Enriching LLMs with non-parametric information using RAG

Jan Henrik Bertrand[1], *André Anan* Gilbert[1], *Felix* Noll[1], and *Marc* Grün[1]

[1] *DHBW Mannheim, Wirtschaftsinformatik - Data Science, Coblitzallee 1-9, 68163 Mannheim, Germany*

Abstract. Today, Large Language Models (LLMs) show fascinating abilities when it comes to generating code. However, they have a major shortcoming: they learn coding by sheer repetition and develop their zero-shot capabilities through studying code samples. This prevents the model from scaling beyond what is included in the training data as well as using versions of packages/libraries that emerged after the model was trained. Hence, we propose a Retriever Augmented Generation (RAG) framework that accesses a vector database with the encoded documentation of every function of a package in its latest version. On request, the agent matches the user input query to the function that is most likely needed in order to generate the code for achieving what the user wants to do. This enables up-to-date knowledge of Python libraries such as pandas without expensive retraining of the LLM. Through automated experiments employing various configurations and test cases, our study demonstrates the superior performance of RAG (+4%) and CoALA (+6%) over the absence of RAG in diverse coding scenarios. The evaluation, spanning different prompting styles and scenarios, provides valuable insights into the strengths and trade-offs of various configurations, contributing to informed decisions in deploying AI agents for code generation.

1 Introduction

LLMs[1] demonstrate impressive capabilities on many different tasks, sometimes showing almost human-like intelligence [1]. However, unlike humans, their immense knowledge is frozen after the training process, i.e., they do not continue to learn throughout their usage. As state-of-the-art models have billions of parameters [2, p. 4], the retraining process is extremely costly in terms of data and computational resources [1]. Thus, extending a model's knowledge without frequent retraining or fine-tuning poses a major opportunity. We propose using RAG[2] in order to enrich the model with additional information in a simple, agile and affordable way.

RAG augments the parametric memory of the LLM with a non-parametric memory [1]. This non-parametric memory can be Knowledge Bases (KBs) of any form containing any information that the user wants the model to have access to, often indexed by vector databases.

2 Related Work

Lewis et al. have initially introduced RAG in a paper from 2020 and proposed to use the method for knowledge intensive NLP tasks [3]. Since then it has been applied and enhanced in multiple case studies [1, 4, 5].

RAG has led to improvement of LLMs in several tasks, including Question Answering that required multiple "hops" [6], specific text generation [7] and on zero-shot slot filling [8].

[1] Large Language Models

[2] Retrieval Augmented Generation

3 Implementation

ReAct[3], which we use to prompt GPT-4, is an innovative technique utilizing LLMs to generate reasoning traces and task-specific actions dynamically. The ReAct prompting technique combines reasoning and acting, facilitating the model's decision-making by integrating diverse information like current objectives, commonsense knowledge, observations, progress tracking, and exception handling. This approach is intuitive, adaptable to various tasks, and excels in decision-making scenarios, outperforming existing methods while maintaining human interpretability and controllability [9, p. 1].

We've implemented a streamlined version of RAG by leveraging FAISS [10] for efficient similarity search. The pandas documentation serves as our indexed dataset, allowing the retrieval system to swiftly identify the nearest neighbors to a user query. This grounding is seamlessly embedded into the prompt, allowing the LLM to generate contextually rich and highly relevant outputs. In addition to the described RAG framework, we implement an extended variant of RAG, the CoALA framework described in the following.

Cognitive Architectures for Language Agents (CoALA), presents a framework where LLMs serve as the central element within a broader cognitive architecture. The framework organizes language agents around external and internal actions, where external actions engage with the environment and internal actions interact with memory. The memory consists of both a short-term memory and a long-term memory, where the possible actions in the

[3] Reasoning + Acting

decision-making cycle are: (1) retrieval from long-term memory, (2) reasoning using the short-term memory and (3) learning by writing back into the long-term memory [10, p. 8]. Notably, CoALA incorporates decades of cognitive architecture research, introducing "reasoning" actions that dynamically generate new knowledge and heuristics, supplanting traditional hand-written rules in cognitive architectures [11, pp. 7-9].

In our CoALA implementation, we prioritize practicality by omitting the need for the AI agent to modify its codebase. Instead, we focus on efficiency and relevance by utilizing a complementary FAISS index. This index is created by combining user input and generated code output, which includes a combination of hand-picked question and answer pairs (simulates human feedback from previous conversations), as well as agent-generated ones. The agent can use this in addition to the documentation memory RAG introduced, representing an episodic memory.

Beyond that, LLMs have immense potential when equipped with tools to navigate their environments effectively. Rather than supplying additional context via RAG on every user request, which can escalate costs, we also gave LLMs the option to autonomously determine the need for extra information to address user queries more efficiently.

4 Experimental Setting

In the evaluation of AI agents, the baseline AI agent serves as a fundamental reference point, equipped with GPT-4 but lacking access to RAG. This foundational setup provides a standard for comparison against more advanced architectures. The evaluation process involves an examination of AI agents that have RAG and CoALA capabilities. The primary focus is on assessing the extent to which the incorporation of RAG enhances the performance of LLMs. By comparing the outputs of the baseline AI agent with those empowered by RAG and CoALA, the evaluation aims to quantify the improvement in output quality achieved through the utilization of retrieved content, shedding light on the effectiveness of these advanced models in augmenting the capabilities of code generation.

We evaluate AI agents based on three key dimensions: accuracy, time taken, and cost. Accuracy is deemed a fundamental metric crucial for assessing the precision and correctness of the generated code. Importantly, our evaluation prioritizes the functional equivalence of code outputs rather than their specific syntactic differences. This approach ensures that even if the generated code differs in structure, as long as the outcome remains consistent with the intended functionality, it contributes positively to the accuracy metric. Simultaneously, the time taken to solve a given task is a critical aspect of our assessment, reflecting the efficiency and responsiveness of the AI agent. We consider the duration from user input to code generation. Additionally, the cost factor in $ is measured which consists of input and output tokens.

The data set for the evaluation consists of 20 test cases, each of them containing an ask for a data transformation with pandas and a correct function (expected result). All test-cases can be found in Appendix C. For 10 of those test cases the model needs the information that is passed in through RAG or needs to find a workaround, while the other 10 can be solved without any additional knowledge of the latest pandas versions. Therefore, we expect the baseline AI agent lacking RAG to score up to 50% accuracy, while agents using RAG and CoALA should achieve higher accuracy.

To enhance the evaluation of CoALA, we implemented 10% of the test cases with "Chain of Thought" necessary to come to the right result to assess its performance in scenarios where thought processes are crucial. Through forcing the LLM into listing a series of intermediate reasoning steps instead of jumping to the solution right away, we expect a better performance in tasks/prompts that require complex reasoning [12].

5 Experiments & Results

In this study, we conducted experiments to evaluate the performance of LLM-powered code generation AI agents. The evaluation process involves testing AI agents on a set of code generation tasks defined by test cases which can be found in Appendix C. Configurations for the experiments are systematically generated using a configuration grid, similarly to "Grid Search" for hyperparameter tuning, allowing for a comprehensive exploration of various options. The full list of configurations options can be found in Appendix A.

In our experiments, we employed three distinct prompting styles to assess the performance of the AI agent. The first approach utilized was Zero-Shot prompting, where the prompt was presented as a single paragraph without specific formatting, aiming to gauge the model's ability to generate responses based solely on general context. The second technique, Zero-Shot structured prompting, involved clearer prompts and a more organized format, breaking down the prompt into user questions, input data, and incorporating line breaks to enhance clarity and minimize the likelihood of the LLM misinterpreting the prompts. This method aimed to evaluate if a structured input could enhance the model's understanding and generation. Lastly, we explored One-Shot prompting, where a user question, input, and desired output were provided together, testing the model's capability to comprehend and generate responses based on a single set of instructions. The results from these varied prompting styles allowed us to analyze the model's adaptability and performance under different input conditions.

Table 1 presents the average performance metrics for the one-shot (1 input/output example) setting across 17 configurations. Among the three methods, RAG achieves the highest average accuracy at 0.6438, with a variance of 0.0028 and a standard deviation of 0.0526. CoALA follows closely with an accuracy of 0.6250, a variance of 0.0019, and a standard deviation of 0.0433. The No RAG scenario exhibits the lowest accuracy at 0.6000. These results suggest that, in the one-shot setting, RAG tends to outperform CoALA and the absence of RAG (No RAG).

Type	Accuracy	Variance	Standard Deviation
RAG	0.6438	0.0028	0.0526
CoALA	0.6250	0.0019	0.0433
No RAG	0.6000	-	-

Table 1: One-Shot (1 input/output example) average accuracy in descending order based on 17 configurations which can be found in Appendix A.

In the zero-shot structured setting, we observe a different trend (cf. Table 2). These findings suggest that, in the zero-shot structured scenario, the absence of RAG (No RAG) leads to better performance, while RAG and CoALA exhibit lower accuracies.

Type	Accuracy	Variance	Standard Deviation
No RAG	0.5500	-	-
RAG	0.5313	0.0037	0.0609
CoALA	0.5188	0.0043	0.0658

Table 2: Zero-Shot structured average accuracy in descending order based on 17 configurations which can be found in Appendix A.

In the zero-shot setting without a structured context (cf. Table 3), CoALA tends to perform better than RAG and No RAG and it comes with the smallest variance.

Type	Accuracy	Variance	Standard Deviation
CoALA	0.4625	0.0011	0.0331
RAG	0.4375	0.0036	0.0599
No RAG	0.4000	-	-

Table 3: Zero-Shot average accuracy in descending order based on 17 configurations which can be found in Appendix A.

In addition to assessing the average performance across various configurations, our study dived into a detailed analysis of the best-performing instances within each RAG type, including No RAG, RAG, RAG as tool, CoALA, and CoALA as tool. Moreover, our findings consistently revealed a notable trend: instances where RAG, RAG as tool, CoALA, or CoALA as tool were employed tended to outperform scenarios without retrieval augmentation (No RAG). This observed superiority in performance held true consistently across multiple test runs, underscoring the efficacy and reliability of RAG and CoALA in enhancing overall system performance.

The One-Shot prompting technique yielded diverse results, as indicated by Table 4. The configurations were assessed based on accuracy, total cost in dollars, and total time in seconds. When utilizing the RAG as a tool, the model demonstrated a commendable accuracy of 0.75,

Type	Accuracy	Total Cost in $	Total Time in sec
RAG as tool	0.75	2.36	304.2
CoALA as tool	0.70	2.49	256.93
RAG	0.70	23.11	722.2
CoALA	0.65	14.04	387.57
No RAG	0.60	14.34	802.25

Table 4: One-Shot prompting results sorted by accuracy for best AI agent configuration of each RAG type.

achieving a balance between precision and efficiency. Additionally, this came at a relatively low cost of $2.36 and a total processing time of 304.2s. CoALA as a tool exhibited a slightly lower accuracy of 0.70 but with comparable costs and a faster processing time of 256.93s. In contrast, using standard RAG resulted in a similar accuracy of 0.7 but with significantly increased cost ($23.11) and processing time (722.2s). The CoALA-only configuration showed an accuracy of 0.65, moderate cost ($14.04), and a processing time of 387.57s. Notably, when the RAG was excluded entirely, the model's accuracy dropped to 0.6, and the total cost and time escalated to $14.34 and 802.25s, respectively. This can be explained in part by looking at the conversations under the use of the ReAct prompting scheme, that exhibit how the model has difficulties coming to a correct result due to limited context.

The results from the Zero-Shot structured prompting approach are detailed in Table 5. Where enforcing the use of RAG and CoALA seems to have led to an information overflow during One-Shot prompting, it clearly has an edge over letting the agent decide himself in Zero-shot structured prompting as the accuracy improves from 0.55 to 0.6. However, despite the difference in accuracy, we observe a significant difference in cost.

Type	Accuracy	Total Cost in $	Total Time in sec
RAG	0.60	5.7	310.22
CoALA	0.60	6.13	352.81
CoALA as tool	0.55	1.82	251.36
RAG as tool	0.55	1.84	266.29
No RAG	0.55	8	537.52

Table 5: Zero-Shot structured prompting results sorted by accuracy for best AI agent configuration of each RAG type.

The Zero-Shot prompting approach (this time without a specially structured prompt) revealed distinct results across various configurations, as illustrated in the Table 6. Employing RAG in a Zero-Shot context yielded an accuracy of 0.55, although at a higher cost of $29.77 and an extended processing time of 558.97s. When CoALA was utilized as an external tool in a Zero-Shot format, the accuracy slightly decreased to 0.50, accompanied by a total cost of $5.14 and a processing time of 439.72s. Similarly, RAG used as a tool in a Zero-Shot scenario exhibited a comparable accuracy of 0.50, with a total cost of $5.88

and a processing time of 480.47s. The configuration using CoALA without external tool assistance achieved an accuracy of 0.50, incurring a higher cost of $12.9 and an increased processing time of 642.44s. Interestingly, excluding RAG entirely led to a reduction in accuracy to 0.40, with a lower total cost of $4.84 and a decreased processing time of 404.12s.

Type	Accuracy	Total Cost in $	Total Time in sec
RAG	0.55	29.77	558.97
CoALA as tool	0.50	5.14	439.72
RAG as tool	0.50	5.88	480.47
CoALA	0.50	12.9	642.44
No RAG	0.40	4.84	404.12

Table 6: Zero-Shot prompting results sorted by accuracy for best AI agent configuration of each RAG type.

6 Discussion

In our experimental comparison of One-Shot, Zero-Shot structured, and Zero-Shot prompting styles, we observed distinct patterns in their impact on accuracy. In the realm of One-Shot prompting, configurations employing RAG as a tool demonstrated the highest accuracy, reaching 0.75. This structured approach, where a user question, input, and desired output were provided together, yielded the most precise results. The intermediate accuracies of 0.70 were achieved by CoALA as a tool in One-Shot and RAG, showcasing a balanced performance. However, when RAG was excluded entirely, the accuracy dropped to 0.60, underscoring the importance of external information in certain configurations. We noticed that providing input/output examples made it easier for the LLM to understand what code it needed to generate to fully answer the user's question. With this method, the agent using RAG/-CoALA was able to improve by 0.20 compared to an agent without non-parametric knowledge.

In Zero-Shot structured prompting, we found that both RAG and CoALA in this structured format achieved the highest accuracy of 0.60. The clear organization of prompts into user questions and input data seemingly contributed to enhanced model understanding and better response generation. Configurations involving CoALA and RAG as tools in Zero-Shot structured, exhibited intermediate accuracies of 0.55. Interestingly, the absence of RAG in the Zero-Shot structured format only resulted in a 5% decrease in accuracy, underlining that the performance of an agent can be improved through prompt engineering without exhibiting external knowledge.

Finally, our exploration of standard Zero-Shot prompting revealed varied accuracies. RAG in the Zero-Shot format displayed the highest accuracy of 0.55, while configurations with CoALA as a tool, RAG as a tool, and CoALA without external tools achieved intermediate accuracies of 0.50. Notably, the configuration without RAG exhibited the lowest accuracy of 0.40, emphasizing the impact of external knowledge on enhancing the model's comprehension in certain scenarios. We noticed that RAG/CoALA can enhance the output quality by providing additional information to understand the user's query when prompts are formulated unclear.

When analyzing the total time and cost of various configurations, a clear pattern emerges. As one might expect, using RAG/CoALA as tool makes a significant difference in terms of cost, as our GPT-4 based agent seems to not employ tools frivolously[4]. It's important to note that the time and cost metrics can be affected by different factors. External elements, such as internet connectivity, can have a significant impact on processing time and result in varying outcomes. Additionally, the response length and server load of the LLM can cause fluctuations in the responding time. Furthermore, complications can arise when the LLM generates unparsable code, which requires the model to fix the code through an iterative process. This process can either result in successfully fixing the code or may require the maximum iterations of the AI agent loop, which can significantly extend the processing time and consume a considerable number of tokens. Therefore, it's crucial to consider external variables and potential inefficiencies when evaluating the total time and cost across different prompting configurations.

7 Conclusion

Our research on code generation using LLMs with RAG has provided valuable insights into how AI agents can improve their understanding and generation of code. Beyond that, it shows how the structure of the request can have a significant impact on the performance. In that, we conducted experiments using One-Shot, Zero-Shot structured, and Zero-Shot prompting and observed significant improvements when structuring the prompt especially clearly. In summary, the One-Shot and Zero-Shot structured prompting styles generally outperformed standard Zero-Shot prompting, with structured and explicit instructions contributing to improved accuracy.

We recommend that developers approach AI agents incrementally, starting with standard usage before opting for RAG/CoALA because, as prompt engineering can already lead to major improvements in output quality with manageable overhead. It's essential to assess whether these approaches are sufficient for the specific use case before resorting to RAG/CoALA, as the latter comes with a significant increase in cost, even if improved results can be achieved. This stepwise approach ensures a balance between performance enhancement and cost-effectiveness, aligning the use of advanced frameworks with the specific requirements of the task at hand. At present, we consider these capabilities to be more of a starting point for development.

In order to improve the evaluation process, further research should look beyond analyzing code syntax. A more comprehensive approach would involve parsing the code and executing it as part of the observation. This way, the LLM can observe the code's output and gain a deeper

[4] We did not encourage or discourage the agent to use his tools through the system prompt.

understanding of whether it is functioning as intended. To ensure compatibility across different coding topics, it would make sense to execute the code as a standalone executable. This will involve testing through stdin and stdout, which will abstract the underlying programming language and make the project more widely applicable.

A RAG Configurations

In our configuration setup, we explored various options and utilized all available configuration parameters. For the distance metric, we opted for the euclidean distance and the max inner product. Regarding the number of search results, we experimented with offering both 1 and 3 results per user query, providing flexibility in the quantity of retrieved documents. To ensure efficient handling of longer documents, we segmented texts into chunks of 512 characters. Throughout our experiments, we set the similarity score threshold to 0.0 and used a weighted average of text chunks.

RAG Type	Description
No RAG	The AI agent lacks RAG capabilities.
RAG	Provides additional information from the pandas documentation.
RAG as tool	The AI agent decides whether to use RAG.
CoALA	Provides additional information from the pandas documentation and question & answer pairs.
CoALA as tool	The AI agent decides whether to use CoALA.

Table 7: RAG options.

Distance Metric	Formula
Euclidean Distance	$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
Max Inner Product	$\max_{\|u\|=1,\|v\|=1}(u \cdot v)$
Cosine Similarity	$\frac{u \cdot v}{\|u\| \cdot \|v\|}$

Table 8: Supported distance metrics.

Setting	Description
Number of Search Results	Number of documents to return.
Similarity Search Score Threshold	Similarity score for a document to be included in the search results.
Text Chunk Size	Divides the input text into chunks of the specified size.
Use Weighted Average	Text chunks are embedded separately (2 chunks -> 2 vectors), or combined using averaging (2 chunks -> 1 vector).

Table 9: Other RAG settings.

References

[1] E. Melz, *Enhancing llm intelligence with arm-rag: Auxiliary rationale memory for retrieval augmented generation* (2023)

[2] Z. Liu, X. Yu, L. Zhang, Z. Wu, C. Cao, H. Dai, L. Zhao, W. Liu, D. Shen, Q. Li et al., *Deid-gpt: Zero-shot medical text de-identification by gpt-4* (2023)

[3] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.t. Yih, T. Rocktäschel et al., Advances in Neural Information Processing Systems **33**, 9459 (2020)

[4] S. Liu, Y. Chen, X. Xie, J. Siow, Y. Liu, arXiv preprint arXiv:2006.05405 (2020)

[5] Z. Jiang, F.F. Xu, L. Gao, Z. Sun, Q. Liu, J. Dwivedi-Yu, Y. Yang, J. Callan, G. Neubig, arXiv preprint arXiv:2305.06983 (2023)

[6] O. Khattab, C. Potts, M. Zaharia, *Baleen: Robust multi-hop reasoning at scale via condensed retrieval* (2021)

[7] J. Kim, S. Choi, R.K. Amplayo, S.w. Hwang, *Retrieval-augmented controllable review generation*, in *Proceedings of the 28th International Conference on Computational Linguistics* (2020), pp. 2284–2295

[8] M. Glass, G. Rossiello, M.F.M. Chowdhury, A. Gliozzo, arXiv preprint arXiv:2108.13934 (2021)

[9] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, Y. Cao, *React: Synergizing reasoning and acting in language models* (2023), 2210.03629

[10] J. Johnson, M. Douze, H. Jégou, IEEE Transactions on Big Data **7**, 535 (2019)

[11] T.R. Sumers, S. Yao, K. Narasimhan, T.L. Griffiths, *Cognitive architectures for language agents* (2023), 2309.02427, https://arxiv.org/abs/2309.02427

[12] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E.H. Chi, Q. Le, D. Zhou, CoRR **abs/2201.11903** (2022), 2201.11903

B Contributions

Jan Henrik Bertrand: Implemented AI agent, RAG, CoALA, and tools for the agent. Worked on section 3, 5 of this paper.

Andre Anan Gilbert: Implemented LLM client, RAG, and automated evaluation. Worked on section 3, 4, 5, 6, and 7 of this paper.

Felix Noll: Crawled & stripped pandas documentation and developed test cases. Worked on the Appendix of this paper.

Marc Grün: Implemented testing framework for code evaluation and demo notebook and embedded pandas documentation. Worked on section 1, 2 and 4 of this paper. Created presentation.

C Test Cases

Examples of all prompting styles are shown below.

Zero-shot unstructured:

PROMPT

```
How can I convert this one-hot encoded dataframe::df = pd.DataFrame({"col1_a":[1, 0, 1],"col1_b":
[0, 1, 0], "col2_a": [0, 1, 0], "col2_b": [1, 0, 0],"col2_c": [0, 0, 1]}) into a categorical dataframe?
```

INPUT

```
data: data = pd.DataFrame({"col1_a": [1, 0, 1], "col1_b": [0, 1, 0],"col2_a": [0, 1, 0], "col2_b":
[1, 0, 0], "col2_c": [0, 0, 1]}
```

EXPECTED OUTPUT

```
import pandas as pd
    def respone_function(data):
    result = pd.from_dummies(data, sep="*")
    return result
```

PROMPT

```
This is my Dataframe:({'Name': ['Alice', 'Bob', 'Aritra'], 'Age': [25, 30, 35], 'Location':
['Seattle', 'New York', 'Kona']},index=([10, 20, 30])) Please take this dataframe as your argument
and display the dataframe while making sure to change the index to 100, 200 and 300.
```

INPUT

```
data = pd.DataFrame({'Name': ['Alice', 'Bob', 'Aritra'], 'Age': [25, 30, 35],
'Location': ['Seattle', 'New York', 'Kona']},index=([10, 20, 30]))
```

EXPECTED OUTPUT

```
import pandas as pd
def respone_function(data):
    data.index = [100, 200, 300]
    return data
```

PROMPT

```
({'animal': ['alligator', 'bee', 'falcon', 'lion','monkey', 'parrot', 'shark', 'whale', 'zebra']})
This is my dataframe, your argument. Please display all but the last 3 rows of the dataframe.
```

INPUT

```
data = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion','monkey',
'parrot', 'shark', 'whale', 'zebra']})
```

EXPECTED OUTPUT

```
import pandas as pd
def respone_function(data):
    result = data.iloc[:-3, :]
    return result
```

PROMPT

ts = pd.Timestamp('2017-01-01 09:10:11') This is your argument. Please add 2 Months to that timestamp.

INPUT

```
data = pd.Timestamp('2017-01-01 09:10:11')
```

EXPECTED OUTPUT

```
import pandas as pd
def respone_function(data):
    result = data + pd.DateOffset(months=2)
    return result
```

PROMPT

ser = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']). Please calculate the expending sum of that series (which is your argument). Make sure to display each row.

INPUT

```
data = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
```

EXPECTED OUTPUT

```
import pandas as pd
def respone_function(data):
    result = data.expanding().sum()
    return result
```

PROMPT

How can I use a function and the 'data_2' DataFrame to calculate the grouped product based on the 'a' column and display the resulting DataFrame? data_1 = [[1, 8, 2], [1, 2, 5], [2, 5, 8], [2, 6, 9]]data_2 = pd.DataFrame(data_1, columns=["a", "b", "c"],index=["tiger", "leopard", "cheetah", "lion"])

INPUT

```
data_1 = [[1, 8, 2], [1, 2, 5], [2, 5, 8], [2, 6, 9]]
data_2 = pd.DataFrame(data_1, columns=["a", "b", "c"] ,
          index=["tiger", "leopard", "cheetah", "lion"])
```

EXPECTED OUTPUT

```
import pandas as pd
def correct_function(data_2):
    result = data_2.groupby('a').prod()
    return result
```

PROMPT

a = pd.Series([1, 1, 1, None], index=['a', 'b', 'c', 'd']) b = pd.Series([1, None, 1, None], index=['a','b', 'd', 'e']) Please takea and b as your arguments and divide a by b. Please also use the fill value 0.

INPUT

```
data_1 = pd.Series([1, 1, 1, None], index=['a', 'b', 'c', 'd'])
data_2 = pd.Series([1, None, 1, None], index=['a', 'b', 'd', 'e'])
```

EXPECTED OUTPUT

```
import pandas as pd
import numpy as np
def correct_function(*args):
    data_1, data_2 = args[1:] # to avoid declaring numpy from the import above
                    as an argument
    result = data_1.div(data_2, fill_value=0)
    return result
```

PROMPT

data = {('level_1', 'c', 'a'): [3, 7, 11],('level_1', 'd', 'b'): [4, 8, 12]('level_2', 'e', 'a'): [5, 9, None],('level_2', 'f', 'b'): [6, 10, None],} Please drop column a and make sure to take data as your argument.

INPUT

```
data = pd.DataFrame({('level_1', 'c', 'a'): [3, 7, 11],('level_1', 'd', 'b'):
[4, 8, 12],('level_2', 'e', 'a'): [5, 9, None],
('level_2', 'f', 'b'): [6, 10, None],})
```

EXPECTED OUTPUT

```
import pandas as pd
def correct_function(data):
    result = data.droplevel(2, axis=1)
    return result
```

PROMPT

Please take following Series, which serves as your agrument, and order it ascending while making sure NAN values are at the beginning s = pd.Series([None, 1, 3, 10, 5, None])

INPUT

```
data = pd.Series([None, 1, 3, 10, 5, None])
```

EXPECTED OUTPUT

```
import pandas as pd
def correct_function(*args):
    data = pd.Series(args[1:]) # to avoid declaring numpy from the import above
           as an argument
    result = data.sort_values(na_position='first')
    return result
```

PROMPT

```
data1 = {'Name': ['Alice', 'Bob', 'Charlie'],'Age': [25, 30, 22],'City': ['New York', 'San Francisco',
'Los Angeles']} data2= {'Name': ['Alice', 'John', 'Charlie'],'Age': [25, 31, 22],'City': ['New York',
'San Francisco', 'Los Angeles']}Please calculate the average age of the people who appear in both
dataframes.Make sure to take data1 and data2 as your arguments.
```

INPUT

```
data_1 = {'Name': ['Alice', 'Bob', 'Charlie'],'Age': [25, 30, 22],'City':
          ['New York', 'San Francisco', 'Los Angeles']}
data_2 = {'Name': ['Alice', 'John', 'Charlie'],
         'Age': [25, 31, 22],'City': ['New York', 'San Francisco', 'Los Angeles']}
```

EXPECTED OUTPUT

```
import pandas as pd
def respone_function(data_1, data_2):
    df_1 = pd.DataFrame(data_1)
    df_2 = pd.DataFrame(data_2)
    merged_df = pd.merge(df_1, df_2, on='Name')
    result = merged_df['Age_x'].mean()
    return result
```

PROMPT

```
data = { 'Timestamp': [ '2023-01-01 12:01:00', '2023-01-01 12:10:00', '2023-01-01 12:25:00',
'2023-01-01 13:05:00', '2023-01-01 13:25:00', '2023-01-01 14:00:00', '2023-01-02 08:30:00',
'2023-01-02 09:00:00', '2023-01-02 09:35:00' ],'User': [1, 1, 1, 2, 2, 2, 3, 3, 3], 'Page': ['Home',
'Product', 'Checkout', 'Home', 'Product', 'Home', 'Home','Product','Checkout'] } Using the pandas
DataFrame, wich is your argument,implement the following operation:Create a new column called
'Session_ID' that labels each row with a uniquesession identifier.Define a session as a series of
consecutive interactions by the same userwith no gap greatern than 30 minutes between interactions.
Ensure that eachsession has a unique identifier. Make sure to give me the full code.
```

INPUT

```
data = pd.DataFrame({'Timestamp': ['2023-01-01 12:01:00', '2023-01-01 12:10:00',
'2023-01-01 12:25:00', '2023-01-01 13:05:00','2023-01-01 13:25:00',
'2023-01-01 14:00:00',
'2023-01-02 08:30:00', '2023-01-02 09:00:00','2023-01-02 09:35:00'],
'User': [1, 1, 1, 2, 2, 2, 3, 3, 3],
'Page': ['Home', 'Product', 'Checkout', 'Home',
'Product', 'Home', 'Home', 'Product', 'Checkout']})
```

EXPECTED OUTPUT

```
import pandas as pd
def respone_function(data):
    data['Timestamp'] = pd.to_datetime(data['Timestamp'])
    data = data.sort_values(by=['User', 'Timestamp'])
    data['TimeDiff'] = data.groupby('User')['Timestamp'].diff()
    data['Session_ID'] = (data['TimeDiff'] > pd.Timedelta(minutes=30)).cumsum()
    data = data.drop('TimeDiff', axis=1)
    return data
```

PROMPT

Please return the rolling rank(3) of this Series [1, 4, 2, 3, 5, 3]. Make sure to take this Series as your argument as well as using the pandas lib.

INPUT

```
data = pd.Series([1, 4, 2, 3, 5, 3])
```

EXPECTED OUTPUT

```
import pandas as pd
def respone_function(data):
    result = data.rolling(3).rank()
    return result
```

PROMPT

Please create a dictionary using the following Dataframe. This dataframe isyour argument. Make sure to order it tight. pd.DataFrame([[1, 3], [2, 4]],index=pd.MultiIndex.from_tuples([("a", "b"), ("a", "c")], names=["n1", "n2"]),columns=pd.MultiIndex.from_tuples([("x", 1), ("y", 2)], names=["z1", "z2"]),)

INPUT

```
data = pd.DataFrame.from_records([[1, 3], [2, 4]],
index=pd.MultiIndex.from_tuples([("a", "b"), ("a", "c")],names=["n1", "n2"]),
columns=pd.MultiIndex.from_tuples([("x", 1), ("y", 2)], names=["z1", "z2"]),)
```

EXPECTED OUTPUT

```
import pandas as pd\ndef respone_function(data):
    result = data.to_dict(orient='tight')
    return result
```

PROMPT

Please take following dataframe (your argument) as your Input data = pd.DataFrame(["g", "g0"], ["g", "g1"], ["g", "g2"], ["g", "g3"],["h", "h0"], ["h", "h1"]], columns=["A", "B"]). This is the desired Output:{'A': ['g', 'g', 'g', 'h'],'B': ['g0', 'g1', 'g2', 'h0']}. Please write some code to go from Input to Output.

INPUT

```
"data = pd.DataFrame([["g", "g0"], ["g", "g1"], ["g", "g2"], ["g", "g3"],
["h", "h0"],["h", "h1"]],columns=["A", "B"])
```

EXPECTED OUTPUT

```
import pandas as pd
    def correct_function(data):
    result = data.groupby("A").head(-1)
    return resultt
```

PROMPT

Please remove the following suffix _str from following Series(which is your argument)=(["foo_str", "_strhead" , "text_str_text" ,"bar_str", "no_suffix"])

INPUT

```
data = pd.Series(["foo_str","_strhead" , "text_str_text" , "bar_str", "no_suffix"])
```

EXPECTED OUTPUT

```
import pandas as pd
def correct_function(data):
    result = data.str.removesuffix("_str")
    return result
```

PROMPT

I have 2 Dataframes which are you arguments. The first one: pd.DataFrame({'key': ['K0', 'K1', 'K1', 'K3', 'K0', 'K1'], 'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']}) And the second one: pd.DataFrame({'key': ['K0', 'K1', 'K2'], 'B': ['B0', 'B1', 'B2']})How do I join the second one on the first one using the key and making sure it is a m:1 relation?

INPUT

```
data_1 = pd.DataFrame({'key': ['K0', 'K1', 'K1', 'K3', 'K0', 'K1'],
         'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
data_2 = pd.DataFrame({'key': ['K0', 'K1', 'K2'], 'B': ['B0', 'B1', 'B2']})
```

EXPECTED OUTPUT

```
import pandas as pd
def respone_function(data_1, data_2):
    result = data_1.join(data_2.set_index('key'), on='key', validate='m:1')
    return result
```

PROMPT

This is your Index:pd.MultiIndex.from_tuples([('bird', 'falcon'),('bird', 'parrot'),('mammal', 'lion'), ('mammal', 'monkey')],names=['class', 'name']) These are your columns:pd.MultiIndex.from_tuples([('speed', 'max'),('species', 'type')]) And this is your input:pd.DataFrame([(389.0, 'fly'),(24.0, 'fly'), (80.5, 'run'),(None, 'jump')],index=index,columns=columns).Index, Columns and Input are your arguments. Please create a dataframe and rename the index to classes and names

INPUT

```
data_1 = pd.MultiIndex.from_tuples([('bird', 'falcon'),('bird', 'parrot'),
        ('mammal', 'lion'),
        ('mammal', 'monkey')],names=['class', 'name'])
data_2 = pd.MultiIndex.from_tuples([('speed', 'max'),('species', 'type')])
data_3 = pd.DataFrame([(389.0, 'fly'),(24.0, 'fly'),(80.5, 'run'),(None,
         'jump')],index=data_1,columns=data_2)
```

EXPECTED OUTPUT

```
import pandas as pd
def correct_function(*args):
    data_1, data_2, data_3 = args[1:] # to avoid declaring numpy from the
                             import above as an argument
    result = data_3.reset_index(names=['classes', 'names'])
    return result
```

PROMPT

Please take this as your Input: pd.Series(['quetzal', 'quetzal', 'elk'], name='animal')?Please also take the Series as your argument. Write some Code to get the Input to this Output:data = ['quetzal', 'quetzal', 'elk'] series = pd.Series(data, name='animal').

INPUT

```
data = pd.Series(['quetzal', 'quetzal', 'elk'], name='animal')
```

EXPECTED OUTPUT

```
import pandas as pd
def respone_function(data):
    result = data.value_counts()
    return data
```

PROMPT

Please take these consecutive values as your Input as well as your argument: pd.Index([10, 20, 30, 40, 50]). Please write some code to transform this Input into the Output i will provide. Index([nan, 10.0, 10.0, 10.0, 10.0], dtype='float64')

INPUT

```
data = pd.Index([10, 20, 30, 40, 50])
```

EXPECTED OUTPUT

```
import pandas as pd
def respone_function(data):
    sum = data.diff()
    return sum
```

PROMPT

df = pd.DataFrame({"a": [1, 1, 2, 1], "b": [None, 2.0, 3.0, 4.0]},dtype="Int64")This is my Dataframe which is also your argument. Please convert the Int64 to Int64[pyarrow] and use df.sum() at the end.

INPUT

```
data = pd.DataFrame({"a": [1, 1, 2, 1], "b": [None, 2.0, 3.0, 4.0]},
dtype="Int64")
```

EXPECTED OUTPUT

```
import pandas as pd
import pyarrow as pa
def respone_function(data):
    data = data.astype("int64[pyarrow]")
    data.sum()
    return data
```

Zero-shot structured:

PROMPT

```
I have a one-hot encoded DataFrame with '_' as the separator.
How can I revert this one-hot encoded DataFrame back into a categorical DataFrame using pandas?

The following DataFrame will be the only function argument:
df = pd.DataFrame({
    'col1_a': [1, 0, 1],
    'col1_b': [0, 1, 0],
    'col2_a': [0, 1, 0],
    'col2_b': [1, 0, 0],
    'col2_c': [0, 0, 1],
})
```

INPUT

```
data = pd.DataFrame({"a": [1, 1, 2, 1], "b": [None, 2.0, 3.0, 4.0]},
dtype="Int64")
```

EXPECTED OUTPUT

```
import pandas as pd
import pyarrow as pa
def respone_function(data):
    data = data.astype("int64[pyarrow]")
    data.sum()
    return data
```

One-shot:

PROMPT

```
I have a one-hot encoded DataFrame with '_' as the separator.
How can I revert this one-hot encoded DataFrame back into a categorical DataFrame using pandas?

The following DataFrame will be the only function argument:
df = pd.DataFrame({
        'col1_a': [1, 0, 1],
        'col1_b': [0, 1, 0],
        'col2_a': [0, 1, 0],
        'col2_b': [1, 0, 0],
        'col2_c': [0, 0, 1],
    })

    Desired Output:
        col1 col2
```

```
    0    a    b
    1    b    a
    2    a    c
```

INPUT

```
data = pd.DataFrame({"a": [1, 1, 2, 1], "b": [None, 2.0, 3.0, 4.0]},
dtype="Int64")
```

EXPECTED OUTPUT

```
import pandas as pd
import pyarrow as pa
def respone_function(data):
    data = data.astype("int64[pyarrow]")
    data.sum()
    return data
```