

# The 13 (and counting) Truths of Debugging

This set of thoughts has been adapted from a [document](#) based on the writing of [Nick Parlante](#) at Stanford. I've added some ideas as well, particularly thinking about modern software tools.

The key point of debugging code is that it depends on an objective and reasoned approach to the problem. It requires that you develop an overall perspective of what your code is doing, and an understanding of the internal workings of your code. Some useful ideas are as follows:

1. Debugging code is more mentally demanding than writing code. As a result, you should always write the simplest code possible to accomplish the task at hand. It's more difficult to debug code than to write that code in the first place – as a result, you should not write the cleverest code you can, because by definition you're not clever enough to debug it!
2. Code that is modular, that has well-chosen variable and function names, and that has documentation that explains the code's intent as well as what it does is generally the easiest to debug. In particular, modular code that is broken up into many functions with straightforward purposes lends itself to easy testing.
3. Intuition and hunches are great – you just have to test them out. When a hunch and a fact collide, the fact wins. Always.
4. Don't look for complex explanations. Even the simplest omission or typo can often lead to weird behavior (swapping of plus and minus signs or factors of two is particularly hard to find in scientific code). Everyone is capable of producing extremely simple and obvious errors from time to time. Look at code critically – don't just sweep your eye over that series of simple statements assuming that they are too simple to be wrong. Using a debugger like [pdb](#) (or its equivalent for your programming language of choice) to step through your code if necessary. The humble print statement is incredibly useful as well.
5. The clue to the error in your code is in the values of your variables and the flow of program control. Try to see what the facts are pointing to. The computer is not trying to mislead you. Work from the facts. Again, the debugger and print statement are your friend!
6. Be systematic and persistent. Don't panic. The bug is not moving around in your code, trying to trick or evade you. It is just sitting in one place, doing the wrong thing in the same way every time.
7. If your code was working a minute ago, but now it doesn't – what was the last thing you changed? This incredibly reliable rule of thumb is the reason you should test your code as you go rather than all at once, and you should ensure that each function behaves as expected (i.e., you get the correct outputs for a given set of inputs). In addition, the [git bisect](#) tool (or its [equivalent](#) in [Mercurial](#)) is incredibly useful, assuming you commit your code changes to your repository frequently!
8. Do not change your code haphazardly trying to track down a bug. You're performing experiments to determine where your bug is – would you change more than one variable in a physics experiment at a time? It makes the observed behavior much more difficult to interpret, and you tend to introduce new bugs. As suggested above, frequent commits to your version control repository help to trace back the changes you have made and determine when errors were introduced.
9. If you find some incorrect code that does not seem to be related to the bug you were tracking, fix that code anyway. Many times the incorrect code was related to, or has obscured, the bug in a way that is not obvious at the outset.
10. You should be able to explain the series of facts, tests, and deductions that led you to find a bug. Alternately, if you have a bug but can't pinpoint it, then you should be able to give an argument to a critical third party detailing why each one of your functions cannot contain the bug. One of these arguments will contain a flaw since one of your functions does in fact contain a bug. Trying to construct the arguments may help you to see the flaw. (See "[Rubber Duck Debugging](#)" for an extremely useful method of debugging.)

11. Be critical of your beliefs about your code. It's almost impossible to see a bug in a function when your instinct is that the function is innocent. Only when the facts have proven without question that the function is not the source of the problem should you assume it to be correct. [Unit testing](#) can be very valuable for this when your codebase is large!
12. Although you need to be systematic, there is still an enormous amount of room for guesses and hunches. Use your intuition about where the bug probably is located to direct the order that you check things in your systematic search. Check the functions you suspect the most first. Good instincts will come with experience.
13. Debugging is mentally demanding, and thus you can experience rapidly diminishing returns on your efforts when you're tired. Realize when you have lost the perspective on your code to debug. Take a break. Get some sleep, and look at your code in the morning. The “go do something else for a while, come back, and find the bug immediately” scenario happens far too often to be an accident.