penguin("Gwen", 8, 70, 15);

        // Step 3: Add Penguins to a list
        List<Penguin> penguins = new ArrayList<>();
        penguins.add(louis);
        penguins.add(bernard);
        penguins.add(vincent);
        penguins.add(gwen);

        // Step 4: Sort the list by name
        Collections.sort(penguins, Comparator.comparing(Penguin::getName));

        // Step 5: Find and print the name of the second penguin
        Penguin secondPenguin = penguins.get(1); // Lists are zero-indexed
        String ans = secondPenguin.getName();
        System.out.println(ans);
    }
}

// Step 1: Define the Penguin class
class Penguin {
    private String name;
    private int age;
    private int height; // in cm
    private int weight; // in kg

    // Constructor
    public Penguin(String name, int age, int height, int weight) {
        this.name = name;
        this.age = age;
        this.height = height;
        this.weight = weight;
    }

    // Getters
    public String getName() {
        return name;
    }
}
```

Figure 14: Java Prompt of the third question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
We now add a penguin to the table:
James, 12, 90, 12
How many penguins are less than 8 years old?

Use the Javascript program to solve the problem. The reasoning progress is clearly shown in the program.

// Step 1: Create an array of penguin objects
let penguins = [
    { name: "Louis", age: 7, height: 50, weight: 11 },
    { name: "Bernard", age: 5, height: 80, weight: 13 },
    { name: "Vincent", age: 9, height: 60, weight: 11 },
    { name: "Gwen", age: 8, height: 70, weight: 15 }
];

// Step 2: Add James to the array
penguins.push({ name: "James", age: 12, height: 90, weight: 12 });

// Step 3: Initialize a counter
let count = 0;

// Step 4: Iterate over the array
for (let penguin of penguins) {
    // Step 5: Check if the penguin's age is less than 8
    if (penguin.age < 8) {
        count++;
    }
}

let ans = count;
```

Figure 15: JavaScript Prompt of the first question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
Which is the youngest penguin?

Use the Javascript program to solve the problem. The reasoning progress is clearly shown in the program.

// Step 1: Create an array of penguins
const penguins = [
    { name: "Louis", age: 7, height: 50, weight: 11 },
    { name: "Bernard", age: 5, height: 80, weight: 13 },
    { name: "Vincent", age: 9, height: 60, weight: 11 },
    { name: "Gwen", age: 8, height: 70, weight: 15 }
];

// Step 2: Find the youngest penguin
let youngest = penguins[0];

for (let i = 1; i < penguins.length; i++) {
    if (penguins[i].age < youngest.age) {
        youngest = penguins[i];
    }
}

ans = youngest.name;
```

Figure 16: JavaScript Prompt of the second question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
What is the name of the second penguin sorted by alphabetic order?

Use the Javascript program to solve the problem. The reasoning progress is clearly shown in the program.

// Define the array of penguins
let penguins = [
    { name: "Louis", age: 7, height: 50, weight: 11 },
    { name: "Bernard", age: 5, height: 80, weight: 13 },
    { name: "Vincent", age: 9, height: 60, weight: 11 },
    { name: "Gwen", age: 8, height: 70, weight: 15 }
];

// Sort the penguins array by the name property
penguins.sort((a, b) => {
    if (a.name < b.name) {
        return -1;
    }
    if (a.name > b.name) {
        return 1;
    }
    return 0;
});

// The name of the second penguin after sorting
let ans = penguins[1].name;
```

Figure 17: JavaScript Prompt of the third question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
We now add a penguin to the table:
James, 12, 90, 12
How many penguins are less than 8 years old?

Use the R program to solve the problem. The reasoning progress is clearly shown in the program.

# Create the data frame
penguins <- data.frame(
  name = c("Louis", "Bernard", "Vincent", "Gwen", "James"),
  age = c(7, 5, 9, 8, 12),
  height = c(50, 80, 60, 70, 90),
  weight = c(11, 13, 11, 15, 12)
)

# Filter and count the penguins younger than 8 years
num_penguins_younger_than_8 <- nrow(subset(penguins, age < 8))

ans = num_penguins_younger_than_8
```

Figure 18: R Prompt of the first question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
Which is the youngest penguin?

Use the R program to solve the problem. The reasoning progress is clearly shown in the program.

# Create a data frame representing the penguins
penguins <- data.frame(
  name = c("Louis", "Bernard", "Vincent", "Gwen"),
  age = c(7, 5, 9, 8),
  height_cm = c(50, 80, 60, 70),
  weight_kg = c(11, 13, 11, 15)
)

# Find the youngest penguin by locating the minimum age
youngest_index <- which.min(penguins$age)

ans = penguins$name[youngest_index]
```

Figure 19: R Prompt of the second question.

```
Question: Here is a table where the first line is a header and each subsequent line is a penguin:
name, age, height (cm), weight (kg)
Louis, 7, 50, 11
Bernard, 5, 80, 13
Vincent, 9, 60, 11
Gwen, 8, 70, 15
For example: the age of Louis is 7, the weight of Gwen is 15 kg, the height of Bernard is 80 cm.
What is the name of the second penguin sorted by alphabetic order?

Use the R program to solve the problem. The reasoning progress is clearly shown in the program.

# Create a data frame with the penguins' information
penguins <- data.frame(
  name = c("Louis", "Bernard", "Vincent", "Gwen"),
  age = c(7, 5, 9, 8),
  height = c(50, 80, 60, 70),
  weight = c(11, 13, 11, 15)
)

# Sort the data frame by the 'name' column
sorted_penguins <- penguins[order(penguins$name),]

# Extract the name of the second penguin in the sorted list
ans <- sorted_penguins$name[2]
```

Figure 20: R Prompt of the third question.