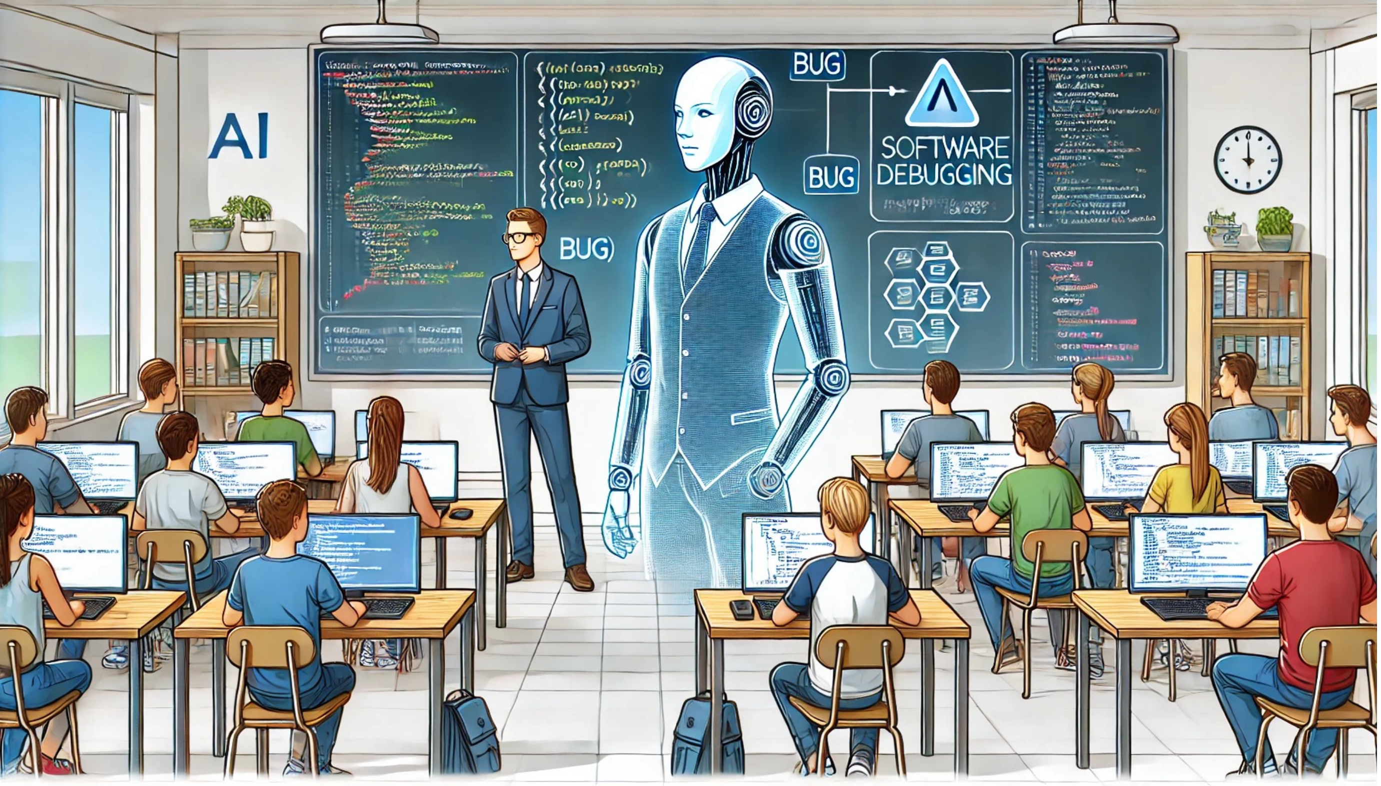


Simulated Interactive Debugging

22/01/2025 – Shonan Meeting 217: Trusted Automatic Programming

Yannic Noller
yannic.noller@rub.de

Disclaimer

- this is work in progress, feedback appreciated!
- short paper is available on arXiv

- joint work with:
    - Erick Chandra (SUTD)
    - Srinidhi HC (SUTD)
    - Kenny Choo (SUTD)
    - Cyrille Jegourel (SUTD)
    - Oka Kurniawan (SUTD)
    - Christopher M. Poskitt (SMU)

arXiv:2501.09694v1 [cs.SE] 16 Jan 2025

Simulated Interactive Debugging

Yannic Noller*
yannic.noller@acm.org
Ruhr University Bochum
Germany

Erick Chandra
erick_chandra@sutd.edu.sg
Singapore University of
Technology and Design
Singapore

Srinidhi HC
srinidhi.hc@gmail.com
Singapore University of
Technology and Design
Singapore

Kenny Choo
kenny_choo@sutd.edu.sg
Singapore University of
Technology and Design
Singapore

Cyrille Jegourel
cyrille_jegourel@sutd.edu.sg
Singapore University of
Technology and Design
Singapore

Oka Kurniawan
oka_kurniawan@sutd.edu.sg
Singapore University of
Technology and Design
Singapore

Christopher M. Poskitt
cposkitt@smu.edu.sg
Singapore Management
University
Singapore

Abstract

Debugging software, i.e., the localization of faults and their repair, is a main activity in software engineering. Therefore, effective and efficient debugging is one of the core skills a software engineer must develop. However, the teaching of debugging techniques is usually very limited or only taught in indirect ways, e.g., during software projects. As a result, most Computer Science (CS) students learn debugging only in an ad-hoc and unstructured way. In this work, we present our approach called *Simulated Interactive Debugging* that interactively guides students along the debugging process. The guidance aims to empower the students to repair their solutions and have a proper "learning" experience. We envision that such guided debugging techniques can be integrated into programming courses early in the CS education curriculum. To perform an initial evaluation, we developed a prototypical implementation using traditional fault localization techniques and large language models. Students can use features like the automated setting of breakpoints or an interactive chatbot. We designed and executed a controlled experiment that included this IDE-integrated tooling with eight undergraduate CS students. Based on the responses, we conclude that the participants liked the systematic guidance by the assisted debugger. In particular, they rated the automated setting of breakpoints as the most effective, followed by the interactive debugging and chatting, and the explanations for how breakpoints were set. In our future work, we will improve our concept and implementation, add new features, and perform more intensive user studies.

Keywords

Intelligent Tutoring, Debugging, Software Engineering, Education

1 Introduction

Despite the paradigm shift towards Artificial Intelligence (AI)-assisted software development, particularly with the emergence of Large Language Models (LLMs), there remains a high demand for qualified software engineers capable of developing reliable, high-quality software. With more auto-generated code, it will become even more critical that junior software developers can effectively debug software and solve bugs, even in code they have not written themselves. Radermacher et al. [28] explored knowledge deficiencies of graduate students from an industry perspective. One of the most frequently mentioned issues with software tools has been version control systems and debuggers. While we have seen the integration of version control systems like Git in the Computer Science (CS) and Software Engineering (SE) curriculum, e.g., with the usage of GitHub Classroom, we have not seen much innovation regarding debugging education. Michaeli and Romeike [25] noted that there are only few studies investigating the "explicit teaching of debugging" and students are often left alone to learn debugging "the hard way". Our experience confirms this observation: most students still learn (interactive) debugging of code in an ad-hoc and unstructured way by trial and error. Using "print/log" statements is often the only extent of students' experience in debugging. They do not receive proper guidance in efficient program comprehension and how to debug programs effectively. Current CS/SE research efforts focus on general programming education, which helps to provide scalable alternatives to mentor and guide a rising number of students. Automated Program Repair (APR) techniques [14, 17, 34] can help to produce patches for incorrect submissions, which then can be used as a basis for personalized feedback, or even automated grading [12]. In fact, we already have seen the application of such techniques in the CS/SE curriculum focusing on repairing students' solutions and providing feedback [11, 19]. However, the existing related work does not address actual debugging.

> Our research objective is to *understand* students' needs in learning to debug and provide *automated* means to *guide* them effectively along the debugging process.

As part of this objective, we strive to seek answers to questions like "How can we guide the debugging process using artifacts generated from APR/SE techniques?" and "How can LLMs be integrated into the debugging process for the benefit of the student?"

As a first step to achieve our research objective, we propose the concept of *Simulated Interactive Debugging*. The essence of this concept is to create a simulation of an interactive debugging experience; "simulated" because the solution is known in the education context due to given reference implementations and artifacts like fault locations and patches that can be auto-generated with SE and APR techniques. Our goal is to provide a controlled but supportive environment, in which the students debug the problem on their own. Similar to a human tutor who would not directly reveal the solution and instead help the student to understand and fix the

---

*Corresponding Author.

1

https://arxiv.org/pdf/2501.09694

RUHR
UNIVERSITÄT
BOCHUM

Context: CS/SE Education

- providing automated help to CS/SE students: Intelligent Tutoring Systems
- debugging education ~ teaching interactive debugging
- in our experience:
  - learning debugging mostly happens adhoc and in an unstructured way
  - students rely on print/log statements

Relevance for “Trusted Automatic Programming“

- Trust in “using“ auto-generated code: being able to **debug** it?
- Students should learn debugging of code effectively.
- How to interact with AI as student? Trust the AI tutor?

Our **research objective** is to understand students’ needs in learning to **debug** and provide **automated** means to guide them effectively along the debugging process.

Also, teach the usage of **AI-based** tools like ChatGPT. By constructing a chatbot that uses **LLMs**, we can control the prompts and the contents of responses, and hence, still foster a **learning experience** for the students, e.g., by filtering direct solutions.

Related Work

- Debugging education:
  - “debugging is a skill that does not immediately follow from the ability to write code. Rather ... it must be taught” [1]
  - Automated generation of code debugging exercises [2]
- Intelligent Tutoring Systems
  - repair, feedback, grading
  - also with LLMs
- But overall no focus on (guided) interactive debugging

---

[1] Kessler, C.M. and Anderson, J.R., 1986, June. A model of novice debugging in LISP. In Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers (pp. 198-212).

[2] V.-A. Pădurean, P. Denny, and A. Singla. 2025. BugSpotter: Automated Generation of Code Debugging Exercises . In Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025).

Concept: Simulated Interactive Debugging

- create a **simulation** of an **interactive debugging** experience
- **"simulated"** ~ the solution is known in the education context
  - reference implementations is given
  - artifacts like fault locations and patches can be auto-generated
  - provides controlled but supportive environment
- requires integration into tools like **Integrated Development Environments** (IDEs)
  - to guide students actively during the debugging
- inputs:
  - from lecturer: programming assignment, reference implementation, public + private test cases
  - from student: (buggy) solution, (additional test case)

Envisioned Dialogue (1/2)

Student

Assistant

You just run your code for the **factorial** function and it fails test case 2. Do you want me to explain the issue?

Yes!

For n=2, the loop will run only once because range(1,2) generates *only* 1. What might be happening to the calculation of the factorial in this case?

clarify/explain failing test cases and the observed failures

I don't know... where is the error in my code?

Okay, no problem. Use the **Set Breakpoints** feature to identify the relevant lines to identify the error. Click on the lines for an explanation.

automatically set breakpoints at locations of interest

Envisioned Dialogue (2/2)

Assistant

Student

I am still stuck. Can you give me a hint?

Absolutely! For the input *n=2*, the loop runs only once because *range (1,2)* generates only 1. Think about what the initial value of *fact* is and how it gets updated in each iteration of the loop.

incremental guidance, e.g., via progressive hints

I see the issue now. Are there any variables I should focus on?

Yes, focus on the variables *fact* and *i* within the loop. Consider how *fact* is initialized and updated in each iteration of the loop. Take a moment to review and adjust how *fact* is being updated within the loop.

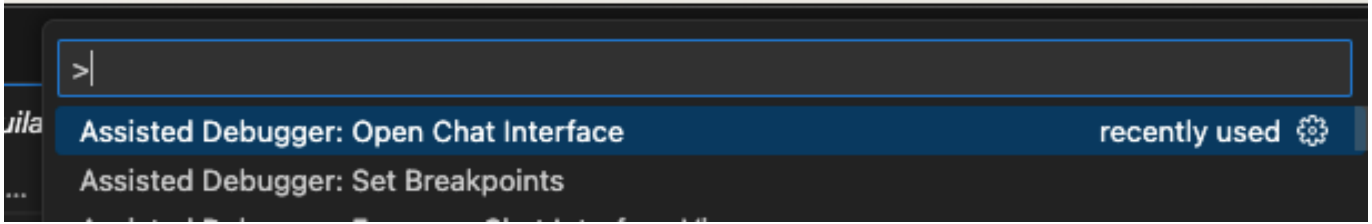
identify and highlight interesting program variables

Features of Simulated Interactive Debugging

(1) **clarify/explain** failing **test cases** and the observed **failures**	LLMs
(2) **automatically set breakpoints** at locations of interest	SBFL, APR
(3) **generate** hints in natural language **explaining** why these breakpoints are of interest	LLMs
(4) **deliver incremental guidance**, e.g., via progressive hints, or partially reveal why a certain test case execution fails	LLMs, APR
(5) **help the student** in the interactive debugging by identifying and highlighting interesting **program variables**	APR, LLMs, static analysis
(6) **test assessment** for	
▪ **lecturers**: Is the test suite strong enough for automated debugging?	mutation testing
▪ **students**: Is the test case correct? Are all relevant parts covered?	execution, coverage

Prototype Implementation

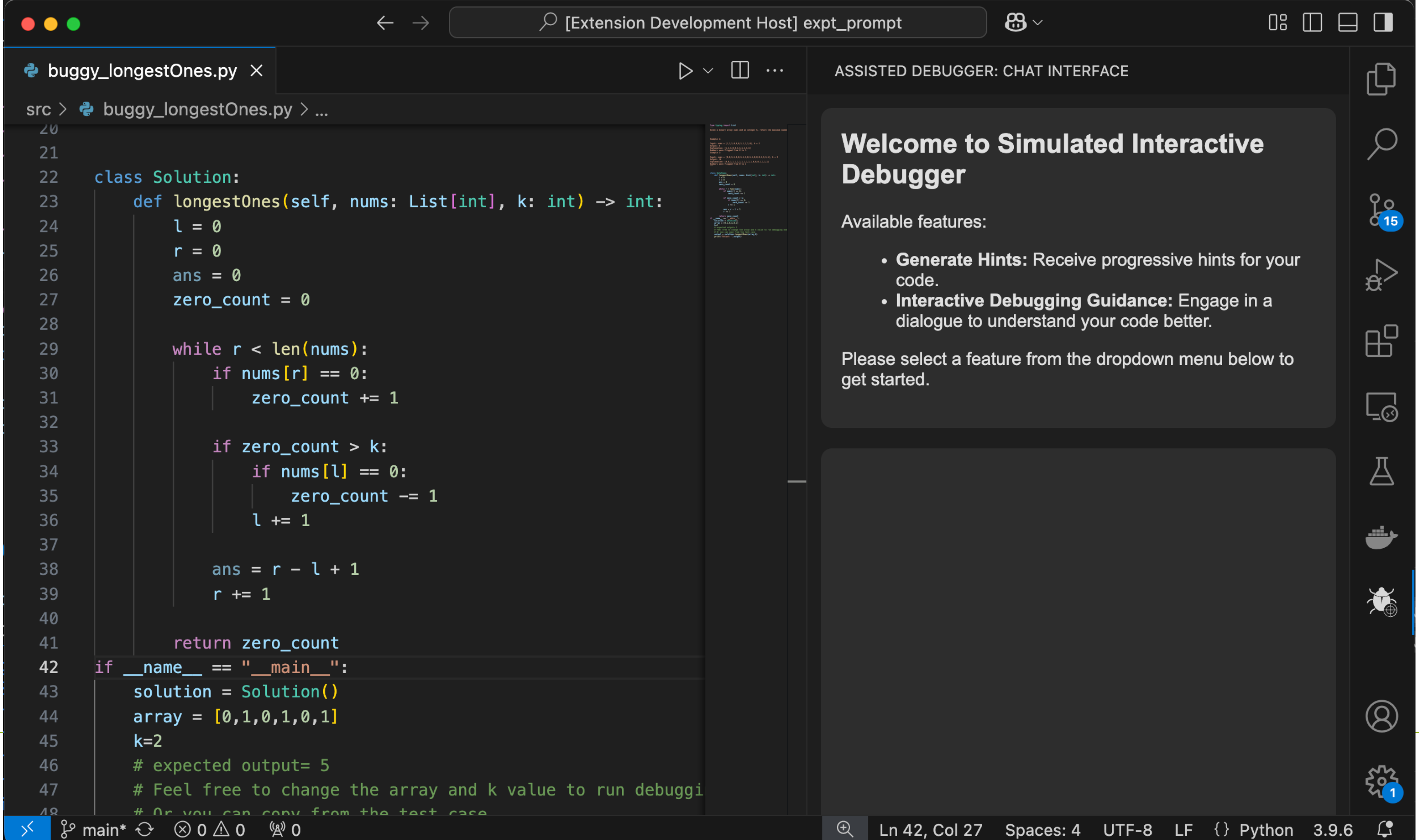
- Visual Studio Code IDE extension
- first focus on supporting Python
- two core features for the assisted debugger:
  - the automatic setting of breakpoints (based on SBFL with FauxPy)
  - and a chat interface providing tailored debugging hints using OpenAI gpt-3.5-turbo
- we omitted the support for the lecturer’s and the student’s test assessment



>|

Assisted Debugger: Open Chat Interface recently used

Assisted Debugger: Set Breakpoints



```python
class Solution:
    def longestOnes(self, nums: List[int], k: int) -> int:
        l = 0
        r = 0
        ans = 0
        zero_count = 0

        while r < len(nums):
            if nums[r] == 0:
                zero_count += 1

            if zero_count > k:
                if nums[l] == 0:
                    zero_count -= 1
                l += 1

            ans = r - l + 1
            r += 1

        return zero_count
if __name__ == "__main__":
    solution = Solution()
    array = [0,1,0,1,0,1]
    k=2
    # expected output= 5
    # Feel free to change the array and k value to run debuggi
    # Or you can copy from the test case
```

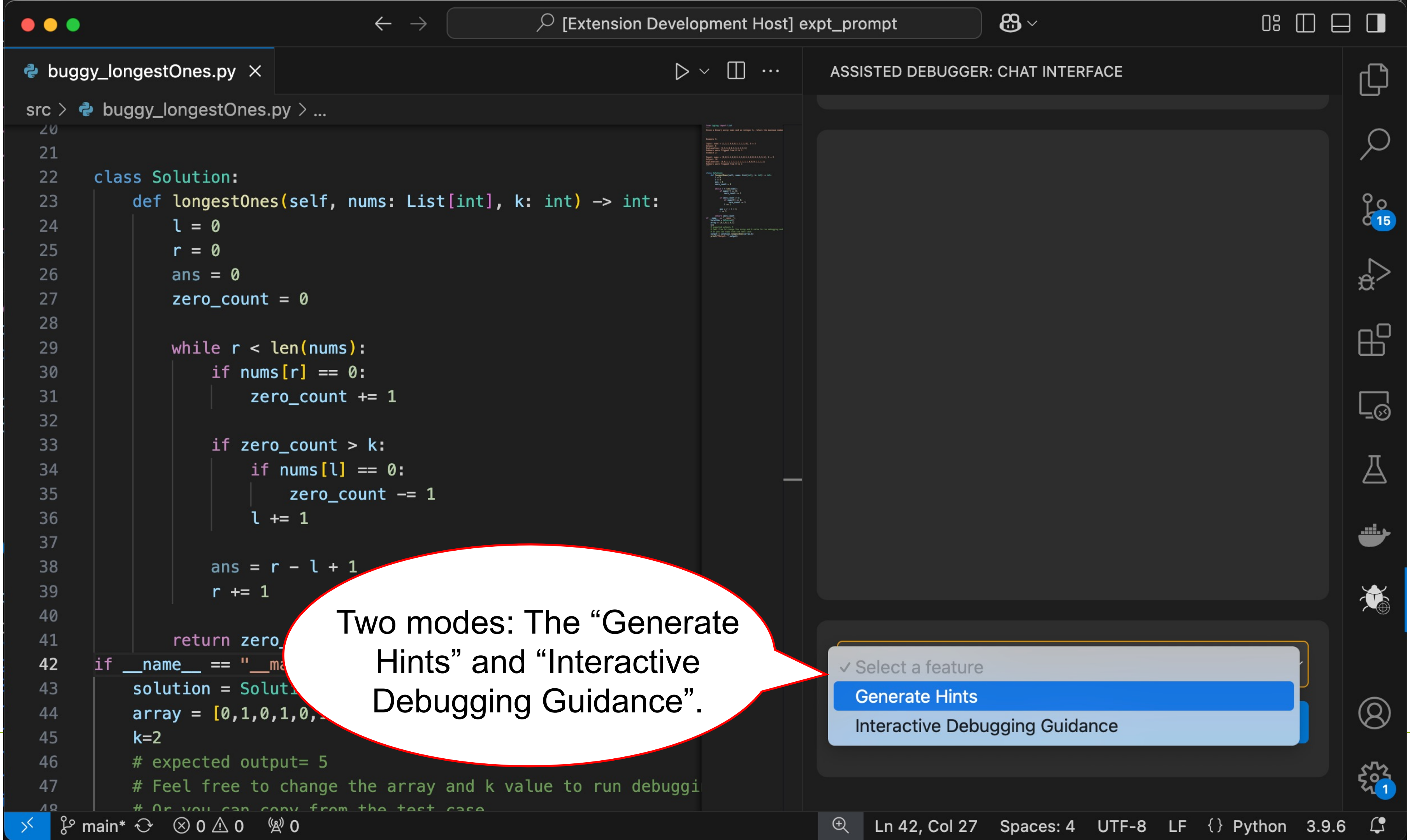
ASSISTED DEBUGGER: CHAT INTERFACE

Welcome to Simulated Interactive Debugger

Available features:

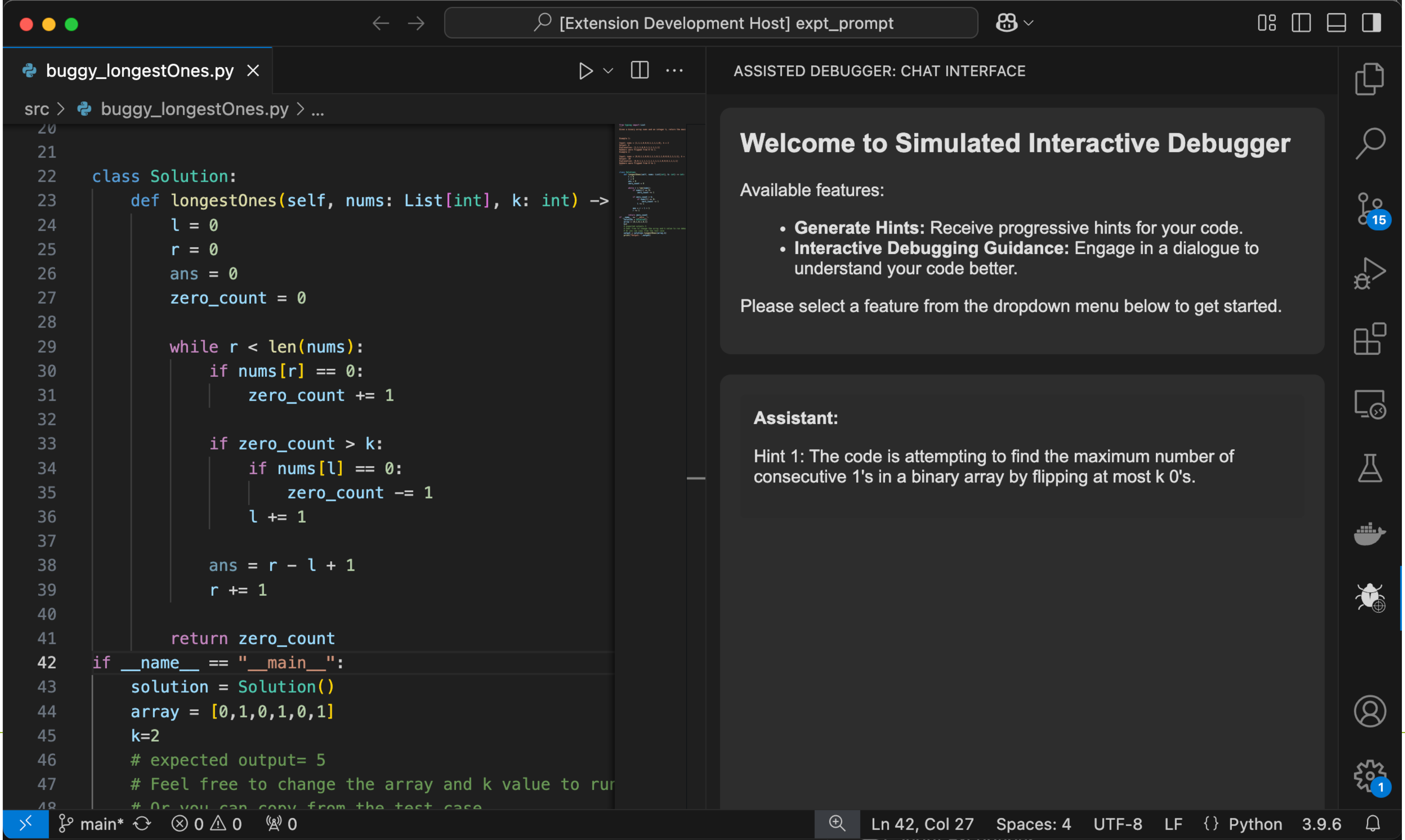
- **Generate Hints:** Receive progressive hints for your code.
- **Interactive Debugging Guidance:** Engage in a dialogue to understand your code better.

Please select a feature from the dropdown menu below to get started.



```
class Solution:
    def longestOnes(self, nums: List[int], k: int) -> int:
        l = 0
        r = 0
        ans = 0
        zero_count = 0

        while r < len(nums):
            if nums[r] == 0:
                zero_count += 1

            if zero_count > k:
                if nums[l] == 0:
                    zero_count -= 1
                l += 1

            ans = r - l + 1
            r += 1

        return zero_
if __name__ == "__ma
    solution = Soluti
    array = [0,1,0,1,0,
    k=2
    # expected output= 5
    # Feel free to change the array and k value to run debuggi
    # Or you can copy from the test case
```

Two modes: The “Generate Hints” and “Interactive Debugging Guidance”.



```
class Solution:
    def longestOnes(self, nums: List[int], k: int) ->
        l = 0
        r = 0
        ans = 0
        zero_count = 0

        while r < len(nums):
            if nums[r] == 0:
                zero_count += 1

            if zero_count > k:
                if nums[l] == 0:
                    zero_count -= 1
                l += 1

            ans = r - l + 1
            r += 1

        return zero_count
if __name__ == "__main__":
    solution = Solution()
    array = [0,1,0,1,0,1]
    k=2
    # expected output= 5
    # Feel free to change the array and k value to rur
    # Or you can copy from the test case
```

Welcome to Simulated Interactive Debugger

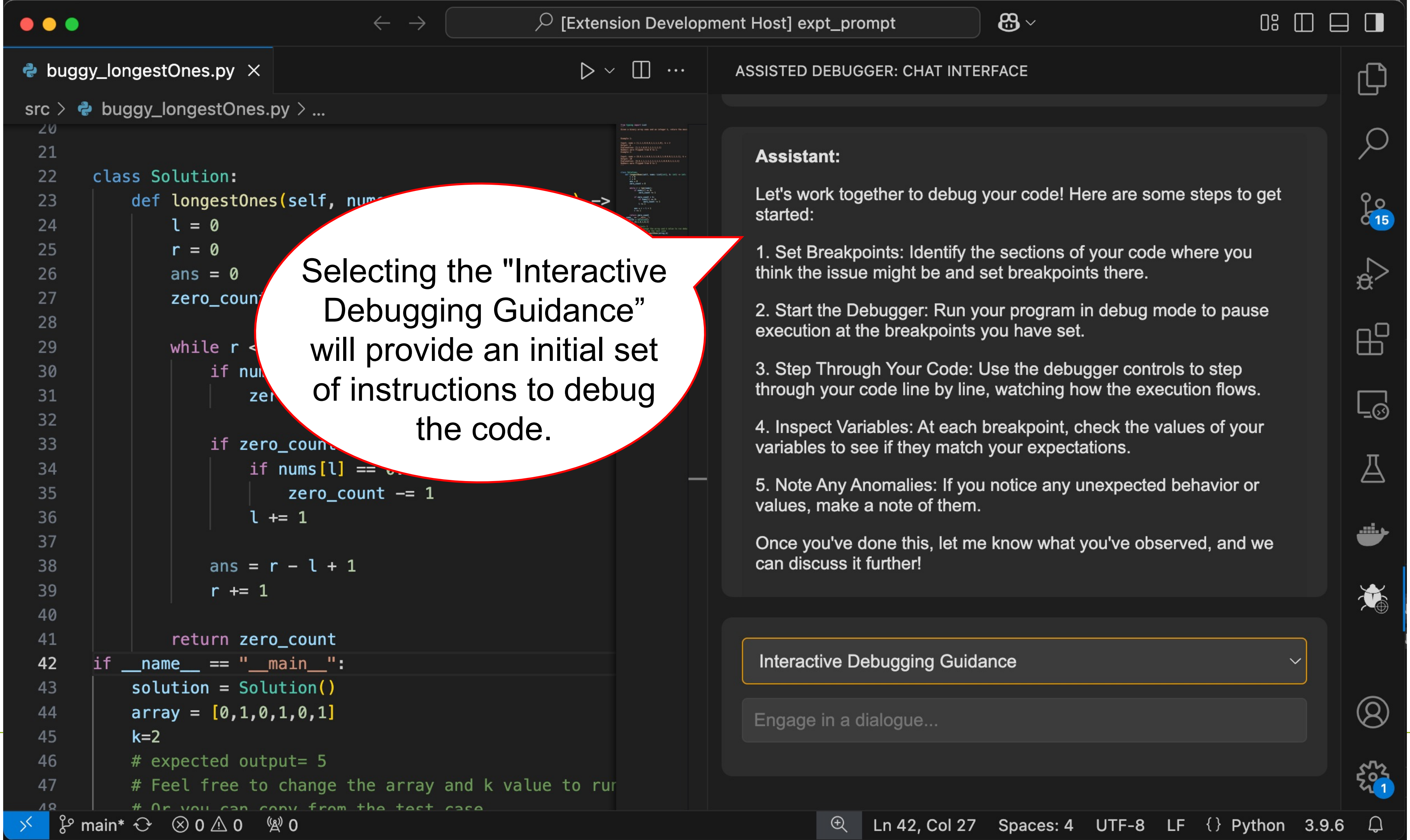
Available features:

- **Generate Hints:** Receive progressive hints for your code.
- **Interactive Debugging Guidance:** Engage in a dialogue to understand your code better.

Please select a feature from the dropdown menu below to get started.

Assistant:

Hint 1: The code is attempting to find the maximum number of consecutive 1's in a binary array by flipping at most k 0's.



```
class Solution:
    def longestOnes(self, num                    ) ->
        l = 0
        r = 0
        ans = 0
        zero_coun

        while r <
            if nu
                zer

            if zero_count
                if nums[l] ==
                    zero_count -= 1
                l += 1

            ans = r - l + 1
            r += 1

        return zero_count
if __name__ == "__main__":
    solution = Solution()
    array = [0,1,0,1,0,1]
    k=2
    # expected output= 5
    # Feel free to change the array and k value to rur
    # Or you can copy from the test case
```

Selecting the "Interactive Debugging Guidance" will provide an initial set of instructions to debug the code.

Assistant:

Let's work together to debug your code! Here are some steps to get started:

1. Set Breakpoints: Identify the sections of your code where you think the issue might be and set breakpoints there.

2. Start the Debugger: Run your program in debug mode to pause execution at the breakpoints you have set.

3. Step Through Your Code: Use the debugger controls to step through your code line by line, watching how the execution flows.

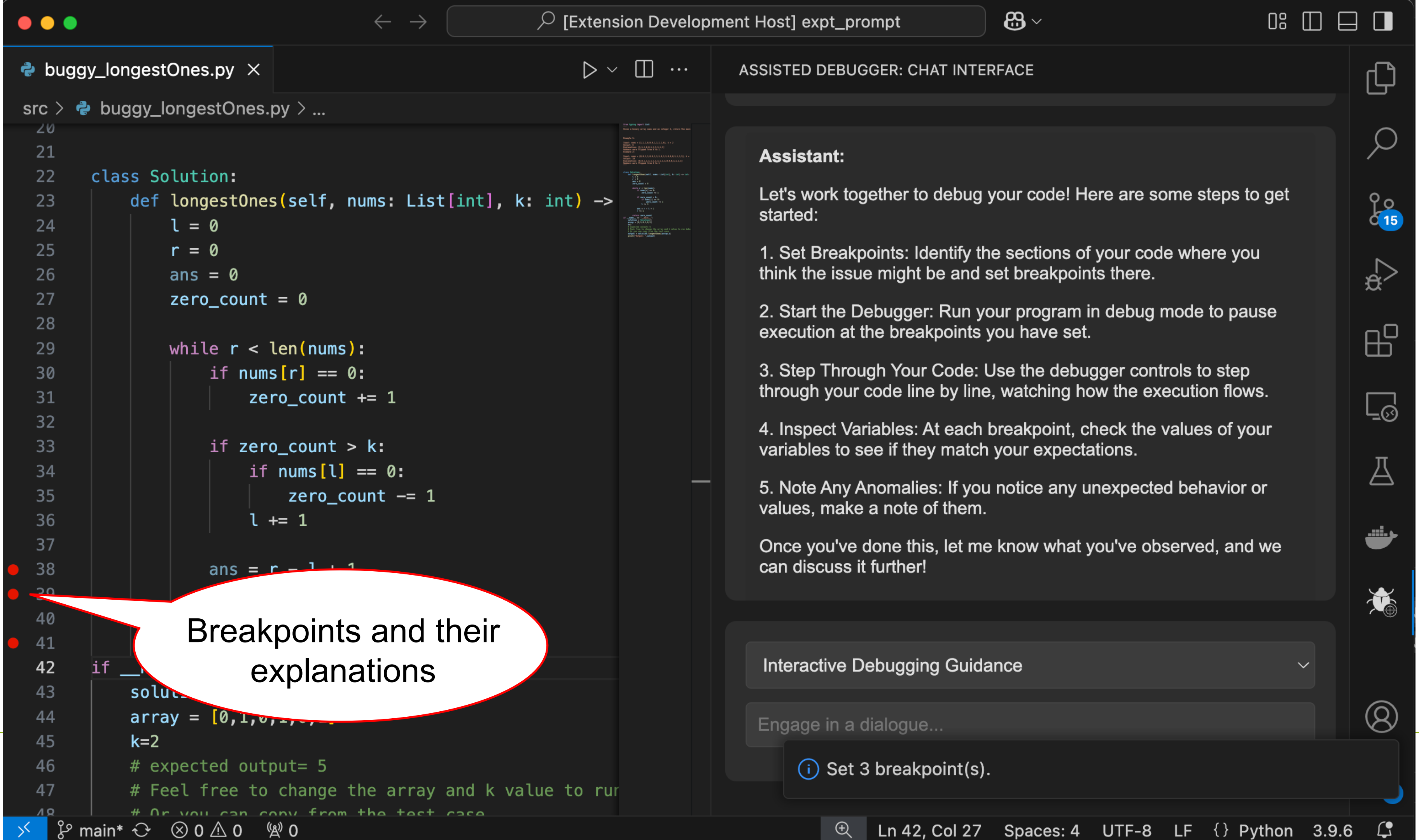
4. Inspect Variables: At each breakpoint, check the values of your variables to see if they match your expectations.

5. Note Any Anomalies: If you notice any unexpected behavior or values, make a note of them.

Once you've done this, let me know what you've observed, and we can discuss it further!

Interactive Debugging Guidance

Engage in a dialogue...



```
class Solution:
    def longestOnes(self, nums: List[int], k: int) ->
        l = 0
        r = 0
        ans = 0
        zero_count = 0

        while r < len(nums):
            if nums[r] == 0:
                zero_count += 1

            if zero_count > k:
                if nums[l] == 0:
                    zero_count -= 1
                l += 1

            ans = r - l + 1



if __
    solu
    array = [0,1,0,1,0,1]
    k=2
    # expected output= 5
    # Feel free to change the array and k value to rur
    # Or you can copy from the test case
```

Breakpoints and their explanations

Assistant:

Let's work together to debug your code! Here are some steps to get started:

1. Set Breakpoints: Identify the sections of your code where you think the issue might be and set breakpoints there.

2. Start the Debugger: Run your program in debug mode to pause execution at the breakpoints you have set.

3. Step Through Your Code: Use the debugger controls to step through your code line by line, watching how the execution flows.

4. Inspect Variables: At each breakpoint, check the values of your variables to see if they match your expectations.

5. Note Any Anomalies: If you notice any unexpected behavior or values, make a note of them.

Once you've done this, let me know what you've observed, and we can discuss it further!

Interactive Debugging Guidance

Engage in a dialogue...

Set 3 breakpoint(s).

Prompting

- Prompting has been used for breakpoint explanations, failure test explanations, and interactive guidance.
- **Guardrails** have been incorporated as keywords to ensure the fairness of the system and prevent misuse; e.g., we filter out code snippets.
- For interactive debugging guidance, the prompt also considers **additional context** involving problem description, line of code where the bug might be, and test cases.

Breakpoint explanation:

Explain the following lines of code where breakpoints are set. Provide a specific, concise explanation for each line:

${code}

Tests explanation:

As a software engineer, I need you to describe the failure observation while explaining the failure, also by showing the failures. Do not provide the root cause of the failure.

${test-output}

Prompt for Interactive Guidance

You are an AI tutor specializing in programming and computer science. Your role is to assist students in debugging their code by guiding them through questions and hints **only based on the code provided**.

Guidelines:

1. Maintain a supportive and encouraging tone.
2. Never provide complete code solutions or direct answers.
3. Guide the student towards understanding concepts and problem-solving strategies.
4. Use the Socratic method by asking probing questions.
5. Provide hints and explanations of relevant concepts.
6. Encourage good coding practices.
7. Use markdown formatting, including for inline code.
8. Always focus on helping the student think critically and arrive at the solution on their own.

Part 1/3

~~~~
Topic of this chat: ${problemDescription || 'Not specified by student.'}

Additional context:


Student's Code:
```
${codeText}
```


Bug Description:
${bugDescription}

Relevant Test Cases:
${testCases}

Debugger Variable Values:
${variableValues}
~~~~

Part 2/3

~~~~
**Important:** Only provide assistance related to the code above. Do not engage in topics or assignments not directly related to this code.

Begin the conversation by engaging the student to help them identify and resolve issues in their code. Remember to encourage critical thinking and self-discovery.
~~~~

Part 3/3

Pilot User Study

- Design Science methodology [1,2]
- Goal:
  - investigate the **practicality** of concept, and
  - receive first insights on the **usability** of our implementation
- We informed participants that the chatbot's hints might not be entirely accurate

---

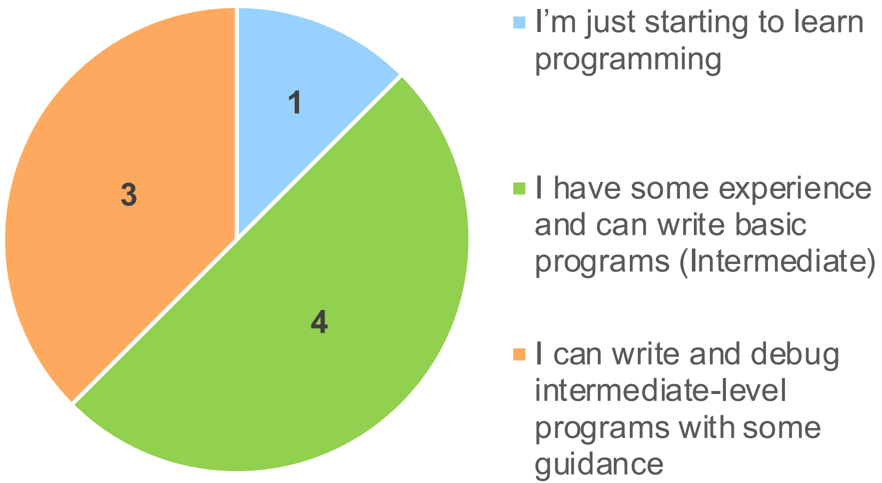
[1] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. 2004. Design Science in Information Systems Research. MIS Quarterly 28, 1 (2004), 75–105. http://www.jstor.org/stable/25148625

[2] Roel J. Wieringa. 2014. Design Science Methodology for Information Systems and Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-43839-8_1

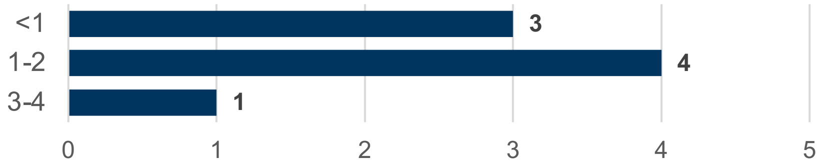
Participants

- **eight** first-year undergraduate CS students

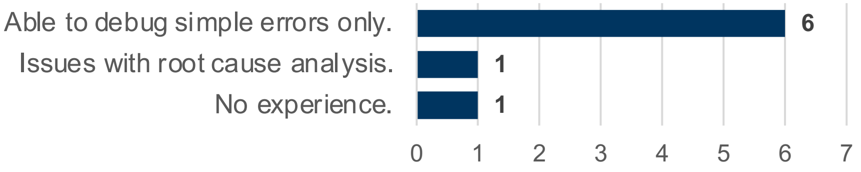
Programming Experience



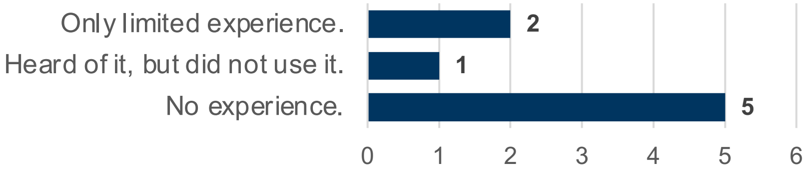
Programming Experience (in years)



Debugging Experience



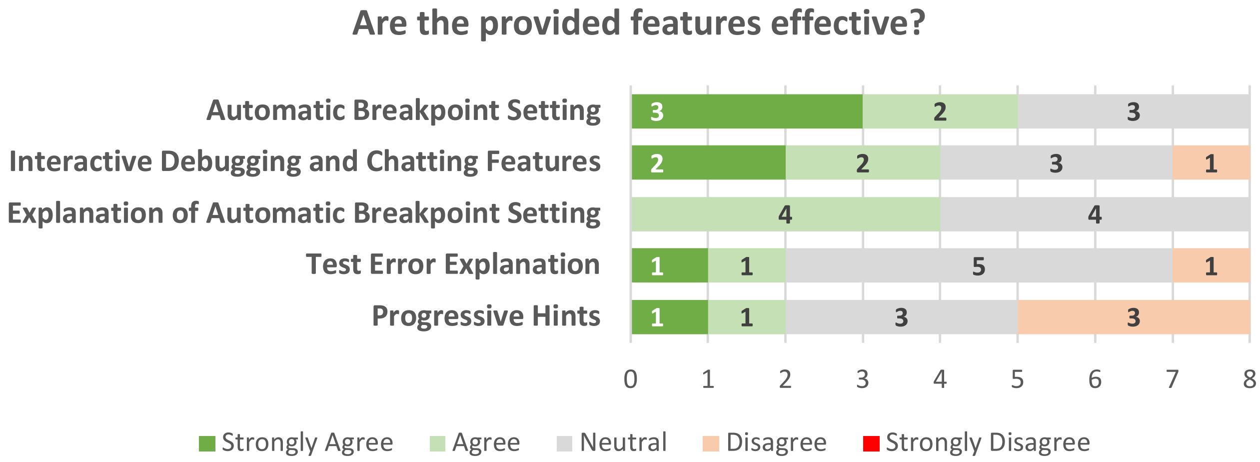
Experience with the Interactive Debugger



Experiment Setup

1. Briefing, IRB, and profiling survey
2. Introduction into debugging and tool demonstration
3. Task solving (~40min): two medium coding tasks with given buggy code
4. Usability and post-task survey

Results: Existing Features



"It did get the **breakpoint** in the right place, which was **a good start in debugging**."

"It helped me brainstorm while guiding me and **not letting me fully rely on AI**."

RUHR
UNIVERSITÄT
BOCHUM

Results: Usability

- based on standardized system usability survey [1]
- average System Usability Scale (SUS) is ~**65** (out of 100)
- providing a good basis for our future work, but also shows improvement potential

---

[1] John Brooke. 1996. SUS: A quick and dirty usability scale. Usability Evaluation in Industry (1996).

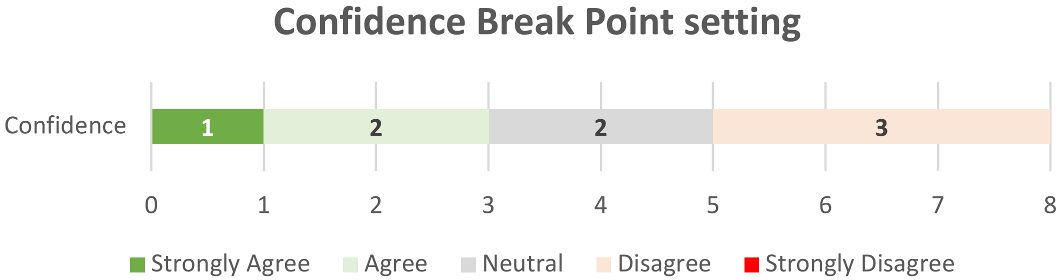
Results: Debugging Process

“Normally I would just **trial and error** until I got it right, but this tool allowed me to **systematically identify** the problems in the code step by step."

„This debugging process is **unfamiliar** to me so it is **quite hard**.“

„More **time consuming** but **effective**.“

"My usual debugging process includes using **print statements** and doing everything myself but sometimes I just get tired and ask ChatGPT. **Simulated Interactive Debugging** helped me think while guiding me through the **right process**.“



RUHR UNIVERSITÄT BOCHUM RUB

Results: Additional Features

- [6/8] further **hints** about **what** and **how** to change would be **helpful**
- [1/8] **user interface** could be improved to enable easier **switching** between the debugging information from the assisted debugger and the actual interactive debugging tool in the IDE
- [1/8] integrating some **automated test case generation** would be helpful

What did we learn?

- \+ confirmed hypothesis: most students do not use interactive debugging
- \+ concept of **Simulated Interactive Debugging** seems to be **useful** for students
- \+ students liked the **interaction** with the **chatbot**
- \+ students enjoyed the exposure to **AI-based tooling** in a controlled environment

- – more improvements regarding **implementation/UI** needed (~ usability)
- – the features „test error explanation“ and „progressive hints“ need to be implemented more robust

What‘s next?

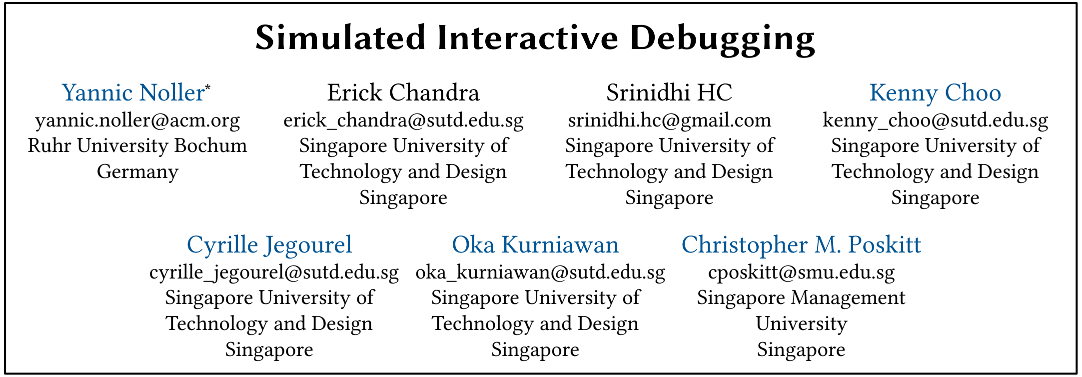
- this work is part of a larger research project to build an AI assistant for CS students integrated in the IDE
- full implementation of the concept, also the test assessment
- more intensive user evaluation
- improve our interactive chatbot with more guidance, e.g., by using a state machine-driven approach [1]
- support for other aspects
  - programming, debugging, test case generation, design, soft qualities: readability, etc.

---

[1] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An autonomous, llm-based agent for program repair. arXiv preprint arXiv:2403.17134 (2024).

Summary

- **Context**: CS/SE education
  ~ teaching interactive debugging
- **Concept**: Create a simulation of an interactive debugging.
  - guide students actively during the debugging process
  - set breakpoints automatically
  - incremental guidance via LLM-powered chatbot
- **Pilot Study:** with eight undergraduate students



Simulated Interactive Debugging

Yannic Noller*
yannic.noller@acm.org
Ruhr University Bochum
Germany

Erick Chandra
erick_chandra@sutd.edu.sg
Singapore University of Technology and Design
Singapore

Srinidhi HC
srinidhi.hc@gmail.com
Singapore University of Technology and Design
Singapore

Kenny Choo
kenny_choo@sutd.edu.sg
Singapore University of Technology and Design
Singapore

Cyrille Jegourel
cyrille_jegourel@sutd.edu.sg
Singapore University of Technology and Design
Singapore

Oka Kurniawan
oka_kurniawan@sutd.edu.sg
Singapore University of Technology and Design
Singapore

Christopher M. Poskitt
cposkitt@smu.edu.sg
Singapore Management University
Singapore