

# Help Us Help You: Things to try before asking for help

## ▼ TABLE OF CONTENTS

- [Help Us Help You: Things to try before asking for help](#)
  - [Prevention](#)
    - [Develop with the browser dev tools open!](#)
    - [Understand what code does](#)
    - [Use correct formatting](#)
  - [Investigation](#)
    - [General advice: Reducing the issue](#)
    - [Problem 1: The page looks wrong](#)
    - [Problem 2: The code doesn't work in the way I expect](#)
    - [Problem 3: 500 error](#)
  - [Debugging tools](#)
    - [The Console](#)
    - [HTML validator](#)
    - [Dev tools Elements panel](#)
    - [Console API](#)
    - [debugger statement](#)
    - [Sources panel](#)
  - [Resolution](#)
  - [Conclusion](#)

Inevitably, you will run into cases where your code does not produce the expected result. These are called *bugs*. The process of investigating and fixing these bugs is called *debugging*. **The vast majority of the time spent programming is actually spent debugging**, especially when you are just starting out. But even seasoned programmers spend *far* more time debugging than writing code.

Debugging consists of three steps: 0. **Prevention:** The things you do to minimize bugs and/or make debugging easier.

- 1 **Investigation:** What is the issue? What is the code supposed to do, and what is it actually doing?
- 2 **Resolution:** Fixing the problem

## Prevention

Prevention consists of adopting good development habits that will either minimize the number of bugs you introduce in the first place, or stop them in their tracks.

### Develop with the browser dev tools open!

I really cannot stress this enough. If stuff is not working, the browser dev tools and especially [the console](#) is the *first* place you look for clues. **In fact, most seasoned web developers have dev tools continuously open while working.**

If you see errors in the console, **do not ignore them**. Even if they don't seem to be related to the issue you're investigating, they could be a symptom of a larger problem.

## Understand what code does

**Do not copy and paste code without understanding what it does.** If we don't expect you to understand it, we'll usually say so. If we give you an entire file or function to copy, it's typically because it doesn't matter to understand it (but it's still beneficial to try).

But in all other cases, you *should* understand what the code is doing and why. We give you code because it's easier to understand existing code and tweak it than to write it from scratch, not to save you the effort of understanding it.

If you don't understand what you're doing and why, you may still be able to guess where things go and move through the steps, but the mismatch between your mental model and what is actually happening will compound as you move through the lab making every step harder than the previous one.

If there are parts of syntax you don't understand, it can be hard to google for things. For example, suppose you get to [Step 3.3 of Lab 9](#) and don't understand this line:

```
label ??= Object.entries(wedges).find(([label, w]) => w === wedge[0]);
```

`Object.entries()` and `array.find()` is easy to google for, but how do you google for `??=` or the `([label, w])` syntax?! **That's what AI is excellent for!**

You could blanket ask it to explain this line of code:



You

Explain what this line of JS code does:

```
label ??= Object.entries(wedges).find(([label, w]) => w === wedge)[0];
```



ChatGPT

This line of JavaScript uses the logical nullish assignment operator (`??=`) along with the `Object.entries` method and the `find` method to achieve a specific behavior regarding a variable named `label`.

Let's break down what each part does:

1. `Object.entries(wedges)` : This method is used to convert the object `wedges` into an array of its own enumerable property `[key, value]` pairs. For example, if `wedges` is `{ apple: 10, banana: 20 }`, `Object.entries(wedges)` would produce `[['apple', 10], ['banana', 20]]`.
2. `find(([label, w]) => w === wedge)` : The `find` method is used to search through the array of entries. It takes a function as an argument, which is executed for each element in the array until the function returns `true`. In this case, the function checks if the second element of each entry (`w`) is equal to the variable `wedge`. The `find` method will return the first `[label, w]` pair that satisfies this condition.
3. `[0]` : This part of the expression accesses the first element of the array returned by `find`, which is the `label`. This is done assuming that the `find` method found a matching entry; otherwise, it returns `undefined`, and trying to access `[0]` of `undefined` would throw an error.
4. `label ??= ...` : The logical nullish assignment operator (`??=`) checks if `label` is `null` or `undefined`. If it is, then `label` is assigned the value on the right-hand side of the operator (i.e., the result of the `Object.entries(wedges).find(...)[0]`). If `label` is already something other than `null` or `undefined`, it retains its existing value.

In summary, this line of code is used to search the `wedges` object for a property whose value matches `wedge`. If found, the property name (`label`) is assigned to the variable `label`, but only if `label` was previously `null` or `undefined`. This effectively sets `label` to the name of the `wedge` entry in the `wedges` object if `label` was not already set.



**Figure 1:** Asking ChatGPT to explain a specific line of code (but other AI assistants are similar).

Or, you could ask about specific things you don't understand, which tends to give you the answer faster:



You

What does the ??= do in the line below?

```
label ??= Object.entries(wedges).find(([label, w]) => w === wedge)[0];
```



ChatGPT

The `??=` operator in JavaScript is known as the logical nullish assignment operator. It assigns the right-hand side expression to the variable on the left-hand side only if the left-hand side is currently `null` or `undefined`.

In the line of code you provided:

javascript

 Copy code

```
label ??= Object.entries(wedges).find(([label, w]) => w === wedge)[0];
```

Here's what happens step-by-step:

1. `Object.entries(wedges)` creates an array of `[key, value]` pairs from the `wedges` object.
2. `find(([label, w]) => w === wedge)` searches through these entries to find the first entry where the value `w` matches the `wedge`. The find function returns the entire entry `[label, w]` where this condition is true.
3. `[0]` extracts the first element of the found entry, which is the label corresponding to the wedge.
4. `label ??=` assigns the extracted label to the variable `label` only if `label` is currently `null` or `undefined`. If `label` already has a value other than `null` or `undefined`, it remains unchanged.

This operator is useful for providing default values or fallback assignments to variables that have not yet been set (i.e., are `null` or `undefined`).



**Figure 2:** Asking ChatGPT to explain a specific part of syntax (but other AI assistants are similar)..

[GitHub Copilot Chat](#) even has a menu option for this:

```

60 }
61
62 function transitionArc (wedge, label) {
63   [label ??= Object.entries(wedges).find(([label, w]) => w === wedge)[0]]}
64
65   let d = pieData.find(d => d.label === label);
66   let d_old = oldData.find(pd => pd.label === label);
67
68   if (sameArc(d_old, d)) {
69     return null;
70   }
71
72   let type = d ? d_old ? "update" : "in" : "out";
73
74   let from = d_old ? {...d_old} : getEmptyArc(label, oldData);
75   let to = d ? {...d} : getEmptyArc(label);
76
77   let angleInterpolator = d3.interpolate(from, to);
78   let interpolator = t => `path("${ arcGenerator(angleInterpolator(t))
79

```

This was the result:

GitHub Copilot used @workspace/explain

Used 1 reference >

This line of code is written in JavaScript and is used in a Svelte component. It uses the nullish coalescing assignment operator (`??=`) and the `Object.entries()` method along with the `Array.find()` method.

Let's break it down:

1. `Object.entries(wedges)`: This method is used to return an array of a given object's own enumerable string-keyed property [key, value] pairs. In this case, it's creating an array from the `wedges` object.
2. `.find(([label, w]) => w === wedge)`: The `find()` method is used on the array returned by `Object.entries()`. It returns the first element in the array that satisfies the provided testing function. Here, it's looking for an array where the second element (`w`) is equal to `wedge`.
3. `[0]`: This is used to access the first element of the array returned by `find()`. In this context, it's the `label` associated with the `wedge`.
4. `label ??= ...`: The nullish coalescing assignment (`??=`) is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand. In this case, it's assigning the found `label` to the `label` variable if `label` is not already defined.

So, in simpler terms, this line of code is looking for the label associated with a specific wedge in the `wedges` object and assigning it to the `label` variable if `label` is not already defined.

#### CAVEAT

Do note that **every time you use AI for anything, it's crucial to fact-check it**. If you ask it to explain syntax you don't understand, take the terms it gives you (in this case "logical nullish assignment operator") and google them to see if they are correct.

## Use correct formatting

It's very hard to understand what your code is doing if it's not formatted correctly. Especially `indentation` is very important as it shows the structure and hierarchy of your code.

- In HTML, if element contents span multiple lines, indent them (unless you have a reason not to).

- In JS, indent the contents of functions, loops, and conditionals.
- In CSS indent declarations (the property-value pairs inside the `{...}`) and nested rules.

Formatting is meant to help readability, not hinder it. There are cases where *not* indenting actually improves readability, e.g. in HTML, since everything is inside `<html>`.

If you find it hard to maintain good formatting manually, many developers use [Prettier](#) to automatically format their code when they save. It doesn't give you much flexibility, but is way better than poorly formatted code. VS Code also has a built-in formatter that you can enable in the settings, or call it on a case-by-case basis by opening the [Command Palette](#) (Ctrl+Shift+P) and typing `Format Document`.

#### CAVEAT

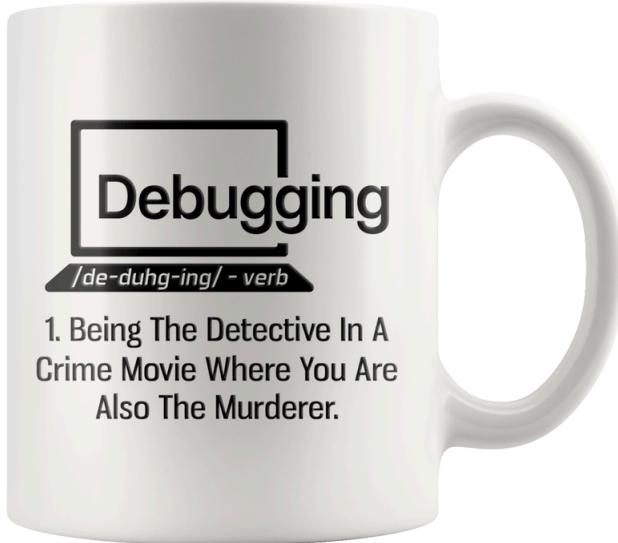
While a lot of code formatting is a matter of personal preference, and the main rule is that it's consistent, there is one exception: those of you who are familiar with other languages, may be used to code styles where the `{` is placed in a new line ([Allman style](#)):

```
function foo() {  
  // code  
}
```

**Don't do this in JS.** Because of [semicolon insertion](#), this can lead to [subtle bugs](#).

If you don't like the loss of control that automated formatters require, you should use a [linter](#) like [ESLint](#) or [stylelint](#) to ensure that the code style you prefer is followed consistently.

## Investigation



**Figure 3:** Quote by [Filipe Fortes](#)

**Investigation** can be broken down further:

- **Narrowing Down:** What part of the code is causing the problem?
- **Understanding:** *Why* is the code not working as expected?

Both typically involve making a series of hypotheses, and testing them to see if they are true. In this section, we will discuss strategies for both.

#### IMPORTANT

The very first step for most issues is to look at the [console](#) and the terminal where you ran `npm run dev`, both of which should already be open if you followed the [prevention](#) steps and check for errors.

## General advice: Reducing the issue

In the next few sections, we will discuss strategies for specific types of problems. This is a strategy that works *regardless* of the type of issue you are debugging.

Generally, **the less code you're dealing with, the easier debugging becomes**. In most cases, the issue is not in the entire codebase, but in a tiny part of it. The *Narrowing Down* phase is about figuring out what part of the code is causing the issue.

If you have a hypothesis about this, you can verify it by removing or commenting out that part of the code and seeing what happens — or removing *everything else* and making sure that the bug still reproduces. If not, there is always the “brute force approach”, where you simply start removing parts of the code until the issue goes away. This may involve:

- Commenting out entire sections of code
- Replacing API calls and computed values with hardcoded values
- Removing event listeners
- Replacing entire components
- Using the component with the issue in isolation on a new page

This is often called “Creating a reduced testcase” and is also a great process to follow when reporting a bug with one of the tools you’re using. Here are some excellent guides on this exact process:

- [Test Case Reduction - WebKit](#)
- [Reduced Test Cases - CSS Tricks](#)

Code playgrounds can be excellent for creating reduced testcases:

- [Svelte REPL](#) for Svelte
- [CodePen](#) for HTML, CSS, and JS

## Problem 1: The page looks wrong

The first step if things don’t quite look right is to verify the HTML structure is what we expect.

In some cases, especially if you are just getting started with HTML, and before you start using Svelte, the [HTML validation](#) can uncover a lot of issues.

However, **HTML that is not quite what you expect is not necessarily invalid**, in which case neither Svelte, nor a validator would catch the issue. In that case, the [Elements panel](#) is the place to be. It can help you *both* understand the structure of the page, and see the CSS that is *actually* applied to each element.

This will immediately resolve issues like:

- Declarations not being applied, either because the property or value is invalid, or because the property only applies to certain elements or when certain other properties have certain values
- CSS selectors not matching the elements you expect

## Problem 2: The code doesn't work in the way I expect

It's a common misconception that we develop a website by slowly getting it to converge to the final state. Think of it instead like a building site: there is scaffolding and tools everywhere while the building is still building built. It's ok for development to look messy while we are still building.

Assuming there are no errors, the first step is to *understand* what the current state is. What are the values of all intermediate variables? What is each function returning?

For HTML & CSS, we can use the [Elements panel](#) to see the CSS that is currently applied to each element and the current structure of the DOM tree.

### TIP

For transitions & animations, it helps to set their duration to a very high value (e.g. 3s) as you're developing them, so that it's very clear what is actually happening.

But for JS, the process is a bit more involved. We need to think like a computer: start from the beginning of the code and follow it through, keeping track of the values of all variables in your head. Then we want to make sure that the values are what we expect them to be at each step.

If the code we are trying to inspect is not called very frequently, we can use [console.log\(\) statements](#) with variables to see what their values are.

### CAVEAT

If the value we are logging is an object, it will be logged as a reference to the object, not its value. This means we will be seeing the *current* value of the object, not the value at the time of the `console.log()` statement. To see current values, we need to log a copy of the object, e.g. `console.log({...myObject})`, or specific values, e.g. `console.log(myObject.myProperty)`.

### TIP

VS Code allows us to just type `log` and then hit Tab to get a `console.log()` statement with the caret between the parentheses, which can be great for quickly logging things.

In some cases you may need to rewrite code a bit to make this possible. For example, assume you have an arrow function that just returns a value:

```
let interpolator = (t) => someFunction(t);
```

If you want to log the return value, you'd need to do something like this:

```
let interpolator = (t) => {
  let result = someFunction(t);
  console.log(result);
  return result;
};
```

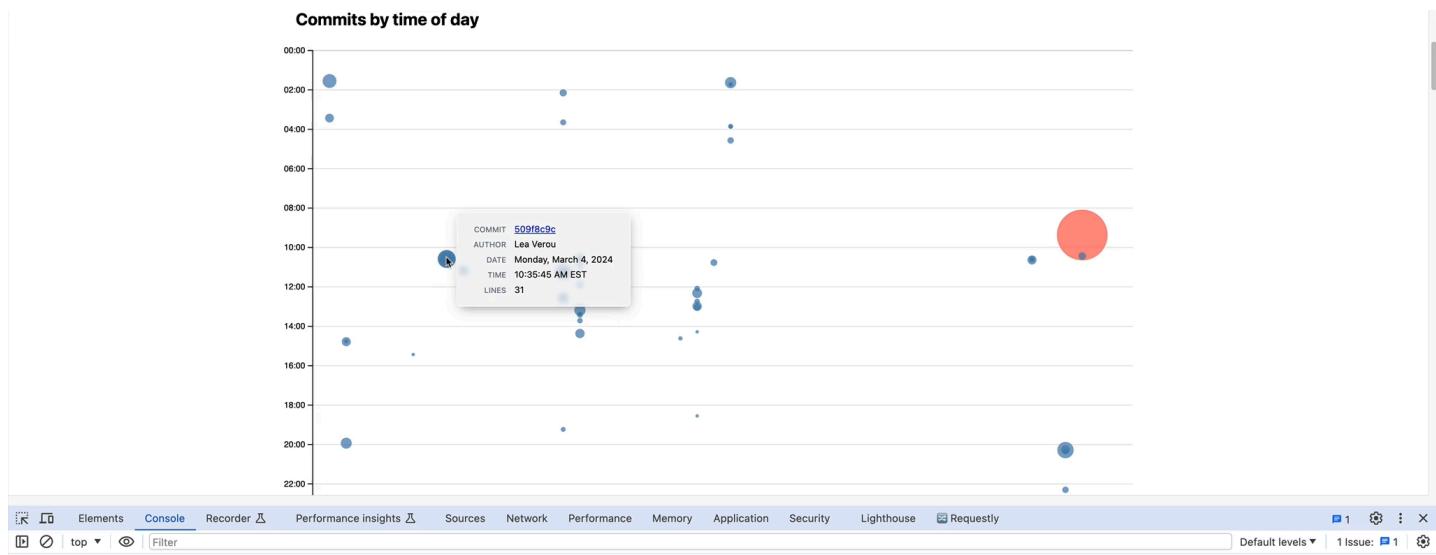
Or you can define a `log()` function that returns its first argument:

```
function log(...args) {  
  console.log(...args);  
  return args[0];  
}
```

And then you can just wrap values with it:

```
let interpolator = (t) => log(someFunction(t));
```

However, if the code is called very frequently, it would be hard to wade through hundreds of log messages. For example, assume we were working on [Step 3.4 of Lab 9](#) and trying to debug why our interpolator didn't work as we expected by logging the CSS we are applying. We'd get something like this:



Good luck wading through all these logs!

In that case, when using Svelte, we can simply print values in the HTML, by using something like `{ myVariable }` if it's a primitive, or something like `{ JSON.stringify(myVariable) }` if it's an object.

The downside is that we don't get historical values, but the upside is that we can see the value *right now* as it changes.

#### TIP

Don't forget to clean up after you've found & fixed the bug, by removing these expressions and/or log statements!

Lastly, if none of these helps zero in on the issue completely, it may be time for the big guns: the [debugger statement](#) and the [Sources panel](#). These allow us to execute our code, see the values of all local variables at each point, and even stop it at certain points and experiment.

### Problem 3: 500 error

If you're lucky, when you make a mistake with Svelte, you will get a nice error display like this:

The screenshot shows a browser window with a red border. Inside, there is a detailed error message from the Svelte compiler. The message starts with a stack trace:

```
[plugin:vite-plugin-svelte] /Users/leaverou/Documents/vis-society/lab-7/src/routes/meta/+page.svelte:130:5 Element with a slot='...' attribute must be a child of a component or a descendant of a custom element
```

Below the stack trace, the code is shown with line numbers 128 through 132. Line 130 contains the problematic line:

```
128|           <CommitScatterplot commits={filteredCommits} bind:selectedCommits={selected
129|
130|           <p slot="viz">{hasSelection ? selectedCommits.length : "No"} commits select
131|           ^
132|           <Pie colors={colors} data={
```

Following the code, there is a long stack trace:

```
CompileError: Element with a slot='...' attribute must be a child of a component or a descendant of a custom element
  at error (file:///Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/compiler/errors.js:536:
  at validate_slot_attribute (file:///Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/compi
  at validate_element (file:///Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/compiler/pha
  at RegularElement (file:///Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/compiler/phase
  at go (file:///Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/compiler/phases/visitors.j
  at visitor (file:///Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/compiler/phases/visit
  at go (file:///Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/compiler/phases/visitors.j
  at visitor (file:///Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/compiler/phases/visit
  at Object.next (file:///Users/leaverou/Documents/vis-society/lab-7/node_modules/zimmerframe/src/walk.js:107:
  at go (file:///Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/compiler/phases/visitors.j
```

At the bottom of the error message, there are instructions:

Click outside, press **Esc** key, or fix the code to dismiss.  
You can also disable this overlay by setting `server.hmr.overlay` to `false` in `vite.config.js`.

However, other times you will only get something like this:

# 500

Internal Error

In that case, the first step is to look at the terminal panel where you ran `npm run dev` to see *what* the actual error is.

If the issue is in the JS, it could be easy to read:

```
12:04:06 PM [vite] page reload src/routes/meta/+page.svelte (x2)
ReferenceError: onMount is not defined
    at _page (/Users/leaverou/Documents/vis-society/lab-7/src/routes/meta/+page.svelte:16:1)
    at /Users/leaverou/Documents/vis-society/lab-7/.svelte-kit/generated/root.svelte:45:41
    at Module.slot (/Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/internal/server/index.js:455:3)
    at _layout (/Users/leaverou/Documents/vis-society/lab-7/src/routes/+layout.svelte:23:24)
    at Root (/Users/leaverou/Documents/vis-society/lab-7/.svelte-kit/generated/root.svelte:44:40)
    at Module.render (/Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/internal/server/index.js:107:2)
    at Function._render [as render] (/Users/leaverou/Documents/vis-society/lab-7/node_modules/svelte/src/legacy/legacy-server.js:26:18)
    at Module.render_response (/Users/leaverou/Documents/vis-society/lab-7/node_modules/@sveltejs/kit/src/runtime/server/page/render.js:171:29)
    at async Module.render_page (/Users/leaverou/Documents/vis-society/lab-7/node_modules/@sveltejs/kit/src/runtime/server/page/index.js:286:10)
    at async resolve (/Users/leaverou/Documents/vis-society/lab-7/node_modules/@sveltejs/kit/src/runtime/server/respond.js:446:18)
```

However, if the issue is in your Svelte HTML, you may have to wade through your entire component code to get to the actual error:



```
'\t</div>\n' +
'\n' +
'<Scrolly>\n' +
'\n' +
'<style>\n' +
'\t:global(body) {\n' +
'\t\tmax-width: min(120ch, 80vw);\n' +
'\t}\n' +
'\n' +
'\tsvg {\n' +
'\t\toverflow: visible;\n' +
'\t}\n' +
'\n' +
'\t.time-filter {\n' +
'\t\t/* position: sticky; */\n' +
'\t\ttop: 1em;\n' +
'\t\tz-index: 1;\n' +
'\t\tbackground: white;\n' +
'\t\tdisplay: grid;\n' +
'\t\tgrid-template-columns: auto 1fr;\n' +
'\t\tgap: .5em;\n' +
'\t\tmargin-block: 1.5em;\n' +
'\n' +
'\t\ttime {\n' +
'\t\t\tgrid-column: 2;\n' +
'\t\t\ttext-align: right;\n' +
'\n' +
'\t\t}\n' +
'\t}
```

**Figure 4:** In this case the actual error is the red text at the top: "Element with a slot='...' attribute must be a child of a component or a descendant of a custom element"

## Debugging tools

Here we describe the tools you can use to investigate and resolve issues. There are links to these in the text above, but going through this section by itself may be useful for those with more experience, or as a quick reference after you have read the whole guide.

### The Console

The console is the second tab in your browser dev tools. It looks like this:

The screenshot shows a web browser window with the title 'Home | Vis & Society 2024'. The URL in the address bar is 'vis-society.github.io'. The page content includes a search bar, navigation links for 'Canvas', 'Forum', and 'Feedback Form', and a main heading 'Interactive Visualization & Society, Spring 2024'. Below the heading is a paragraph about the course's purpose and objectives, followed by a list of 7 learning outcomes.

**Interactive Visualization & Society, Spring 2024**

The world is awash with increasing amounts of data, and we must keep afloat with our relatively constant perceptual and cognitive abilities. Visualization provides one means of combating information overload, as a well-designed visual encoding can supplant cognitive calculations with simpler perceptual inferences and improve comprehension, memory, and decision making. Moreover, visual representations may help engage more diverse audiences in the process of analytic thinking.

By the end of this course, you should expect to be able to:

- 1 Understand key visualization techniques and theory.
- 2 Design, evaluate, and critique visualization designs.
- 3 Wrangle and explore datasets through visualization using Tableau.
- 4 Tell stories with data, and use visualization for policy change.
- 5 Implement interactive data visualizations using D3.js.
- 6 Develop a substantial visualization project.
- 7 Read and discuss visualization research papers (graduate students only).

Among other things, the console will show you:

- JS errors
- Failed HTTP requests (e.g. file not found)
- Some HTML errors (e.g. invalid value for certain attributes)

#### FYI

#### Why doesn't the browser just notify me when there are errors on the page?

The browser doesn't know you're the website developer, and most web users don't want to know about JS errors on websites they visit. In fact, early browsers did report them to users, but that went away as browser UIs became more user-friendly, and started targeting less technical folk. In fact, most websites you visit on a daily basis have a cornucopia of JS errors and you'd never know about them unless you open the browser console.

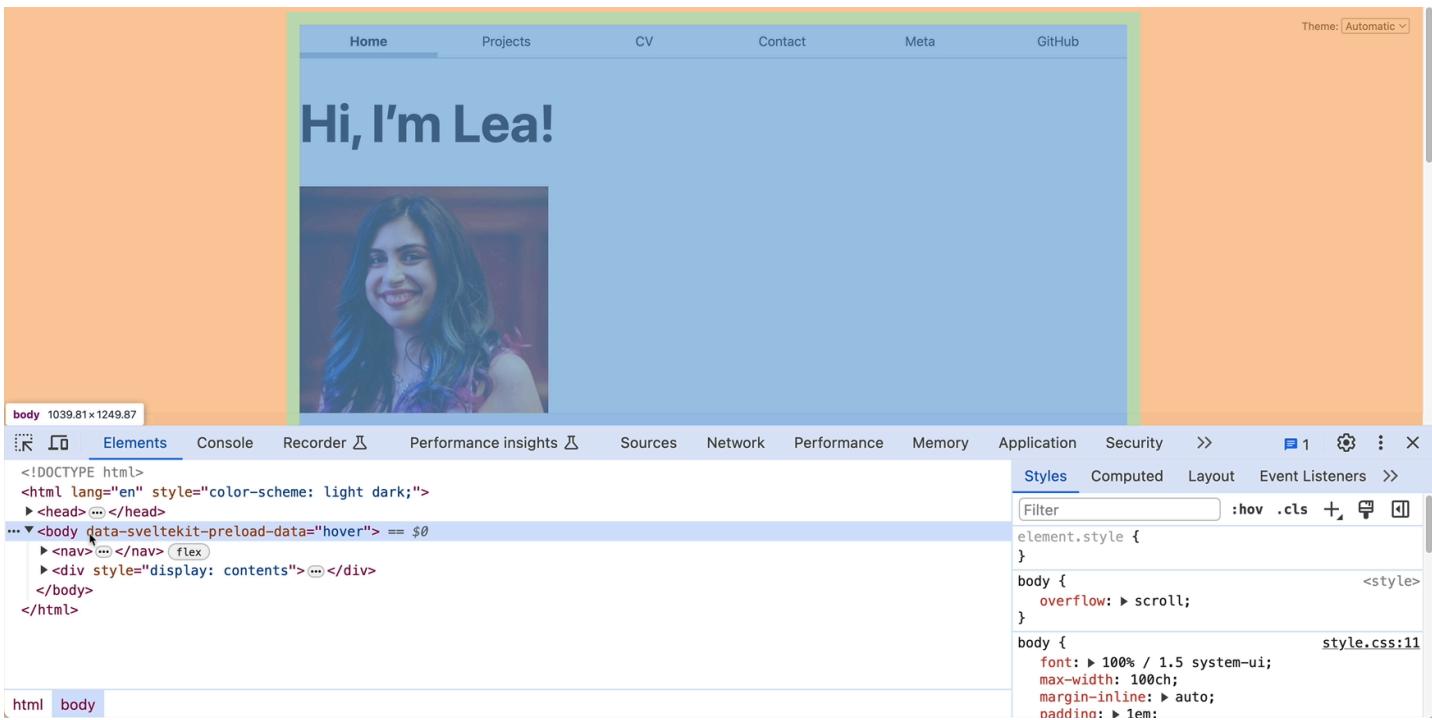
## HTML validator

The [W3C HTML validator](#) is a tool that checks your HTML for errors. You can provide your HTML to it by uploading it, giving it a URL, or perhaps most usefully when developing, [pasting it in directly](#).

Svelte markup will throw the validator off, so this becomes less useful as we start using Svelte. You can still validate Svelte's output, but depending on how much of your code is rendered by the build process, this can be more or less useful. The silver lining is that Svelte also catches some HTML issues itself and then they become [500 errors](#).

## Dev tools Elements panel

It looks like this:



It can help you catch issues in a number of ways. First, it helps you explore the [DOM structure](#) by expanding and collapsing element contents. This can be very helpful when your elements are not nested in the way you expect, which can be hard to see in source code alone.

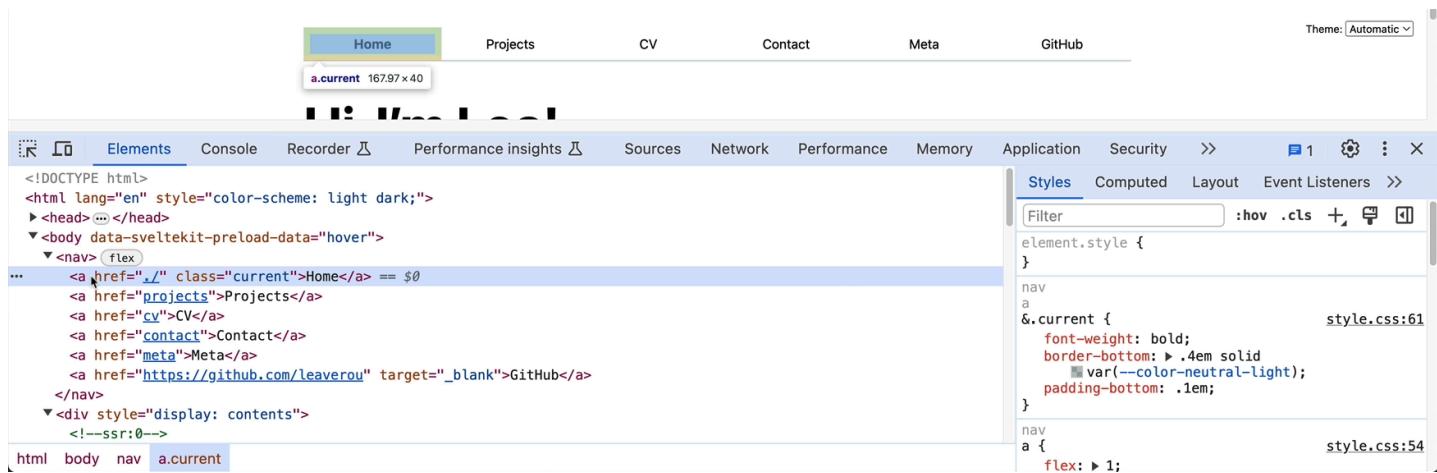
But the elements panel is *especially* useful to show you the CSS currently applied, which is done in several ways:

- As you hover over each element, it is highlighted and its [box model](#) is shown (orange for margin, purple for padding).
- If the element is a grid or flex container, its layout is shown as well.
- The right panel shows you all the CSS rules that currently apply to the element, whether they come from your CSS or the browser's, and which ones are overridden by others.

The CSS panel shows a bunch of things:

- The CSS rules that apply to the element, and where they come from (your CSS, browser defaults, etc.)
- The computed styles, which are the final styles that apply to the element after all the rules are applied
- The box model, which shows you the padding, border, and margin of the element
- Which CSS properties are being overridden by others?
- Which CSS properties are inherited from ancestors?
- Which CSS properties are invalid and thus ignored?
- Which CSS properties are being ignored because of the values of other properties? For example, Flexbox properties like `flex` and `flex-*` have no effect if `display` is not `flex` or `inline-flex`. Note that the dependencies between CSS properties are more complex than the dev tools can show, so this will not uncover all such issues.

If the issue you are investigating only happens during certain user actions (e.g. `:hover`), you can simulate those actions:



It also allows you to **edit the CSS in real time** and see the page update. This can be incredibly useful both in the [investigation](#) and [resolution](#) phases. Keep in mind that the browser cannot possibly update your source code, so if you refresh you will lose your changes — you need to remember to copy them over! However, you can see this as a feature, not a bug: you can try out changes in a non-destructive way.

In addition to editing properties and values, it allows you to quickly remove declarations to see what will happen (which is very useful in the [investigation](#) phase)

## Console API

The [Console API](#) is a set of functions you can use in your JS code to log messages to the console. The one you would be using most often is `console.log()` which logs a message to the console. However, `console.trace()` can also be useful if you can't understand why a certain function is being called.

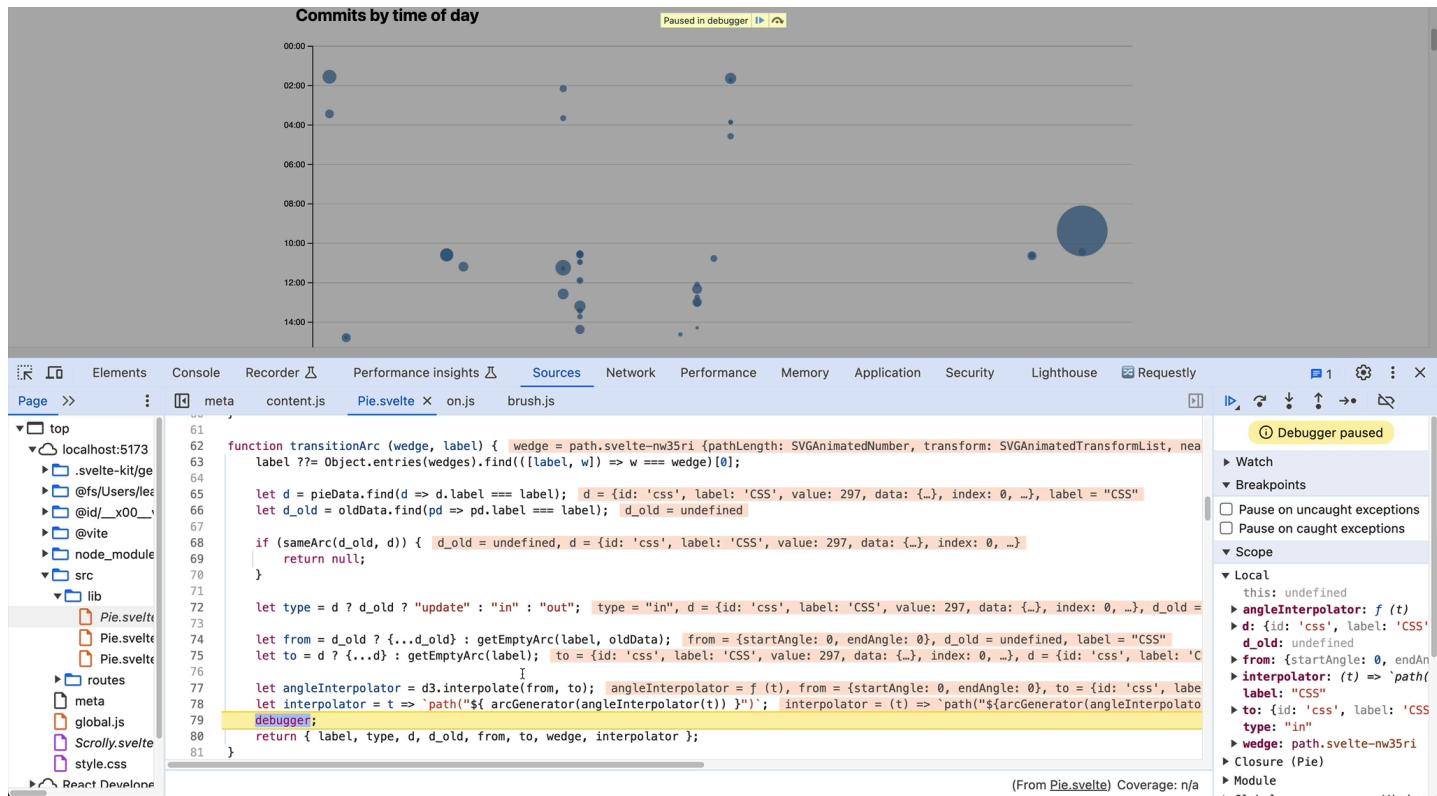
### `debugger` statement

The `debugger;` statement allows you to stop execution of your code and inspect its current state, as well as the *call stack*, i.e. how you got there.

E.g. suppose we're working on [step 3.4 of Lab 9](#) and want to debug why our interpolator didn't work as we expected.

We could add `console.log()` statements with all the local variables or print them out using Svelte expressions, but in some cases, especially when we have a lot of variables or simpler methods have failed to narrow down the issue, there is a better way: the debugger.

You can add a `debugger;` statement to stop your code at any point and inspect its current state in the [Sources panel](#). You can hover over any variable and see its current value. It looks like this:



The `debugger;` statement works like any other JS statement, meaning you can put it inside conditionals as well:

```
if (someCondition) {
  debugger;
}
```

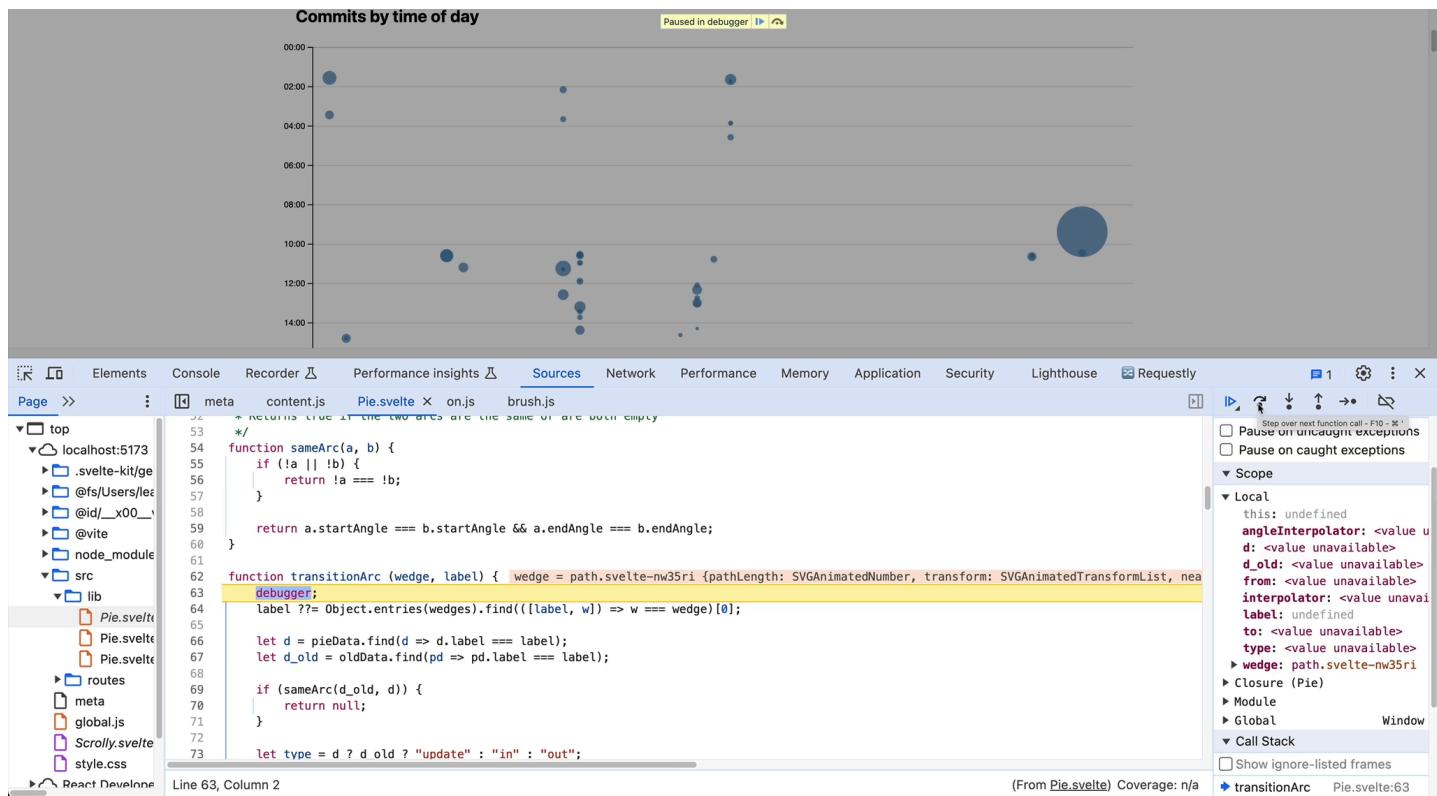
#### FYI

These statements create *breakpoints* in your code, which are points where the code execution stops. You can also set breakpoints manually in the Sources panel, without modifying the code.

## Sources panel

In the previous section we saw how to use `debugger;` statements to inspect the state of our code at any point in time. However, the Sources panel does a lot more than that.

It also allows us to *step through* our code, meaning we can execute it line by line and see the state of our variables at each point.

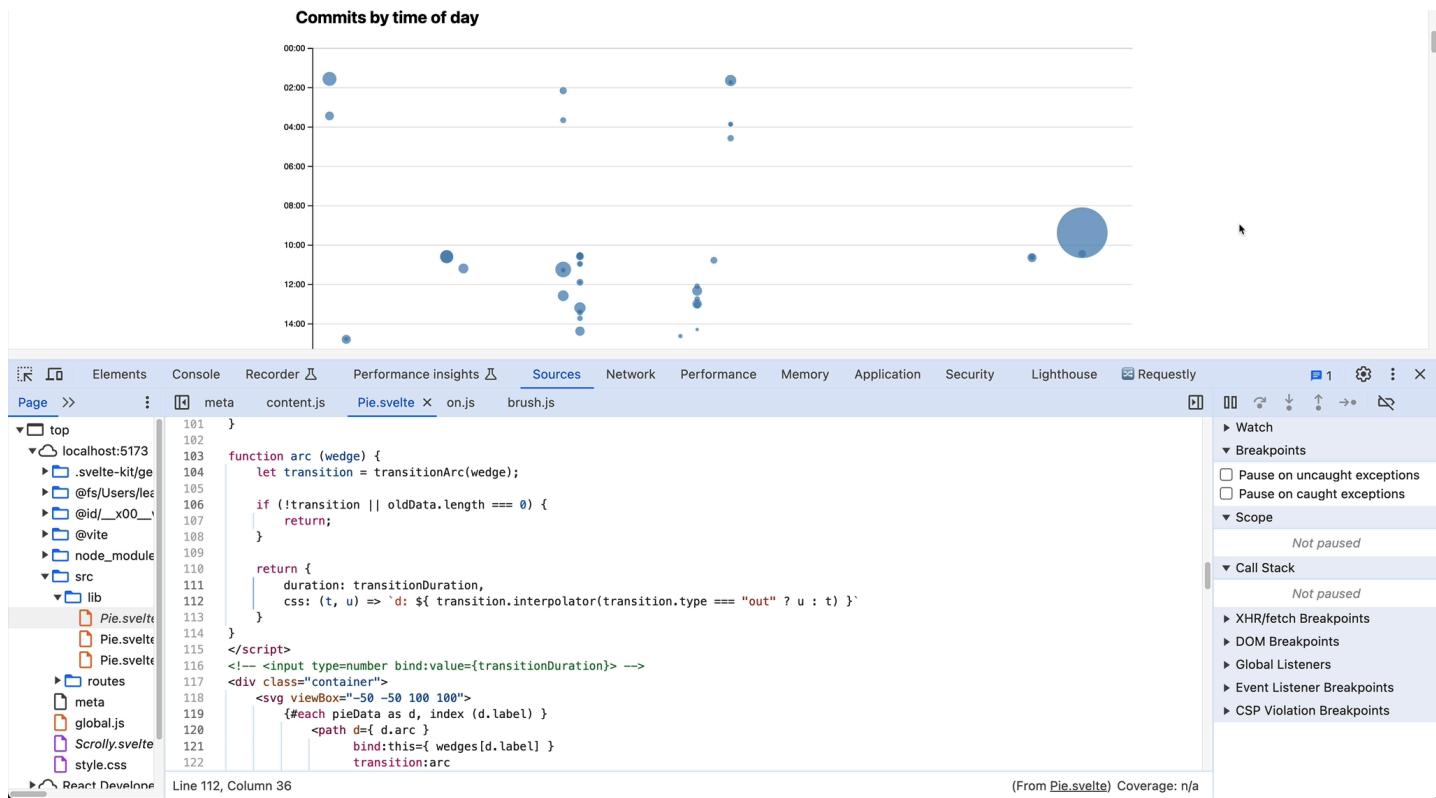


To step through our code, we use this toolbar at the top right: Here's what the icons mean:

- **Resume execution:** Continue executing the code until the next breakpoint or the end of the script.
- Buttons 2-5 are all about stepping through the code, i.e. executing the current line and moving to the next one. However, there are different ways you may want to do that, mainly around how low-level you want your inspection to be: do you actually want to see the state within each function called in your code, or only the one you are currently in?
- **Step over next function call:** This stays within the function you are currently inspecting, without moving into other functions being called. This is the most high level of them all, and the one I suggest you start with. It helps you to get a higher level overview of what is happening in your code, without having to suffer through every single line.
- **Step into next function call:** If the current line is a function call, move into the function and execute the first line.
- **Step out of current function:** Execute the rest of the current function and move to the next line *after* the function call. This is very useful when you realize you don't actually care about the function you're in and you know the bug is not there, effectively helping you reverse the decision to step into.
- **Step:** Execute the current line and move to the next one, whether that is in another function or in the current one.
- **Deactivate breakpoints:** This will deactivate all breakpoints in your code, so that it will run without stopping at any of them. This is useful when you have a lot of breakpoints and you want to run the code without stopping at any of them.

It also shows you the *call stack*, which is the list of functions that have been called to get to the current point. For example, suppose we're working on [step 3.4 of Lab 9](#) and we want to figure out if the `interpolator` function was called from a D3

Transition or a Svelte transition. We can look at the call stack, and click on the function that called it (`css`): turns out it was a Svelte transition this time!



### CAVEAT

Keep in mind that when async code is involved, the call stack may not be as useful as you'd like.

## Resolution

In most cases, >99% of debugging time is spent narrowing down the problem, and the resolution is typically pretty simple, and often takes seconds (e.g. changing a variable name).

There is one exception: when the bug is *not* in our code, but in one of the tools we're using (D3, Svelte, Mapbox, etc.) or even the browser itself (browsers have bugs too — [thousands of them!](#)) In those cases, the right thing to do is to make sure that a bug report already exists, and if not to file one. Make sure to include a reduced testcase in your bug report (see [Reducing the issue](#) above on how to create one). Reduce *very* aggressively, as a rule of thumb, most testcases are less than 10 lines of (non-boilerplate) code.

However, we usually cannot wait for a fix, so we need to find a *workaround*. This means that we need to find a way to get the same functionality, but in a different way that doesn't trigger the bug.

Finding a good workaround often involves exactly the same steps as investigating the bug in the first place: narrowing down the issue, understanding why it happens, and what exactly triggers it (make sure to include all your findings in the bug report to save the developers time and increase the odds it may be fixed soon).

For example, in [lab 9](#) we had to work around a bug in Svelte's `animate` directive. A quick investigation revealed [many bug reports around similar issues](#), so we decided not to file another one.

To investigate the bug, we clicked on `flip` from replaced `import {flip} from 'svelte/animate';` to get the `flip` source code, then pasted it as a plain function in our own code. Then started adding `console.log()` statements. We found that the problem was

related to the part that is executed before each transition only getting executed once (instead of once per transition), so the positions it was capturing were getting progressively more and more out of sync.

A workaround was to try and *fool* Svelte into thinking that every time we were transitioning, we were using a brand new function:

```
import { flip as originalFlip } from 'svelte/animate';

function getFlip() {
    return originalFlip;
}

// This will get called every time files changes:
$: flip = getFlip(files);
```

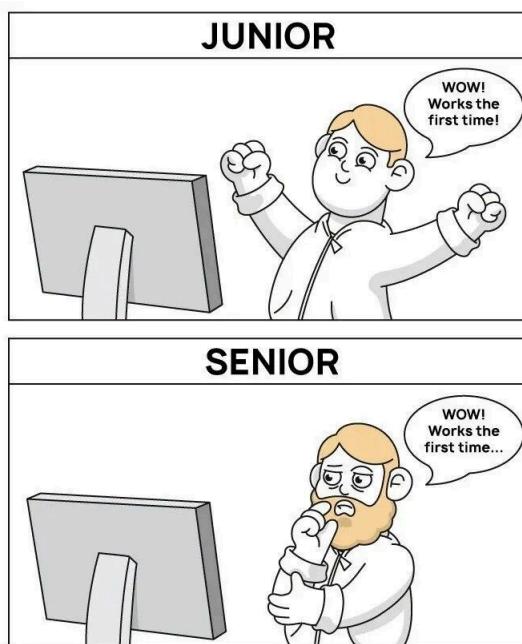
Workarounds are fragile. They may break with future updates to the library, or they may not work in all cases. E.g. in this case, if Svelte updates to be smarter about dependencies and realize that `getFlip()` returns the same function every time, our workaround will break. This is why it's important to report these bugs, so they can be fixed in the long term. Workarounds should be seen as a temporary bandaid, not a long-term solution.

#### FYI

In some cases, even the underlying technologies can be buggy! For example, when I was working on Lab 9 I [reported](#) the default SVG arc interpolation as a bug.

## Conclusion

Debugging is the least fun part of programming, despite it also being the biggest chunk of it. Especially in the beginning, it's absolutely normal to spend about 5% writing code and 95% debugging it. But even as you gain experience, debugging will often occupy around 70% of your time developing anything.



**Figure 5:** Even seasoned programmers are \_surprised\_ when their code works the first time. Image source: [5 traits to go from Junior to Senior Software Engineer - DEV Community](<https://dev.to/jenesh/5-traits-to-go-from-junior-to-senior-software-engineer-11ol>)

**It's important to remember that everyone spends a fair bit time debugging, even programmers with decades of experience.** Programming is a completely different communication paradigm than anything else we do. Natural (human) languages are full of ambiguity. There is so much that is simply implied in every sentence, and the other humans can just infer it.

In programming, we have to tell the computer *exactly* what to do, with detailed steps that leave no room for ambiguity. This can be very hard as it goes against everything we have ever learned about communication!

**NOTE**

AI may change this in the future, there are a lot of efforts to make programming more like natural language, but we're not there yet.

**Never feel like making mistakes and having to debug them is a personal failing of yours.** It's a crappy part of programming, but one that we all go through.