Concise Guide to Debugging Anything (1)

Example: A Python script that processes user data from a CSV file and stores results in a database.

1. Understand the System (Read the code, documentation or ask)

- Thoroughly read the system's documentation and consult with experts to understand its intended functionality.
- Carefully step through the code, understanding what each part is supposed to do. When in doubt, don't guess – refer back to the documentation or ask for clarification.

Read your script line by line, understand how Python's CSV module works, and know how the script interacts with the database.
If unsure about a function, you look it up in the Python documentation.

2. Replicate the Failure (Observe and record the bug occurring again)

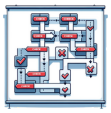
- Replicate the failure under the exact conditions it occurred, not just similar ones.
- For intermittent failures, vary the conditions until the issue can be can consistently replicated. Document every detail, including any seemingly irrelevant ones.

Your script fails to process certain rows in the CSV file. To replicate this, you ensure the testing environment mirrors production with the same CSV file, Python version, and database setup. You run the script and observe it failing on the same rows, confirming that you've replicated the issue.

3. Search and Identify the Bug (Don’t theorize without evidence)

- Engage in a thorough search to identify the exact cause of the bug. Rely on debug logging to make the bug and its cause visible.
- Avoid theorizing about potential causes without evidence - do a practical search.
- Remember the debug process might inadvertently modify the conditions and hide the failure.

You add print() statements or use Python's logging module to log the data processed at each step. You notice that the script fails when encountering special characters. By logging the exact input processed at the failure point, you identify that the script doesn't handle Unicode characters correctly.



4. Narrow the Search (Divide & Conquer or Successive Approximation)

- Apply the divide & conquer algorithm to narrow down the search area. Understand the range of the search and determine if the bug is upstream or downstream from the current point.
- Ideally, start at a known problematic point and work back up through the system, checking each branch until finding the source of the bug.

Your script is a multi-step process, and you're unsure where it's failing. You add checkpoints at the halfway point of each major section. If the script fails before reaching the midpoint of a particular section, you know the issue is upstream. This helps you isolate the problematic code block quickly.

5. Change One Thing at a Time (Control all other variables)

- When attempting to replicate the failure or identify the bug, change only one variable at a time, changing back any variables to the original condition before the next test.
- Keep a forensic mindset, analyzing what has changed since the last time the system worked correctly.

You suspect the failure might be due to the Python version or the CSV file format. First, you change only the Python version while keeping the same file to see if the issue persists. Then, you revert to the original version and try a different CSV file format. This controlled approach helps identify the exact cause.

Based on Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems by David Agans; Examples and guide by andypi.co.uk

Concise Guide to Debugging Anything (2)

6. Keep an Audit Trail (Write down details of the debugging process)

- Keep a detailed audit trail of all actions taken, the order in which they were done, and the results of each action.
- This record is invaluable in ensuring that all areas have been checked, in providing a clear account of your debugging process to others, and serving as a reference for future issues

You maintain a detailed log file using Python's logging module. Each action, such as opening a file, processing a row, or making a database entry, is logged with a timestamp. When the script fails, you have a comprehensive record of what happened immediately before the failure. You explain the bug fix in a git commit message.

7. Check Obvious Assumptions (that are fast to verify)

- Always verify the most basic assumptions first, such as whether the system is powered on, the service is running or expired data is cached.
- Before narrowing your search, confirm that the entire scope of the system is being checked, including all tools, dependencies and platforms you're working with.

Before diving deep into debugging, you check the basics: Is the CSV file present in the expected folder? Is the network connection to the database dropping? Is the database server running? You add checks in your script like os.path.isfile() to verify the file's existence before proceeding.

8. Ask for Help (from online resources and then experts)

- When you've hit a wall, a fresh perspective from others can shed new light on the problem.
- Start with async online resources like StackOverflow, ChatGPT before asking experts (e.g. Github issues) which introduces delay
- Always provide a bug description, logs, errors and what has been checked so far, etc.

If you fail to solve the issue independently, you write a detailed question on ChatGPT. You include the Python version, a snippet of your code, the exact error message, and what you've tried so far. ChatGPT lists a number of possible reasons for the failure, including one you haven't thought of, which gives you a new point to test.

9. Confirm the Fix Works (By testing with it applied and removed)

- Rigorously test the fix to confirm it addresses the issue. Then, remove the fix and retest to ensure that the issue reoccurs, confirming that your fix is directly resolving the problem.
- For particularly elusive bugs, add logging to capture details of the failure so it can be traced if it occurs again in production.

After adjusting your script to handle Unicode characters, you re-run it with the same problematic CSV file. It processes all rows successfully. Then, you remove the fix and confirm the script fails again, validating that your fix directly addresses the issue.

10. Fix the Underlying Process (Find the design or systemic problem)

- Reflect on the debugging process and identify any systemic improvements that could prevent similar issues in the future.
- For example, standardizing error and debug logs, enhancing test coverage, automating testing and deployment to minimize human error, and thoroughly understanding dependencies during the design phase.

To prevent similar issues, you decide to add more logging items and robust error handling to your script. You also incorporate a unit test that runs automatically on every git commit, testing a range of CSV formats and special cases to ensure the script is resilient to common data issues.

Based on Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems by David Agans; Examples and guide by andypi.co.uk