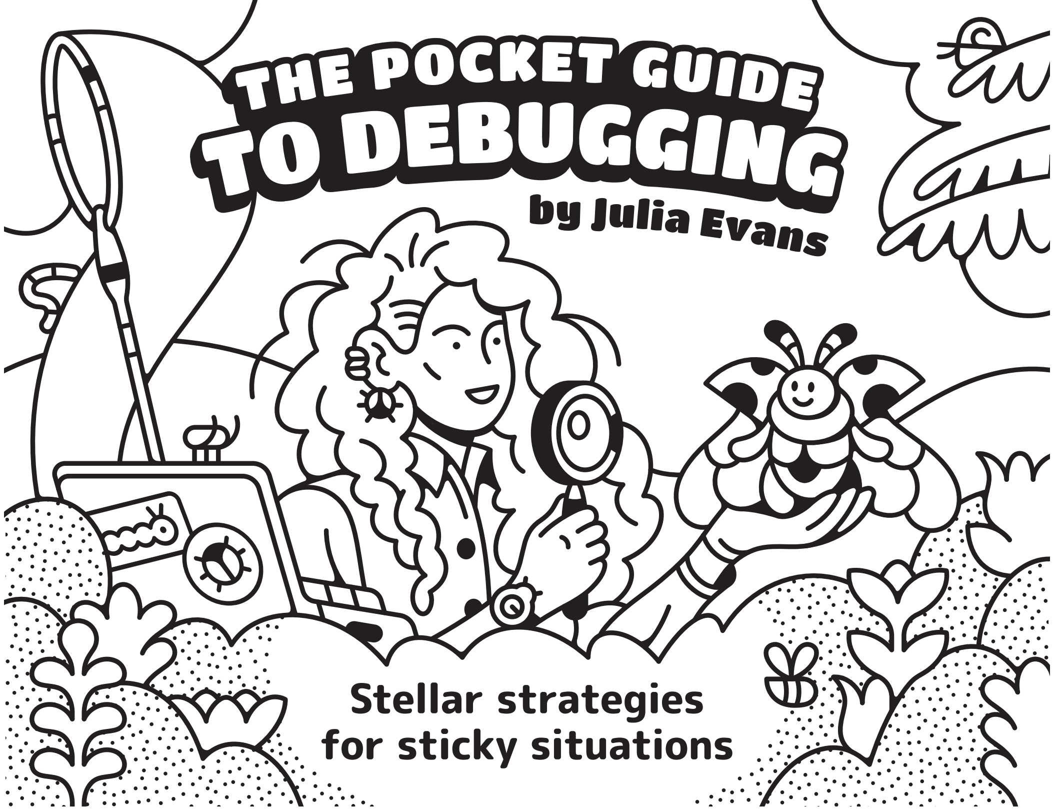
love this?
more at
☆ wizardzines.com ☆



THE POCKET GUIDE TO DEBUGGING

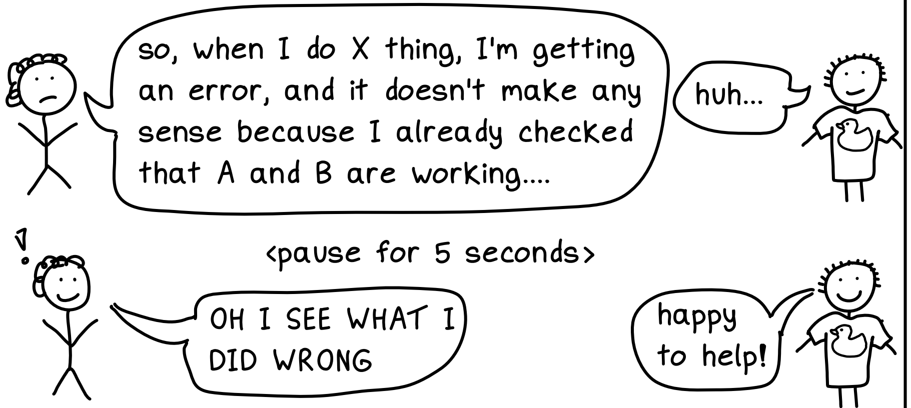
by Julia Evans

Stellar strategies for sticky situations

explain the bug out loud

GET UNSTUCK

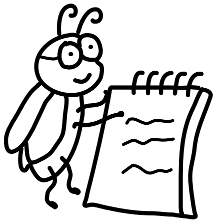
Explaining what's going wrong out loud is magic.

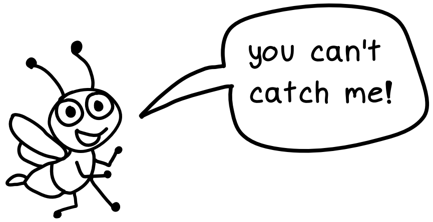

so, when I do X thing, I'm getting an error, and it doesn't make any sense because I already checked that A and B are working....
huh...
<pause for 5 seconds>
!
OH I SEE WHAT I DID WRONG
happy to help!

People call this "rubber ducking" because the other person might as well be a rubber duck (they don't say anything!)

chapter 2

GET ORGANIZED





thanks for reading

One more thing: I also built a choose-your-own-adventure debugging game to go with this zine, where you can solve computer networking mysteries:

https://mysteries.wizardzines.com

credits

Cover art: Vladimir Kašiković
Copy editing: Gersande La Flèche
Editing: Dolly Lanuza, Kamal Marhubi
Pairing: Marie Claire LeBlanc Flanagan
and thanks to all the beta readers ☺

brainstorm some suspects

GET ORGANIZED

Brainstorming every possible cause I can think of helps me not get stuck on the 1 or 2 most obvious possibilities.

could I be using the wrong version of this library?

am I passing the wrong argument to function X?

is something wrong with the server?

is the entire internet broken???

sometimes I find it easier to think clearly when writing by hand on paper

no filter! even ridiculous ideas!

write a message asking for help

GET UNSTUCK

When I'm REALLY stuck, I'll write an email to a friend:

→ "Here's what I'm trying to do..."

→ "I did X and I expected Y to happen, but instead..."

→ "Could this be because....?"

→ "This seems impossible because..."

→ "I've tried A, B, and C to fix it, but...."

This helps me organize my thoughts, and often by the time I finish writing, I've magically fixed the problem on my own!

It has to be a specific person, so that the imaginary version of them in my mind will say useful things :)

document your quest

AFTER IT'S FIXED

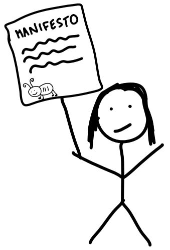
For very tricky bugs, writing up an explanation of what went wrong and how you figured it out is an amazing way to share knowledge and make sure you really understand it.

Ways I've done this in the past:

- complain about it in the internal chat! ← so people can search for it!
- write a quick explanation in the commit message
- write a fun blog post telling my tale of woe!
- for really important work bugs, write a 5-page document with graphs explaining all the weird stuff I learned along the way

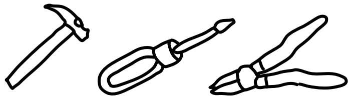
about this zine

This zine has:



① a manifesto with my general debugging principles

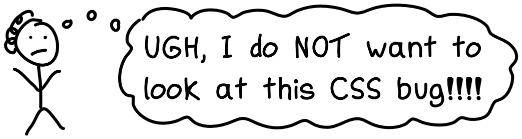
② a list of my favourite debugging strategies, which you can try in any order that makes sense to you



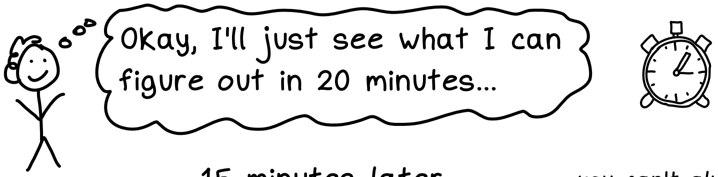
timebox your investigation

GET UNSTUCK

Sometimes I need to trick myself into getting started:



Giving myself a time limit really helps:

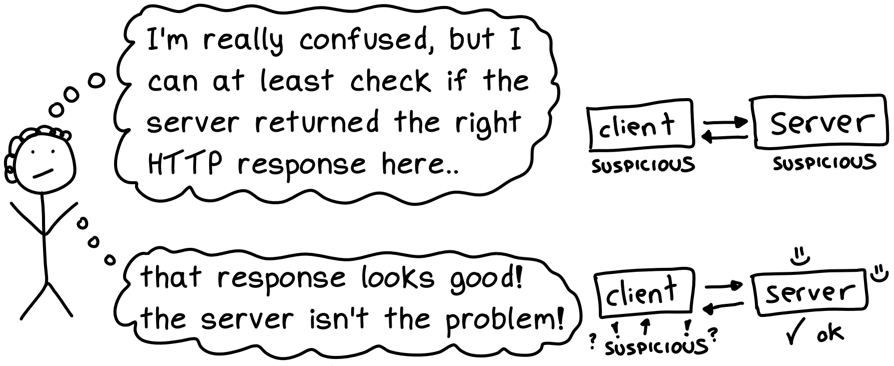


... 15 minutes later ... ← you can't always solve it in 15 minutes, but this works surprisingly often!

rule things out

GET ORGANIZED

Once I have a list of suspects, I can think about how to eliminate them.


I'm really confused, but I can at least check if the server returned the right HTTP response here..
client
SUSPICIOUS
Server
SUSPICIOUS
that response looks good! the server isn't the problem!
client
SUSPICIOUS
server
✓ ok

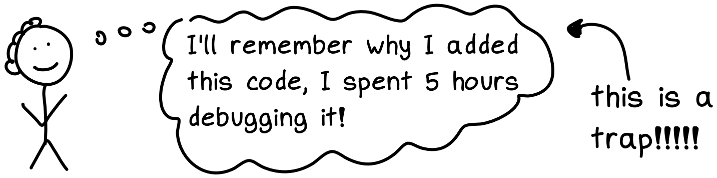
here we're assuming that was the only request being made. Otherwise this wouldn't be a safe conclusion :)

table of contents

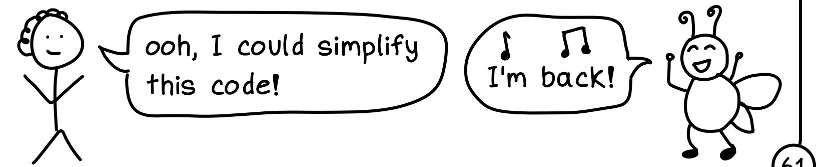
add a comment

AFTER IT'S FIXED

Some bug fixes are a little counterintuitive. Otherwise you would have written the code that way in the first place! You might think:


I'll remember why I added this code, I spent 5 hours debugging it!
this is a trap!!!!!

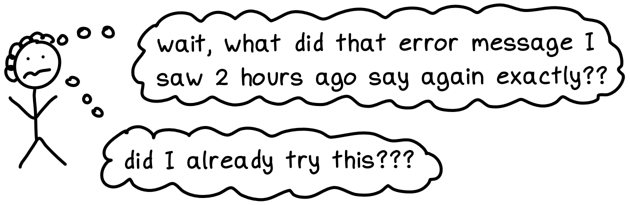
Adding a comment can help future you (or your coworkers!) avoid accidentally reviving a bug later.



keep a log book

GET ORGANIZED ☑

I don't usually write things down. But 2 hours into debugging, I get really confused:


wait, what did that error message I saw 2 hours ago say again exactly??
did I already try this???

Keeping a document with notes makes it WAY easier to stay on track. It might contain:

→ specific inputs I tried
→ error messages I saw
→ stack overflow URLs

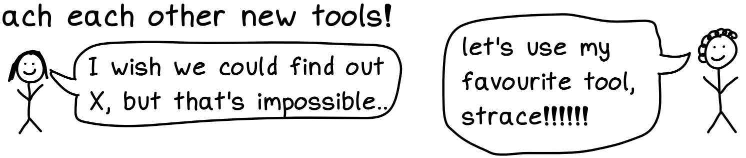
The log makes it easier to ask for help later if needed!

investigate the bug together

GET UNSTUCK

I find investigating a bug with someone else SO MUCH more fun than doing it alone. Debugging together lets you:

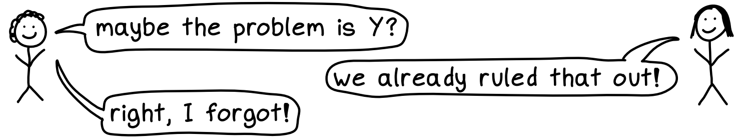
→ teach each other new tools!


I wish we could find out X, but that's impossible..
let's use my favourite tool, strace!!!!!!

→ learn new concepts!

what is this CORS thing?!?!
oh, I can explain that!

→ keep each other on track



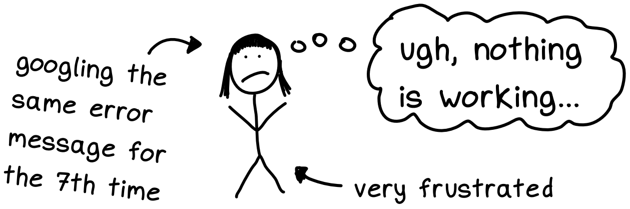
find related bugs

When you're done fixing a bug, glance around to see if there are any obvious places in your code that have the same bug.

I was calling function X wrong, I'll check if we're calling that function wrong anywhere else!
wow, my assumption about how Y worked was TOTALLY wrong, I should go back and fix some things...

take a break

Investigating a tricky bug requires a LOT of focus.


googling the same error message for the 7th time
ugh, nothing is working...
very frustrated

Instead, try one of these magical debugging techniques:
(even a 5 minute break can really help!)

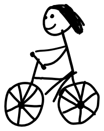
get a coffee!

go to bed!

ride your bike!

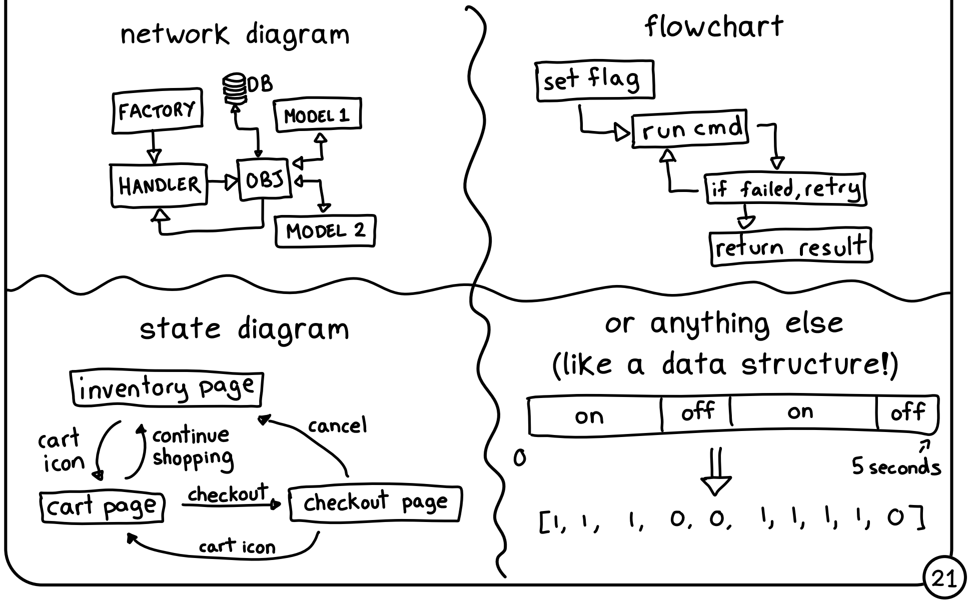
eat lunch!

have a shower!



draw a diagram

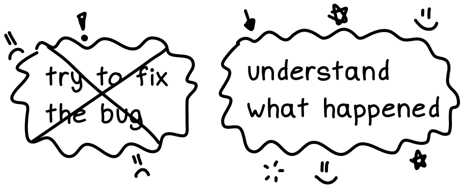
Some ideas:



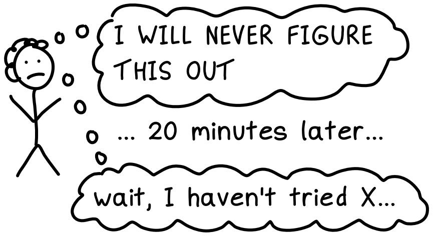
a debugging manifesto

DEBUGGING MANIFESTO

1 inspect, don't squash



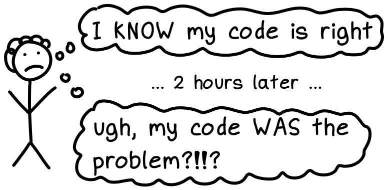
2 being stuck is temporary



3 trust nobody and nothing

this library can't be buggy... or CAN IT???
slowly growing horror

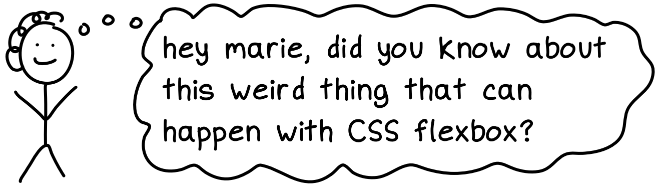
4 it's probably your code



tell a friend what you learned

AFTER IT'S FIXED

I love to celebrate squashing a bug by telling a friend:


hey marie, did you know about this weird thing that can happen with CSS flexbox?

Some possible outcomes of this:

→ they've seen that bug too, and teach me something else!

→ they learn something new!

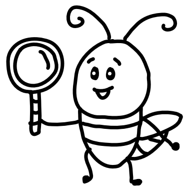
→ they ask questions I hadn't thought of

→ they tell me about a website/tool I didn't know about

→ it helps solidify my knowledge!

chapter 3

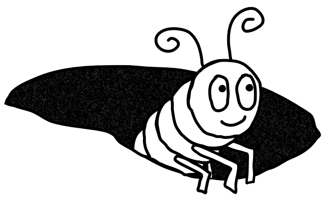
INVESTIGATE



chapter 6

GET UNSTUCK

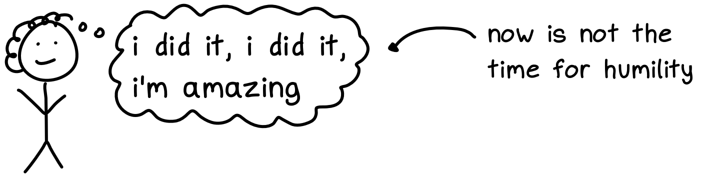
My favourite tricks to get from:
"I'm NEVER going to figure this out!"
to: "it seems obvious now!"



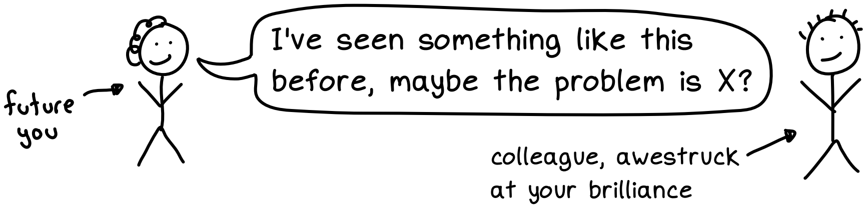
do a victory lap

AFTER IT'S FIXED

Once you've solved it, don't forget to celebrate! Take a break! Feel smart!


i did it, i did it, i'm amazing
now is not the time for humility

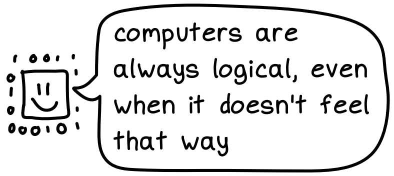
The best part of understanding a bug is that it makes it SO MUCH easier for you to solve similar future bugs.


I've seen something like this before, maybe the problem is X?
future you
colleague, awestruck at your brilliance

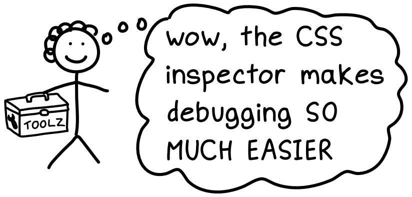
DEBUGGING MANIFESTO

5 don't go it alone

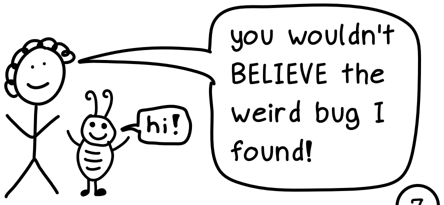
6 there's always a reason


computers are always logical, even when it doesn't feel that way

7 build your toolkit



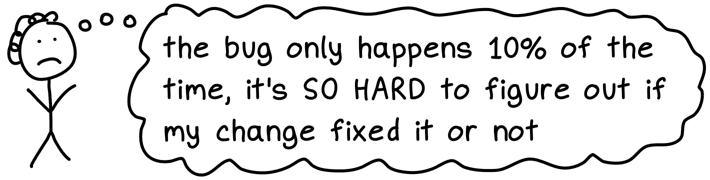
8 it can be an adventure



reduce randomness

SIMPLIFY

It's much easier to debug when your program does the exact same thing every time you run it.


the bug only happens 10% of the time, it's SO HARD to figure out if my change fixed it or not

There are a bunch of tools for controlling your program's inputs to reduce randomness, for example:

- → many random number generators let you set the seed so you get the same results every time
- → `faketime` fakes the current time
- → libraries like Ruby's `vcr` can record HTTP requests
- → record/replay debuggers like `rr` record everything

add lots of print statements

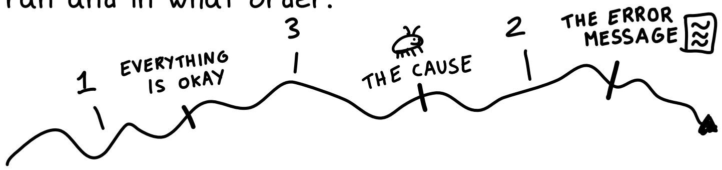
INVESTIGATE

I love to add print statements that print out 1, 2, 3, 4, 5...

```
console.log(1)
console.log(2)
console.log(3)
```

using descriptive strings is smarter, but I usually use numbers or "wtf???"

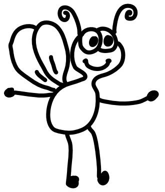
This helps me construct a timeline of which parts of my code ran and in what order:



Often I'll discover something surprising, like "wait, 3, never got printed??? Why not???".

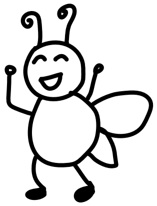
chapter 1

FIRST STEPS



chapter 8

AFTER IT'S FIXED

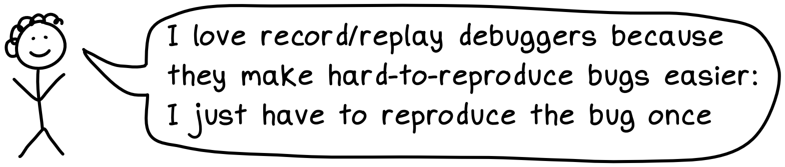


use a debugger

INVESTIGATE

A debugger is a tool for stepping through your code line by line and looking at variables. But not all debuggers are equal! Some languages' debuggers have more features than others. Your debugger might let you:

- jump into a REPL to poke around (see page 25)
- watch a location in memory and stop the program any time it's modified
- "record replay" debuggers let you record your entire program's execution and ✩ time travel ✩

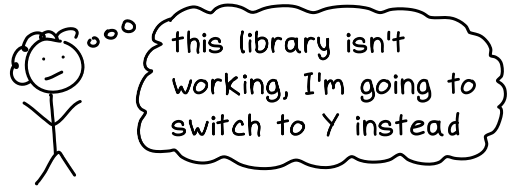

I love record/replay debuggers because they make hard-to-reproduce bugs easier: I just have to reproduce the bug once

delete the buggy code

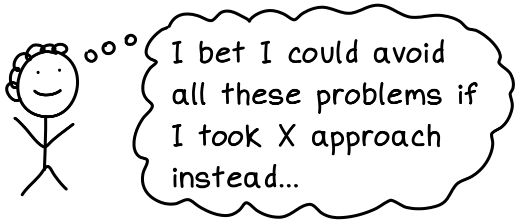
SIMPLIFY

Sometimes the buggy code is not worth salvaging and should be deleted entirely. Reasons you might do this:

- it uses a confusing library / tool


this library isn't working, I'm going to switch to Y instead

- you have a better idea for how to implement it

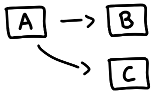

I bet I could avoid all these problems if I took X approach instead...

colours, graphs, and sounds

IMPROVE YOUR TOOLKIT

Instead of printing text, your program can tell you about its state by generating a picture! Or playing sounds at key moments!

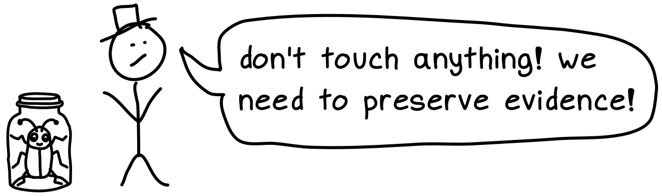
Some ways your programs can generate pictures or sounds:

- add colours to your log lines
- add red outlines around every HTML element!
- Haskell has an option to beep at the start of every major garbage collection
- draw a chart of events over time
- use graphviz to generate a diagram of your program's internal state 

preserve the crime scene

FIRST STEPS

One of the easiest ways to start is to save a copy of the buggy code and its inputs/outputs:



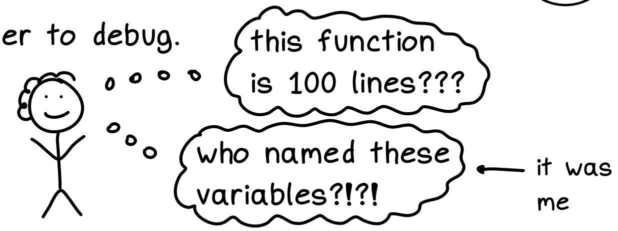
Depending on the situation, you might want to:

- make a git commit of the buggy code! (on a branch, just for you)
- save the input that triggered the bug
- save logs/screenshots to analyze later

tidy up your code

SIMPLIFY

Messy code is harder to debug.



Doing a tiny bit of refactoring can make things easier, like:

- rename variables or functions
- format it with a code formatter (go fmt, black, etc.)
- add comments
- delete old/untrue comments

Don't go overboard with the refactoring though: making too many changes can easily introduce new bugs.

jump into a REPL

INVESTIGATE

In dynamic languages (like Python / Ruby / JS), you can use a debugger to jump into an interactive console (aka "REPL") at any point in your code. Here's how to do it in Python 3:

1. edit your code
```
my_var = call_some_function()
breakpoint()
```
← add this!
2. refresh the page
3. play around in the REPL! You can call any function you want / try out fixes!

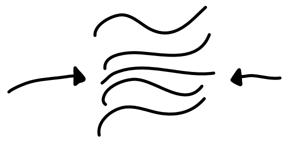
How to do it in other languages:

- Ruby: `binding.pry`
- Python (before 3.7): `import pdb; pdb.set_trace()`
- Javascript: `debugger;`

read the error message

FIRST STEPS

Error messages are a goldmine of information, but they can be very annoying to read:



giant 50 line stack trace full of impenetrable jargon, often seems totally unrelated to your bug

can even be misleading, like "permission denied" sometimes means "doesn't exist"

Tricks to extract information from giant error messages:

★ If there are many different error messages, start with the first one. Fixing it will often fix the rest.

★ If the end of a long error message isn't helpful, try looking at the beginning (scroll up!)

★ On the command line, pipe it to `less` so that you can scroll/search it (`./my_program 2>&1 | less`)

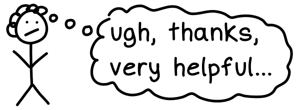
if you don't include 2>&1, less won't show you the error messages (just the output)

add pretty printing

IMPROVE YOUR TOOLKIT

Sometimes you print out an object, and it just prints the class name and reference ID, like this:

```
MyObject<#18238120323>
```



Implementing a custom string representation for an class you're often printing out can save a LOT of time.

The name of the method you need to implement is:

Python: `.__str__` Ruby: `.to_s` JavaScript: `.toString`
Java: `.toString` Go: `String()`

Also, pretty-printing libraries (like `pprint` in Python or `awesome_print` in Ruby) are great for printing out arrays/hashmaps.

find a version that works

INVESTIGATE

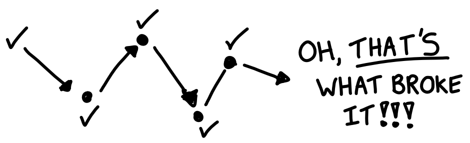
If I have a bug with how I'm using a library, I like to:

→ find a code example in the documentation

→ make sure it works

→ slowly change it to be more like my broken code

→ test if it's still working after every single tiny change



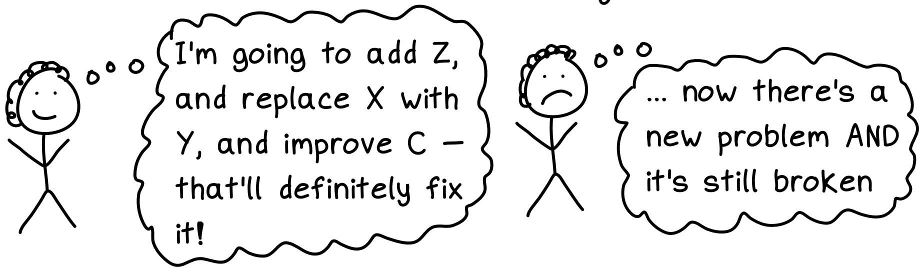
This puts me back on solid ground: with every change I make that DOESN'T cause the bug to come back, I know that change wasn't the problem.

one thing at a time

SIMPLIFY

It's tempting to try lots of fixes at once to save time:

dream: reality:


I'm going to add Z, and replace X with Y, and improve C — that'll definitely fix it!
... now there's a new problem AND it's still broken

If I found I've done this by accident, I'll:

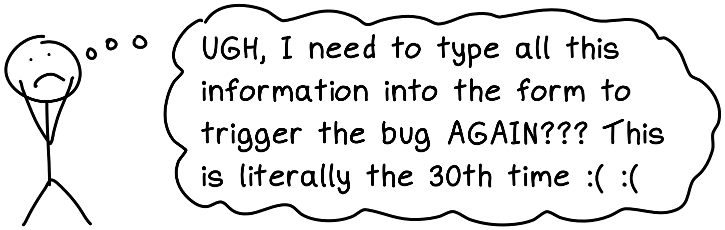
→ undo all my changes (`git stash`!)

→ make a list of things to investigate, one at a time

shorten your feedback loop

IMPROVE YOUR TOOLKIT

When you're investigating a bug, you'll need to run the buggy code a million times.


UGH, I need to type all this information into the form to trigger the bug AGAIN??? This is literally the 30th time :(:(

Ways to speed it up:

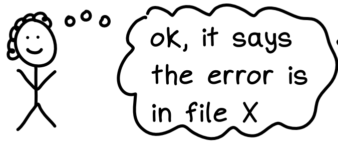
- ★ use a browser automation tool to fill in forms / click buttons for you!
- ★ write a unit test!
- ★ autorun your code every time you save!

reread the error message

FIRST STEPS

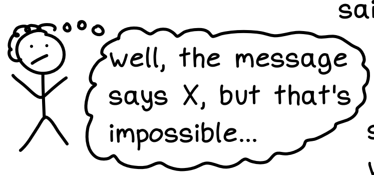
After I've read the error message, I sometimes run into one of these 3 problems:

① misreading the message



spoiler: it actually said file Y

② disregarding what the message is saying



spoiler: it was possible

③ not actually reading it

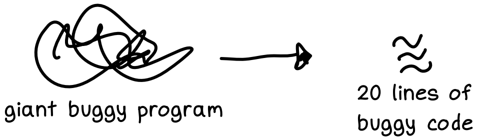
spoiler: she did not read it

write a tiny program

SIMPLIFY

Does your bug involve a library you don't understand?

I like to convert my code using that library into a tiny standalone program which has the same bug:



I find this makes it WAY EASIER to experiment and ask for help. And if it turns out that library actually has a bug, you can use your tiny program to report it.

look at recent changes

INVESTIGATE

Often when something is broken, it's because of a recent change. Usually I look at recent changes manually, but `git bisect` is an amazing tool for finding exactly which git commit caused the problem.

We don't have space for a full `git bisect` tutorial here, but here's how you start using it:

```
git bisect start
git bisect bad HEAD
git bisect good 1fe9dc
```

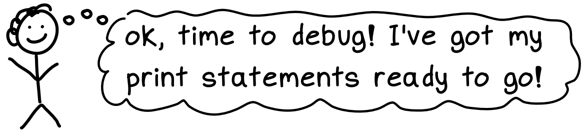
ID of a commit that doesn't have the bug

Then you can either tag buggy commits manually or run a script that does it automatically.

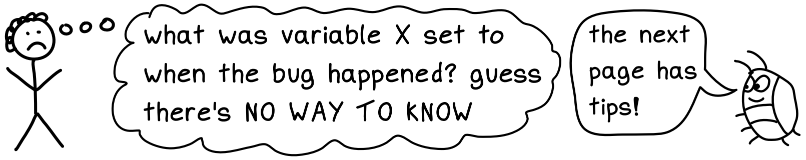
reproduce the bug

My favourite way to get information about buggy code is to run the buggy code and experiment on it. (Add print statements! Make a tiny change!)

If the bug is happening on your computer every time you run your program: hooray! You've reproduced the bug!


ok, time to debug! I've got my print statements ready to go!

But if you can't make the bug happen, you're left guessing.


what was variable X set to when the bug happened? guess there's NO WAY TO KNOW
the next page has tips!

types of debugging tools

Here are some tools I've found useful:

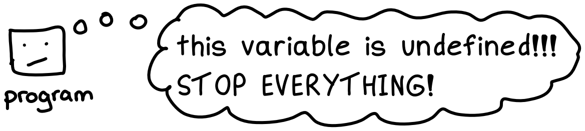
debuggers!	(most languages have one!)
profilers!	perf, pprof, py-spy
tracers!	strace, ltrace, ftrace, BPF tools
network spy tools!	tcpdump, wireshark, ngrep, mitmproxy
web automation tools!	selenium, playwright
load testers!	ab, wrk
test frameworks!	pytest, RSpec
linters/static analysis tools!	black, eslint, pyright
data formatting tools!	xxd, hexdump, jq, graphviz
dynamic analysis tools!	valgrind, asan, tsan, ubsan
fuzzers/property testing!	hypothesis, quickcheck, Go's fuzzer

I've never used these but lots of people say they're helpful

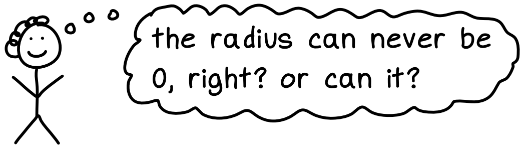
sprinkle assertions everywhere

Some languages have an `assert` keyword that you can use to crash the program if a condition fails. Assertions let you:

- come up with something that should ALWAYS be true
- immediately crash the program if it isn't

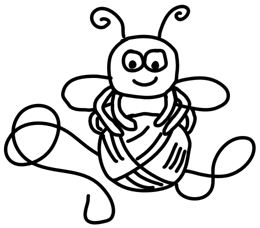


This is a great way to force yourself to think about what's ALWAYS true in your program, and check if you're right.



chapter 5

SIMPLIFY



try out a new tool

IMPROVE YOUR TOOLKIT

There are TONS of great debugging tools (listed on the next page!), but often they have a steep learning curve. Some tips to get started:

- → get someone more experienced to show you an example of how they'd use the tool ← this is SO helpful!!!
- → try it out when investigating a low stakes bug, so it's no big deal if it doesn't work out
- → take notes with examples of the options you used, so you can refer to them next time

inspect unreproducible bugs

FIRST STEPS

When you can't reproduce a bug locally, it's tempting to just try random fixes and pray. Resist the temptation! Some ways to get information:

- → try to reproduce the environment where it happened
- → ask for screenshots / screen recordings
- → add more logging, deploy your code, and repeat until you understand what caused the bug
- → read the code VERY VERY carefully ← incredibly boring but it actually does work sometimes
- → do your experimentation somewhere where you *can* reproduce the bug ← on a staging server? on someone else's computer?

find a new source of info

RESEARCH

We all know to look at the official documentation. Here are some less obvious places to look for answers:

- ★ the project's Discord, Slack, IRC channel, or mailing list
- ★ code search (search all of GitHub for how other people are using that library!)
- ★ GitHub issues (did someone else have the same problem?)
- ★ release notes (is the bug fixed in the new version?)
- ★ a book chapter (you might have a book on this topic!)
- ★ blog posts (sometimes there's an amazing explanation on the 2nd page of Google results)

comment out code

INVESTIGATE

Commenting out code is an amazing way to quickly do experiments and figure out which part of your code is to blame. You can:

- ★ comment out a function call and replace it with a hardcoded value, to check if the function call is broken
- ★ if the error message doesn't give you a line number, comment out huge chunks of the program until the problem goes away
- ★ comment out some code and rewrite it to see if the new version is better

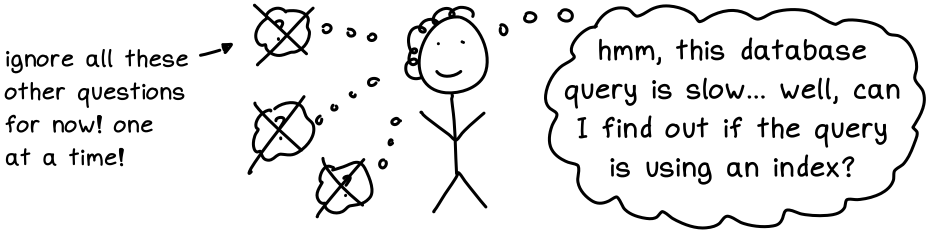
identify one small question

Debugging can feel huge and impossible. But all you have to do to make progress is:

① come up with ONE QUESTION about the bug

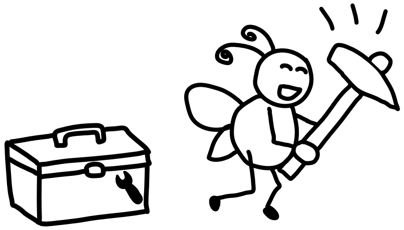
② make sure the question is small enough that you can investigate it in ~20 minutes

③ figure out the answer to that question


ignore all these other questions for now! one at a time!
hmm, this database query is slow... well, can I find out if the query is using an index?

chapter 7

IMPROVE YOUR TOOLKIT



analyze the logs

If you can't reproduce a bug, sometimes you need to comb through the logs for clues. Some tips:

→ filter out irrelevant lines (for example with `grep -v`)

→ find 1 failed request and search for that request's ID to get all the logs for that request

→ build a timeline: copy and paste log lines (and your interpretations!) into a document

→ if you see a suspicious log line, search to make sure it doesn't also happen during normal operation

→ if there's a cascade of errors, find the first error that started the problems

read the library's code

Lots of code isn't documented. But when there are no docs, there's always the source code! It sounds intimidating at first, but a quick search of the code sometimes gets me my answer really quickly.

Tips for exploring an unfamiliar library's code:

✩ search the tests! Tests are a GREAT source of examples

✩ `git clone` it locally to make it easier to navigate

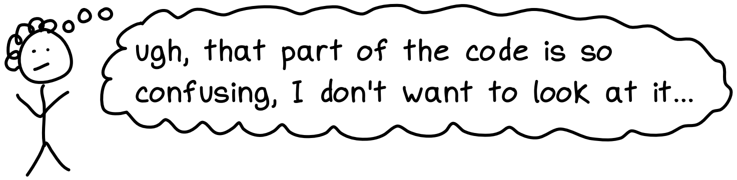
✩ search for your error message and trace back

✩ if it's a Python/JS/Ruby library, sometimes I'll edit the library's code on my computer to add print statements (just remember to take them out after!)

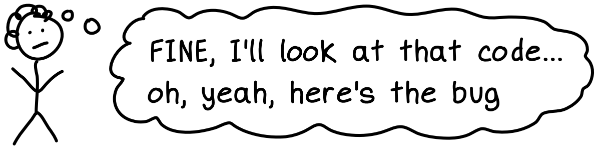
do the annoying thing

GET UNSTUCK

Sometimes when I'm debugging, there are things I'll refuse to try because they take too long.


ugh, that part of the code is so confusing, I don't want to look at it...

But as I become more and more desperate, eventually I'll give in and do the annoying thing. Often it helps!



retrace the code's steps

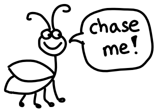
FIRST STEPS

Here's a classic (but still very effective!) way to get started:

① find the line of code where the error happened

② trace backwards to investigate what could have caused that error. keep asking "why?"

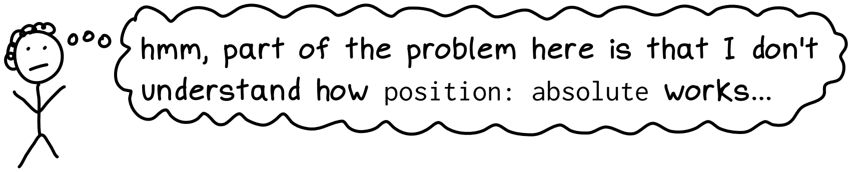
There's an error on line 58...
↳ that's because this variable has the wrong value...
↳ the value is set by calling this function...
↳ that function is making an HTTP request to the API...
↳ the API response doesn't have the format I expected! Why is that?



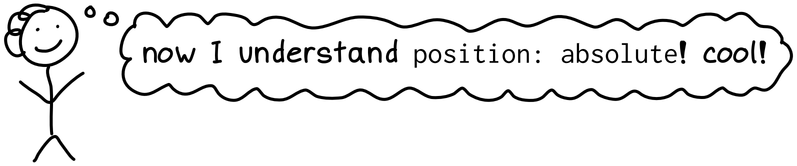
learn one small thing

RESEARCH

Bugs are a GREAT way to discover things on the edge of your knowledge.

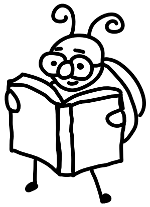

hmm, part of the problem here is that I don't understand how `position: absolute` works...

Finding one small thing I don't understand and learning it is really useful (and pretty fun!)



chapter 4

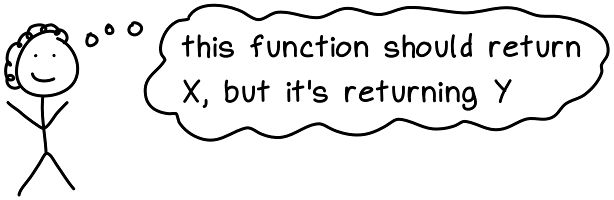
RESEARCH



write a failing test

FIRST STEPS

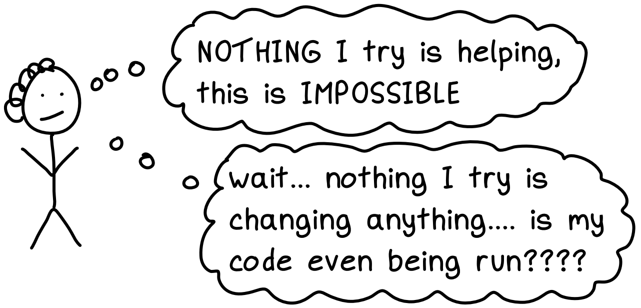
If your program already has tests, adding a failing test is a great way to work on your bug!



- it forces you to pinpoint what exactly the bug is
- it's easy to tell when you've fixed it (the test passes!)
- you can keep the test to make sure the bug doesn't come back

make sure your code is running

GET UNSTUCK


NOTHING I try is helping, this is IMPOSSIBLE
wait... nothing I try is changing anything.... is my code even being run????

If my changes have no effect at all, often it means I've made a silly mistake (like forgetting to restart the app) and my changes aren't being run!

I like to check that my code is being run by printing something out (like `print("asdf")`). Or, if that's not possible, I'll introduce an error so that it crashes.

read the docs

RESEARCH

There are many ways to read the docs!

the surgical strike:
Search for a specific function, find an example on the page, copy it and leave. ← this is often me :)

the question quest:
You have a specific question and you'll keep skimming different pages until you find the answer.

the IDE integration:
Set up your editor or IDE so that you can instantly jump to a function's documentation.

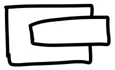
the rigorous read:
Get a cup of coffee and read all of the docs cover to cover, like a book.

find the type of bug

RESEARCH

If the bug is totally new to you, find out if there's a name people use for that type of bug!

this bug is happening intermittently, it's so weird.
oh, what's a race condition?
that sounds like it might be a race condition...

`terminated by signal SIGSEGV (address boundary error)`	→	segmentation fault
`flexbox: div doesn't fit in other div` 	→	item overflowing container (CSS)
`nodename nor servname provided, or not known`	→	DNS lookup failure
`RecursionError: maximum recursion depth exceeded`	→	stack overflow