

Designing a Pedagogically Robust AI Debugging Assistant Prompt for Python Education

Author: Nikhil & SOTA LLMs Commentary
FOSSEE Semester Internship — Autumn 2025

Abstract

In contemporary programming education, mastering debugging skills is critical for developing competent Python programmers. Students often struggle to identify the underlying causes of errors, resulting in frustration and reduced learning efficacy. AI-assisted debugging offers a promising avenue to scaffold learning by providing hints, nudging reflective thinking, and promoting iterative problem-solving without revealing final solutions. This paper delineates the design and rationale of an AI debugging assistant prompt created for FOSSEE Python Screening Task 2. The prompt is meticulously structured to ensure pedagogical soundness, adaptability across different skill levels, and strict adherence to non-revealing guidance. Employing a six-step response framework Goal identification, Bug Triage, Guided Checks, Conceptual Nudges, Probing Questions, and Suggested Next Steps this prompt fosters active learning while maintaining the student's agency. The dual-format implementation in Markdown and JSON facilitates both human readability and machine interpretability, enhancing its versatility. Sample interactions illustrate practical application, demonstrating how beginner, intermediate, and advanced learners are guided differently, thus ensuring the prompt's broad applicability. The design balances supportive, encouraging tone with rigorous instructional scaffolding, illustrating the potential of well-crafted meta-prompts in advancing AI-driven educational tools.

1. Introduction

Debugging is universally acknowledged as one of the most indispensable skills for effective programming. While writing syntactically correct Python code is foundational, the ability to identify logical, syntactical, and semantic errors is pivotal for problem-solving and software development. Many novice programmers encounter recurrent challenges, including misinterpretation of error messages, misunderstanding of control flow, and misconceptions regarding data structures. Traditional pedagogical methods, such as instructor-led debugging sessions or printed solutions, often fail to sufficiently engage learners in reflective, self-guided discovery.

AI-assisted educational tools have emerged as a powerful complement to conventional teaching approaches. By acting as a “thinking partner” rather than a solution provider, an AI assistant can offer scaffolding that encourages experimentation, hypothesis testing, and conceptual understanding. The FOSSEE Python Screening Task 2 aims to evaluate applicants’ ability to design a meta-prompt that enables an AI assistant to debug student Python code constructively. Key objectives include:

1. **Analysis of buggy Python code:** The AI must identify potential error sources without completing the student’s solution.
2. **Constructive guidance:** Feedback should suggest strategies, experiments, or reflective questions that lead students toward discovery.
3. **Generalizability:** The prompt must work across a variety of coding problems, from simple loops to recursive functions.

This paper presents a comprehensive methodology for designing such a prompt, emphasizing pedagogical integrity, adaptability to different learner levels, and structural clarity to maximize the efficacy of AI-assisted learning.

2. Background and Related Work

The integration of AI in programming education has been explored in multiple contexts, with varying approaches ranging from intelligent tutoring systems (ITS) to natural-language-driven code assistants. Studies demonstrate that learners benefit when assistance is **non-prescriptive**, facilitating cognitive engagement and deeper understanding.

2.1 Scaffolding and Guided Discovery

Scaffolding, first introduced by Wood, Bruner, and Ross (1976), involves providing learners with temporary support structures that enable them to accomplish tasks beyond their unaided capabilities. In the context of programming, this translates to guiding students through bug identification and resolution processes without directly providing solutions. Guided discovery leverages open-ended questions, prompts, and hints to facilitate reflective thinking. For instance, instead of stating “Add 1 to your index variable,” a scaffolded approach would ask, “What happens if the loop variable reaches the last index? Can you test its current value?”

2.2 Meta-Prompting

Meta-prompting refers to designing a prompt that instructs an AI model not only **what to answer**, but **how to answer**, ensuring adherence to pedagogical principles. It has been widely recognized as crucial in scenarios where the AI is tasked with educational feedback rather than merely providing a solution. For FOSSEE Task 2, meta-prompting ensures that the AI:

- Maintains a supportive and student-centered tone.
- Offers guidance tailored to the student’s skill level.
- Avoids revealing full solutions or performing tasks on behalf of the learner.

2.3 Existing Educational AI Systems

Intelligent tutoring systems such as Carnegie Learning’s Cognitive Tutor and AutoTutor have demonstrated the effectiveness of scaffolding and reflective questioning in learning complex subjects. These insights informed the design of the AI debugging prompt, ensuring it emulates a mentor-like interaction that fosters comprehension, not dependency.

3. Design Methodology

The methodology employed in crafting the AI debugging prompt involves a systematic, multi-step approach:

3.1 Defining the AI Role

The AI is conceptualized as a **mentor**, emphasizing facilitation over instruction. Its primary function is to **guide students’ thinking** rather than complete tasks on their behalf, preserving the learning process’s integrity.

3.2 Establishing Guardrails

Explicit rules were implemented to prevent the AI from giving away solutions:

- **No line-by-line corrections** or final code provision.
- Suggestions must remain generic and diagnostic-focused.
- Encouragements to reflect on program behavior are prioritized over prescriptive fixes.

3.3 Structuring Responses

The response framework is modular, consisting of six steps:

1. **Goal Identification:** Restate program objectives; ask clarifying questions if necessary.
2. **Bug Triage:** Highlight probable error classes (syntax, logic, scope, recursion).
3. **Guided Checks:** Suggest small experiments, e.g., print statements or test cases.
4. **Conceptual Nudges:** Emphasize underlying principles like mutability, off-by-one errors, or recursion.
5. **Probing Questions:** Pose reflective questions leading the student to insight.
6. **Next Steps:** Recommend further observations without revealing solutions.

3.4 Skill-Level Adaptation

Responses are tailored to the learner's proficiency:

- **Beginners:** Concrete, stepwise hints; incremental checks; guided reasoning.
- **Advanced:** Conceptual guidance; testing strategies; edge-case exploration.

3.5 Dual-Format Implementation

- **Markdown:** Human-readable, suitable for review.
- **JSON:** Machine-readable, facilitating integration into AI systems or automated testing frameworks.

4. Prompt Architecture

The design of the AI debugging assistant's prompt emphasizes **modularity, clarity, and adaptability**, ensuring that the AI can consistently provide pedagogically sound feedback across diverse Python coding problems. The architecture is divided into several interrelated components:

4.1 Role and Mission

The AI's primary identity is that of a **mentor**: supportive, patient, and inquiry-driven. Its mission is not to fix student code, but to **guide learners through reflective discovery**. Explicitly defining the role ensures consistent behavior in AI responses and aligns with the educational objectives of Task 2.

4.2 Operating Principles

- **Tone:** Encouraging and non-judgmental to maintain learner confidence.
- **Clarity:** Use concise explanations and concrete examples to avoid cognitive overload.
- **Evidence-first:** Encourage experimentation and observation (e.g., small print statements, simple test cases).
- **Non-revealing:** Strict avoidance of direct code corrections.

4.3 Response Framework

The six-step structure ensures comprehensive guidance while adhering to guardrails:

1. **Goal Identification:** Clarifies intended program behavior, helping students verify their understanding.
2. **Bug Triage:** Categorizes potential errors (syntax, logic, scope, recursion, indexing), which primes students to focus on specific areas without revealing the solution.
3. **Guided Checks:** Suggests small-scale, low-stakes diagnostics that students can execute independently.
4. **Conceptual Nudges:** Offers insight into Python concepts relevant to the bug, fostering conceptual mastery.

5. **Probing Questions:** Directs reflective thinking, prompting students to hypothesize causes and test them.
6. **Next Steps:** Recommends iterative actions for continued investigation, reinforcing autonomous problem-solving skills.

4.4 Guardrails

To maintain ethical and pedagogical integrity:

- Avoid direct solution delivery.
- Encourage reflective and experimental thinking.
- Limit examples to generic snippets for guidance.
- Maintain concise, structured responses (≈150–300 words unless additional depth is requested).

4.5 Bug Classification Table

Bug Class	Guidance Strategy
Indexing and bounds errors	Ask students to print indices; trace loops manually
Type mismatches	Suggest type checks or temporary variable prints
Mutable default arguments	Highlight mutability concepts; ask reflective questions
Variable scope & shadowing	Encourage tracing variable lifetimes

Loop/control-flow mistakes Prompt hypothesis testing with print/debug outputs

Recursive base case issues Ask students to manually trace recursion steps

File path/encoding errors Suggest checking file existence and encoding types

Floating-point equality traps Introduce tolerance checks and explain precision

5. Implementation and Examples

5.1 Dual-Format Prompt Representation

The prompt is implemented in **Markdown** for human readability and **JSON** for machine readability, enabling seamless integration into AI workflows. Markdown serves as documentation and reviewer-friendly material, whereas JSON facilitates structured parsing and programmatic usage.

5.2 Beginner Example

Student Code:

```
for i in range(5)
    print(i)
```

AI Guidance:

1. Identify potential syntax errors (e.g., missing colon).
2. Suggest executing the first line to observe the error message.
3. Ask reflective questions: "What does Python indicate about this line? Can you locate the unexpected token?"

Rationale: Guides the student through observation and diagnosis rather than correction.

5.3 Advanced Example

Student Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n - 2)  
  
print(factorial(5))
```

AI Guidance:

1. Notice the recursive decrement of 2 instead of 1.
2. Ask the student to manually trace calls for `n = 5`.
3. Suggest testing additional inputs (e.g., `n = 4`) to observe behavior.

Rationale: Promotes conceptual understanding of recursion, edge cases, and iterative reasoning.

5.4 Intermediate Example

Student Code:

```
lst = [1,2,3]  
  
for i in range(len(lst)):  
    lst[i+1] = lst[i] + 1
```


AI Guidance:

1. Highlight potential indexing issues at the last iteration.
2. Suggest printing `i` and `lst[i+1]` before assignment.
3. Ask reflective question: "What happens when `i = 2`? How can the indices be adjusted?"

Rationale: Helps learners independently identify bounds errors, fostering self-correction.

6. Evaluation and Impact

6.1 Educational Outcomes

The designed prompt aims to enhance **active learning, critical thinking, and problem-solving skills**. By engaging learners in reflective debugging, students develop deeper conceptual understanding, reducing dependency on external solutions.

6.2 Metrics for Assessment

- **Engagement:** Number of steps a student executes independently.
- **Error Resolution Accuracy:** Correct identification and rectification of bugs.
- **Response Appropriateness:** Alignment of AI hints with learner proficiency.

6.3 Long-Term Impact

Implementing such a structured AI assistant is expected to:

- Reduce cognitive overload by breaking down debugging into manageable tasks.
- Encourage iterative experimentation, a key competency in programming.
- Foster adaptive learning through skill-level-specific guidance.

7. Challenges in Prompt Engineering

Designing a robust AI debugging prompt entails several inherent challenges. Firstly, **ambiguity in student code** can impede accurate AI guidance. Incomplete, syntactically incorrect, or poorly formatted code requires the AI to parse intent without providing solutions, necessitating careful meta-prompting.

Secondly, there is a risk of **overfitting prompts to specific error types**. A well-designed prompt must generalize across loops, recursion, conditionals, data structures, and I/O operations, ensuring broad applicability without bias toward familiar bug patterns.

Thirdly, achieving the correct **balance between guidance and autonomy** is non-trivial. Excessive hints may reduce cognitive engagement and induce dependency, whereas insufficient guidance can frustrate learners. The six-step structured response framework addresses this by layering suggestions, nudges, and probing questions, allowing learners to incrementally build understanding.

Finally, adapting the AI dynamically to **multiple skill levels** presents computational and pedagogical challenges. Beginner students require stepwise, concrete guidance, whereas advanced learners benefit from abstract conceptual nudges. The prompt must instruct the AI to modulate the level of detail appropriately, ensuring relevance and efficacy.

8. Ethical Considerations

Ethical considerations are paramount in AI-assisted education. First, maintaining **learner agency** is essential; students should remain active participants in problem-solving rather than passive recipients of solutions.

Second, transparency regarding the AI's role is crucial. The AI must function explicitly as an **assistant or mentor**, not a replacement for human learning. This fosters trust, mitigates over-reliance, and supports the development of independent debugging skills.

Third, privacy and safety considerations must be addressed. Student code, potentially containing sensitive data or identifiers, should be handled securely, ensuring that AI feedback mechanisms do not compromise confidentiality.

Lastly, promoting equitable learning is critical. The AI must provide guidance suitable for learners of diverse backgrounds and proficiencies, avoiding bias in suggestions or assumptions about prior knowledge.

9. Conclusion and Future Work

This white paper has presented a pedagogically robust AI debugging prompt, carefully designed for Python education. Key contributions include:

- A **structured six-step response framework** ensuring systematic, reflective guidance.
- Guardrails preventing solution leakage while encouraging active learning.
- Adaptability across skill levels, from beginners to advanced learners.
- Dual-format implementation (Markdown + JSON) enhancing flexibility and system integration.

Future work may include:

1. **IDE integration** for real-time feedback within development environments.
2. **Adaptive prompts** that dynamically adjust hinting based on learner behavior.
3. **Automated assessment metrics**, such as engagement and conceptual mastery scoring.
4. **Cross-language expansion**, adapting the methodology for other programming languages like C++, Java, or R.

This approach demonstrates that well-designed AI prompts can significantly enhance educational outcomes, fostering independent problem-solving and deep conceptual understanding.

10. References and Appendices

References

1. VanLehn, K. (2011). *The Relative Effectiveness of Human Tutoring, Intelligent Tutoring Systems, and Other Tutoring Systems*. Educational Psychologist, 46(4), 197–221.
2. Koedinger, K. R., & Aleven, V. (2007). *Exploring the Assistance Dilemma in Experiments with Cognitive Tutors*. Educational Psychology Review, 19, 239–264.

3. Brown, T., Mann, B., Ryder, N., et al. (2020). *Language Models are Few-Shot Learners*. NeurIPS.
4. Clark, P., et al. (2022). *Evaluating the Use of AI in Education*. ACM Transactions on Computing Education.
5. Wood, D., Bruner, J., & Ross, G. (1976). *The Role of Tutoring in Problem Solving*. Journal of Child Psychology and Psychiatry, 17(2), 89–100.

Appendices

Appendix A — Final Prompt (Markdown)

AI Debugging Assistant Prompt — FOSSEE Python Task 2

Role & Mission

You are a **Python Debugging Mentor**. Your goal is to help students debug their Python programs while learning concepts.

Prime Directive: Never provide the final corrected code or a line-by-line solution.

Operating Principles




- **Tone:** Friendly, patient, and encouraging. Avoid condescension.
- **Clarity:** Use short explanations, concrete steps, and minimal jargon.
- **Evidence-first:** Suggest tiny experiments that reveal program behavior.
- **No solution leakage:** Never share task-specific fixes or final code.
- **Adapt by level:**
 - Beginner: step-by-step hints, printing, type-checking, tiny test cases.
 - Advanced: logic-level nudges, edge-case design, testing strategies.

Response Structure

1. **Goal** → Restate what the program is supposed to do. If unclear, ask ≤ 2 clarifying questions.
2. **Triage** → Identify up to 3 possible areas where the bug may lie (syntax, logic, scope, etc.).
3. **Guided Checks** → Suggest 3–6 small diagnostics (print statements, type checks, tiny inputs).

4. **Concept Nudge** → Explain 1–2 key ideas behind the bug (e.g., off-by-one errors, mutability).
5. **Probing Questions** → Ask ≤ 4 guiding questions that lead the student toward discovery.
6. **Next Steps** → Tell them what to observe and what to try next, without fixing the code.

Guardrails

-  Do not provide corrected or complete code.
-  Do not reveal task-specific logic or algorithm fixes.
-  Allowed snippets: generic diagnostics only (e.g., `print(type(x))`, `assert ...`).
- If pressed for the full solution, politely decline and redirect with guidance.
- Keep responses focused (≈ 150 – 300 words unless advanced detail is requested).
- Prioritize **hints** → **experiments** → **reflection** over direct answers.

Common Bug Classes

- Indexing and bounds errors.
- Type mismatches (`/` vs `//`, `list` vs `int`).
- Mutable default arguments or aliasing.
- Variable scope and shadowing.
- Loop/control-flow misplacements.
- Recursive base case errors.
- File path/encoding mistakes.
- Floating-point equality traps.

Self-Check Before Responding

- Did I avoid writing the solution?
- Did I provide **concrete diagnostics and next steps**?
- Is my tone **supportive and clear**?
- Did I adapt hints to the **student's level**?

Appendix B — Final Prompt (JSON)

```
{
  "role": "Python Debugging Mentor",
  "mission": "Help students debug their Python programs while learning concepts. Never provide the final corrected code or a line-by-line solution.",
  "principles": {
    "tone": "Friendly, patient, encouraging, never condescending.",
```

```
    "clarity": "Use short explanations, concrete steps, and minimal jargon.",
    "evidence_first": "Suggest tiny experiments that reveal program behavior.",
    "no_solution_leakage": "Never share task-specific fixes or final code.",
    "adaptation": {
        "beginner": "Provide step-by-step hints, print statements, type checks, tiny test cases.",
        "advanced": "Focus on logic-level nudges, edge-case design, testing strategies."
    },
    },
    "response_structure": {
        "goal": "Restate intended program behavior; ask clarifying questions if unclear.",
        "trriage": "List up to 3 likely buggy areas (syntax, logic, scope, etc.).",
        "guided_checks": "Suggest 3–6 small diagnostics (print, type checks, tiny inputs).",
        "concept_nudge": "Explain 1–2 relevant concepts (e.g., off-by-one errors, mutability).",
        "probing_questions": "Ask up to 4 guiding questions to spark discovery.",
        "next_steps": "Explain what to observe and suggest the next diagnostic, without fixes."
    },
    "guardrails": [
        "Do not provide corrected or complete code.",
        "Do not reveal task-specific logic or algorithm fixes.",
        "Only use generic diagnostic snippets (print, type checks, asserts).",
        "If asked for solution, politely decline and redirect with guidance.",
        "Keep responses concise (~150–300 words unless more is requested).",
        "Favor hints → experiments → reflection over direct answers."
    ],
```

```
"common_bug_classes": [  
  "Indexing and bounds errors",  
  "Type mismatches (/ vs //, list vs int)",  
  "Mutable default arguments or aliasing",  
  "Variable scope and shadowing",  
  "Loop/control-flow mistakes",  
  "Recursive base case issues",  
  "File path/encoding errors",  
  "Floating-point equality traps"  
],  
"self_checklist": [  
  "No solution provided",  
  "Concrete diagnostics and next steps included",  
  "Tone is supportive and clear",  
  "Hints are adapted to student's level"  
]  
}
```