Past as a Guide: Leveraging Retrospective Learning for Python Code Completion

Seunggyoon Shin[1,2], Seunggyu Chang[2], Sungjoon Choi[1]
[1] Korea University
[2] NAVER Cloud

Abstract

This work presents Past as a Guide (PaG), a simple approach for Large Language Models (LLMs) to improve the coding capabilities by integrating the past history with interactive and iterative code refinements. To be specific, inspired by human cognitive processes, the proposed method enables LLMs to utilize previous programming and debugging experiences to enhance the Python code completion tasks. The framework facilitates LLMs to iteratively refine the Python code based on previous execution and debugging results and optimize learning and reasoning capabilities. The proposed methodology achieved a 92% pass@1 on HumanEval, demonstrating the potential to advance the field by leveraging retrospection from past experiences and interactive and iterative refinement processes without external correctness indicators.

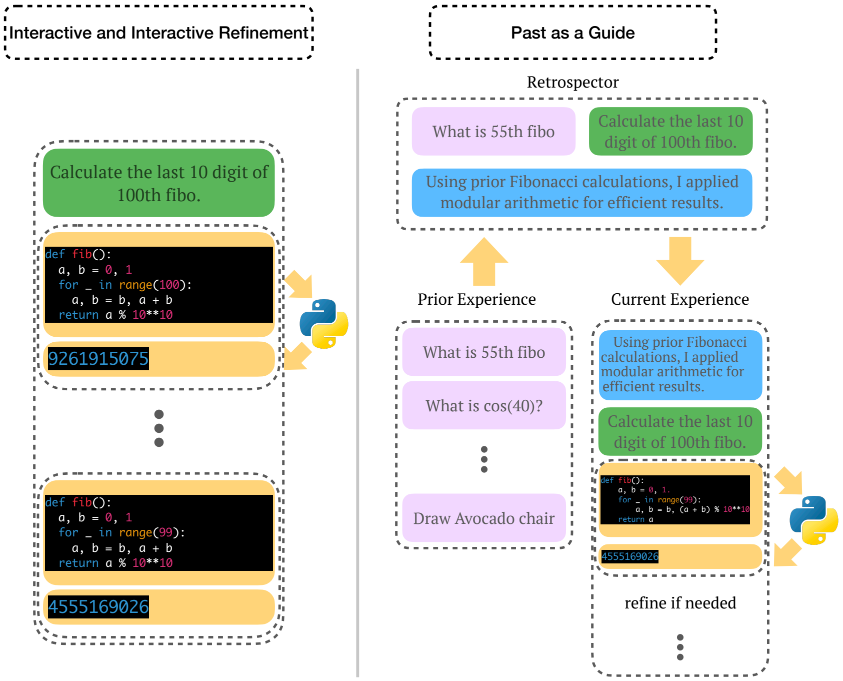
1 Introduction

Large Language Models (LLMs) play a pivotal role in advancing artificial intelligence research, exhibiting versatility across various applications including coding interfaces and search mechanisms. This versatility is emphasized by seminal works such as ReAct [21], SayCan [1], Toolformer [16], among others. Concurrently, leveraging code is becoming a substantial medium for improving task performance, as illustrated by PAL [7] and Program of Thoughts [5]. Our study, "Past as a Guide(PaG)," aligns with these advancements, aiming to optimize LLM performance through the integration of interactive and iterative code refinement from past retrospective guidance.

This mirrors a fundamental trait of human cognition: the intrinsic ability to harness past experiences. By adeptly employing short-term and long-term memory encoding and retrieval, humans address complex reasoning tasks, especially in areas like programming. Consider, for instance, the task of finding the last 100 digits of the n-th Fibonacci number. Often, when confronted with such challenges, humans draw from their reservoir of past experiences. They recall similar tasks, such as computing the n-th Fibonacci number, and intuitively gauge the similarities and differences between the old and new challenges. This process demonstrates a synergy: humans use long-term memory retrieval to construct retrospections from prior experiences and concurrently engage in short-term iterative refinements.

We conduct empirical evaluations to assess the performance of LLMs in harnessing retrospections, leading to a significant achievement of 92% pass@1 at HumanEval. Instead of relying on direct external feedback on correctness, we utilize the MBPP dataset [2] to construct a long-term memory for the LLMs. When faced with a problem from HumanEval [4], the model retrieves and leverages similar problems from its MBPP experiences, effectively boosting its performance. This method, combining interactive and iterative code refinements informed by past experiences, signifies a pivotal advancement in LLM capabilities, indicating a promising trajectory for their future evolution.

Submitted to the Workshop on Instruction Tuning and Instruction Following at NeurIPS 2023. Do not distribute.


Interactive and Interactive Refinement
Past as a Guide
Retrospector
What is 55th fibo
Calculate the last 10 digit of 100th fibo.
Using prior Fibonacci calculations, I applied modular arithmetic for efficient results.
Prior Experience
Current Experience
What is 55th fibo
What is cos(40)?
Draw Avocado chair
Using prior Fibonacci calculations, I applied modular arithmetic for efficient results.
Calculate the last 10 digit of 100th fibo.
refine if needed
Calculate the last 10 digit of 100th fibo.

```
def fib():
    a, b = 0, 1
    for _ in range(100):
        a, b = b, a + b
    return a % 10**10
```

```
9261915075
```

```
def fib():
    a, b = 0, 1
    for _ in range(99):
        a, b = b, a + b
    return a % 10**10
```

```
4555169026
```

```
def fib():
    a, b = 0, 1.
    for _ in range(99):
        a, b = b, (a + b) % 10**10
    return a
```

```
4555169026
```

Figure 1: Left: **Interactive and Iterative Code Refinement** showcased through a process of repeated corrections. Right: Retrieval of the most similar question, aligned with the **Current instruction**, sourced from the Past Experience. By merging these insights, we form **retrospections**. The subsequent **code generation** follows an interactive and iterative Code refinement process, emphasizing code execution and debugging.

2 Related work

2.1 Code Generation LLMs

There has been a surge of interest in generating code using LLMs. Notable advancements in this domain include the methodologies proposed by Roziere et al. [15], Li et al. [11], Chen et al. [4], and another separate work by Li [11]. These research pieces underscore the capability of LLMs to craft precise and functional code structures. Additionally, innovative techniques have emerged that harness models like ChatGPT or those that deploy self-instruction paradigms [20]. Such models are first primed to generate paired datasets of instructions and code, which are subsequently used to fine-tune the LLMs [13, 9]. In a similar vein, Zhou et al. [23] leverage external knowledge through "DocPrompting" by integrating code documentation for code generation. While this approach underscores the value of external references, our approach uniquely focuses on leveraging the past experiences of LLMs to improve problem-solving in programming tasks.

2.2 Code Execution LLMs

An alternative but equally compelling line of work exploits code not merely as an output but as a vital operative tool for LLMs. For instance, the PAL [7] utilizes code execution results embedded within prompts to boost the model's proficiency in mathematical and numerical benchmarks. Moreover, the Binder approach [6] demonstrates that LLMs, when equipped with SQL or Python execution capabilities, manifest a superior comprehension of tabular datasets. Exploring the intersection of robotics and LLMs, the Code as Policies [12] postulates that LLMs can produce code, serving as operational policies, to govern robotic actions fulfilling specific instructions. Furthermore, ViperGPT [18], an innovative paradigm that synergizes vision foundation models with Python-executable code generated by GPT to address vision-oriented tasks.

2.3 Self Improvement

Various studies have highlighted innovative approaches to enhance the programming performance of Language Learning Models (LLMs). For instance, Haluptzok et al. [8] demonstrated that crafting programming puzzles and validating solutions collaboratively can produce verified synthetic data. When LLMs are fine-tuned with this data, there's a marked improvement in their programming capabilities. On a different tangent, "Reflexion," as developed by Shinn et al. [17], grants agents dynamic memory and the ability for self-reflection to improve reasoning abilities. Although Reflexion employs a form of external memory, its application is constrained to scenarios involving identical instruction sets. This methodology, while effective for repeated tasks, may not generalize well to varied problems that share conceptual similarities but differ in specifics, as is common in real-world applications. Consequently, in our comparative analysis, Reflexion is not categorized as using external memory in a broad sense, which we reserve for systems that leverage past insights dynamically across distinct but conceptually related problems. In the realm of mathematics, Zhou et al. [22] emphasize that prompts focused on verification, combined with voting techniques, can notably enhance GPT-4's performance in math-related benchmarks. Another intriguing approach is presented by Madaan et al. [14], wherein the model iteratively refines its outputs leveraging self-generated feedback. "Voyager" by Wang et al. [19] conceptualizes code as a skill. It recalls and incorporates skills acquired from prior tasks, thus sharpening its proficiency in abstractions. In contrast to the above methodologies that often rely on external rewards or feedback, our proposed PaG utilizes generated codes to rectify errors discerned from prior retrospections. Notably, our method constructs these retrospections solely based on its own execution results, without the need for any external input or feedback.

Table 1: Comparison of LLM Enhancement Methodologies. *As discussed in [10], we do not classify receiving right or wrong feedback from an oracle evaluator as oracle feedback; instead, we utilize tools to obtain intrinsic feedback.

	Iterative Self Correction	**Dynamic External Memory Use***	**No Oracle Feedback†**
Voyager	X	✓	✓
Self-refine [14]	✓	X	✓
Reflexion [17]	✓	X*	X
PaG (Ours)	✓	✓	✓

*Dynamic External Memory Use is considered here as the ability to apply learned insights across varied but conceptually related problems. Reflexion's memory use is limited to identical instructions. †In the context of oracle feedback, we refer to the use of explicit right/wrong evaluation from an oracle. Our approach utilizes intrinsic feedback mechanisms as delineated in [10].

3 Method

3.1 Interactive and Iterative Code Refinements

Overview: Consider an instruction I presented in natural language. To achieve the desired outcome based on this instruction, we utilize a policy, denoted as π. This policy, in the context of our work, is embodied by a Large Language Model (LLM) denoted as `LLM`. The policy π produces a verbal action A which, in essence, is textual content. When the content of A contains a code segment, it can be systematically parsed and executed. The outcome of this execution is subsequently appended to A. Engaging in this process iteratively allows for the comprehensive completion of the task at hand. To explicate further, the action A is comprised of a set, such that:

$$A = \{(\texttt{text}, \texttt{code}, \texttt{execution\_result})\} \times \text{\# of trials}$$

Here, each element of the set A represents a trial, constituting the text, the parsed code, and the result of executing that code, whose collective enumeration accomplishes the instructed task.

Implementations: For our experiment, we employ the GPT4 API. It has been observed that several popular LLM APIs, such as ChatGPT and GPT4, exhibit constraints when tasked with code generation. Primarily, many extant LLM models are trained with safety mechanisms to prevent potential harm, as discussed by Bai et al. (2022) [3]. Consequently, when requests are made for operations like fetching data from 'yfinance' or image manipulation tasks, these models might assert their inability to execute them. Contrarily, they are indeed capable of producing the correct code snippets necessary for task completion. To circumvent this limitation, we delineated the model's role and its execution capabilities within the `SYSTEM_PROMPT` section of ChatGPT. A comprehensive version of this prompt can be found in Appendix A. Then, We utilized Python's regex functionality to extract and filter out code blocks. If a code segment is detected, it is executed using a Jupyter Notebook client. Subsequently, the resultant output from the code segment is assimilated into the GPT4 trajectories.

3.2 Exploring stage

Memory Curation: In the exploration stage, each run of an `LLM` produces a tuple, denoted by t, and is mathematically represented as:

$$t = \langle \texttt{USER\_INSTRUCTION}, \texttt{ASSISTANT\_ANSWER} \rangle$$

This tuple, t, is then appended to a memory store, symbolized as $\mathcal{M}$, in accordance with the following relation:

$$\mathcal{M} \leftarrow \mathcal{M} \cup \{t\}$$

Here, $\mathcal{M}$ acts as a repository embodying the aggregate experiences or memories of the `LLM`. Each element within $\mathcal{M}$ is a unique tuple comprising a user instruction and the corresponding assistant response. The incorporation of new tuples into $\mathcal{M}$ signifies the `LLM`'s process of updating its knowledge base, learning, and adapting through each run. For the purpose of this paper, we leverage the initial subset of MBPP [2], consisting of the first 470 examples, to formulate the $\mathcal{M}$

3.3 Past as a Guide

PaG concentrates on utilizing experienced memories $\mathcal{M}$ to proficiently formulate an action, A, capable of fulfilling a provided instruction, I. This method unfolds systematically through several interconnected steps.

Initially, an instruction, I, instigates a search within our structured memory store $\mathcal{M}$. To facilitate this, we calculate embeddings for I and each instruction in $\mathcal{M}$ using OpenAI's `text-embedding-ada-002` model and employ cosine similarity as the metric to quantify resemblance between the instruction embeddings, thereby extracting the most analogous instruction from $\mathcal{M}$.

$$I_{\text{similar}} = \underset{I' \in \mathcal{M}}{\operatorname{argmax}} \ \text{Similarity}(I, I') \tag{1}$$

When a similar instruction is identified, we use its corresponding action trajectories, A, together with the original instruction, I, to query the LLM. This query is designed to derive retrospections, denoted as R. It emphasizes the extraction and application of knowledge from past experiences to the present context.

$$R = \texttt{LLM}\,(\text{“How can prior experience "}A_{\text{similar}}\text{" be applied to solve "}I\text{"?”}) \tag{2}$$

After the acquisition of retrospections, R, they are seamlessly integrated as a prefix to the initial instruction, I, enriching the instruction base to generate the refined action, A, targeting the accomplishment of the outlined task in the initial instruction.

$$A = \texttt{LLM}(R + I) \tag{3}$$

PaG utilizes knowledge from previous interactions stored in $\mathcal{M}$. This knowledge helps refine the model's responses to new instructions, placing retrospection of past experiences in the context of new situations. As a result, the model becomes more effective in completing tasks.

For a more detailed, practical illustration of our method refer to Appendix B.

4 Code Completion Experiments

Task: The task is derived from HumanEval [4], where the objective is to complete a provided Python code skeleton. This task presented a natural language description along with an incomplete Python code snippet. The completed code is then verified for correctness by executing the associated test code, which either yields a pass or fail outcome based on the code's accuracy and efficacy. The primary goal is to successfully fulfill the requirements delineated in the natural language description by augmenting the Python skeleton to create a functional and correct code piece.

Results: In our experiments, we assess the effectiveness of our proposed methods against baseline models. The following table 2. 2 summarizes the performances of the different approaches:

Method	% Pass@1
GPT-4	67.00
GPT-4 + Interactive and Iterative Code Refinement (max 6 tries)	84.15
GPT-4 + Interactive and Iterative Code Refinement (max 12 tries)	90.85
Reflexion [17]	91.00
GPT-4 + Interactive and Iterative Code Refinement (max 12 tries) + PaG	**92.07**

Table 2: Performance comparison of different methods, with our method highlighted.

PaG achieved state-of-the-art performance, a commendable feat realized without relying on external correctness feedback from the test code. Our approach uniquely combines retrospective guidance from past experiences with an interactive and iterative code refinements loop driven by code execution. This emphasizes the dual strengths of our method: harnessing past experiences for informed guidance and using interactive refinements for precision. Both factors collectively enhance the model's performance, showcasing the potential of integrating experienced knowledge with dynamic refinements without the need for external correctness indicators.

The actions generated by the Large Language Model, denoted as $\mathcal{A}$, include numerous code blocks, each undergoing multiple trials. Every block within $\mathcal{A}$ is subjected to a comprehensive examination against corresponding test cases. However, due to the possibility of discrepancies arising during intermediate executions of the solution function, an erroneous 'fail' flag may be prompted even if the final solution is accurate. To counteract this, we employ a refinement procedure, using regular expressions to meticulously filter and eliminate all print and assert statements within the generated code blocks. Any cases initially identified as failures undergo a re-assessment phase, wherein the accurate code block is extracted from the ensemble within $\mathcal{A}$ and subjected to a secondary evaluation.

5 Conclusion and Limitation

This work has developed a methodology that, akin to human learning, utilizes prior experiences and retrospection to enhance performance from the initiation phase through the reasoning stages, employing a multi-attempt code interpreter. Our proposed method, PaG, integrates past experiences with current instructions to craft retrospections. Further, it employs interactive and iterative code refinements through execution. By drawing on past lessons, PaG attained a 92% performance on the HumanEval benchmark. However, the current scope of this methodology is largely confined to programming-centric tasks, similar to those in HumanEval, presenting a limitation in its applicability to more language-intensive tasks. As a prospective avenue for future research, the exploration of incorporating online learning to augment retrospections is promising, potentially allowing models to mirror the continual learning and adaptability inherent in human cognition and addressing a broader array of tasks with increased efficacy and precision.

References

[1] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

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

[3] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

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

[5] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

[6] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. Binding language models in symbolic languages. *arXiv preprint arXiv:2210.02875*, 2022.

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

[8] Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. *arXiv preprint arXiv:2207.14502*, 2022.

[9] Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. Unnatural instructions: Tuning language models with (almost) no human labor. *arXiv preprint arXiv:2212.09689*, 2022.

[10] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023.

[11] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[12] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE, 2023.

[13] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

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

[15] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[16] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[17] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

[18] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*, 2023.

[19] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[20] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.

[21] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[22] Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. *arXiv preprint arXiv:2308.07921*, 2023.

[23] Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*, 2022.

A Code Interpreter System Prompt

SYSTEM_PROMPT

As an advanced language model, you can generate code as part of your responses. To make the code more noticeable and easier to read, please encapsulate it within triple backticks. For instance, if you're providing Python code, wrap it as follows:

```
print('hellow world')
```

Wrapped code block will automatically be executed and appended to the prompt.

```
hellow world
```

After presenting the results from the code, you will provide a useful explanation or interpretation of the output to further aid your understanding. Additionally, when generating plots or figures,
You'll save them to a specified path, like ./tmp/plot.png so that they can be viewed. After saving the plot, I'll use the following markdown syntax to display the image at the end of the response:

```
![plot]('./tmp/plot.png')
```

You are using a Jupyter Notebook currently.
This approach allows me to visually present data and findings.

B Retrospections Example Trajectory

Retrospections

From Previous Similar Questions:
The `largest_number` function sorts a list of digits in descending order and joins them to form the largest possible number. However, this is not directly applicable to the `rolling_max` problem.
Application to the Question:
The `rolling_max` function requires maintaining a rolling maximum value as we traverse through the list. This is different from the `largest_number` function which sorts the entire list first. For `rolling_max`, we need to keep track of the maximum value found so far and update it as we encounter larger numbers in the list. The output list should contain the maximum value found at each point in the input list. The `largest_number` function does not provide a useful trajectory for solving this problem.

Instruction

Write a Python script to solve the following problem:

```python
from typing import List, Tuple

def rolling_max(numbers: List[int]) -> List[int]:
  # From a given list of integers, generate a list
  # of rolling maximum element found until given moment
  # in the sequence.
  >>> rolling_max([1, 2, 3, 2, 3, 4, 2])
  [1, 2, 3, 3, 3, 4, 4]
```

Ensure the solution is verified by printing the expected output.

Assistant

Here is the Python script that solves the problem:

```python
from typing import List

def rolling_max(numbers: List[int]) -> List[int]:
    max_so_far = numbers[0]
    result = []
    for num in numbers:
        if num > max_so_far:
            max_so_far = num
        result.append(max_so_far)
    return result

print(rolling_max([1, 2, 3, 2, 3, 4, 2]))
```

```
RESULT
[1, 2, 3, 3, 3, 4, 4]
```

Additional Test Cases for Verification:

```python
print(rolling_max([1]))  # Single element
print(rolling_max([5, 5, 5, 5]))  # All elements are the same
print(rolling_max([9, 8, 7, 6, 5, 4, 3, 2, 1]))  # Descending order
print(rolling_max([]))  # Empty list
```

```
RESULT
[1]
[5, 5, 5, 5]
[9, 9, 9, 9, 9, 9, 9, 9, 9]
Error: IndexError: list index out of range
```

Error Resolution and Final Implementation:
The function fails when the input list is empty, as the code tries to access the first element without prior existence verification. A conditional return for empty lists has been added to address this.

```python
from typing import List

def rolling_max(numbers: List[int]) -> List[int]:
    if not numbers:
        return []
    max_so_far = numbers[0]
    result = []
    for num in numbers:
        if num > max_so_far:
            max_so_far = num
        result.append(max_so_far)
    return result

# Re-verification with the adjusted implementation
print(rolling_max([]))  # Empty list
```

```
RESULT
[]
```

The corrected implementation now successfully handles all edge cases, producing accurate results consistently.