CSS Variable Wizardry

— STUDY GUIDE —

What Are CSS Variables?

Custom properties that store specific values for reuse throughout a stylesheet. Think of them as named ‘runes’ or **labels for values**, not mini CSS files. They turn your stylesheets into a living control panel.

```
/* Declaration in :root */
:root {
  --rune-text-strong: #ffffff;
}
```

Why They Matter

Reusability

Write a value once, use it everywhere. Avoids repetition and ensures consistency.

Maintainability

Update a value in one place to change it globally. Refactoring becomes trivial.

Themability

Instantly switch looks (e.g., light/dark mode) by changing variable values on a parent element.

Scalable Systems

The foundation of modern design systems, allowing you to build and evolve complex UIs with confidence.

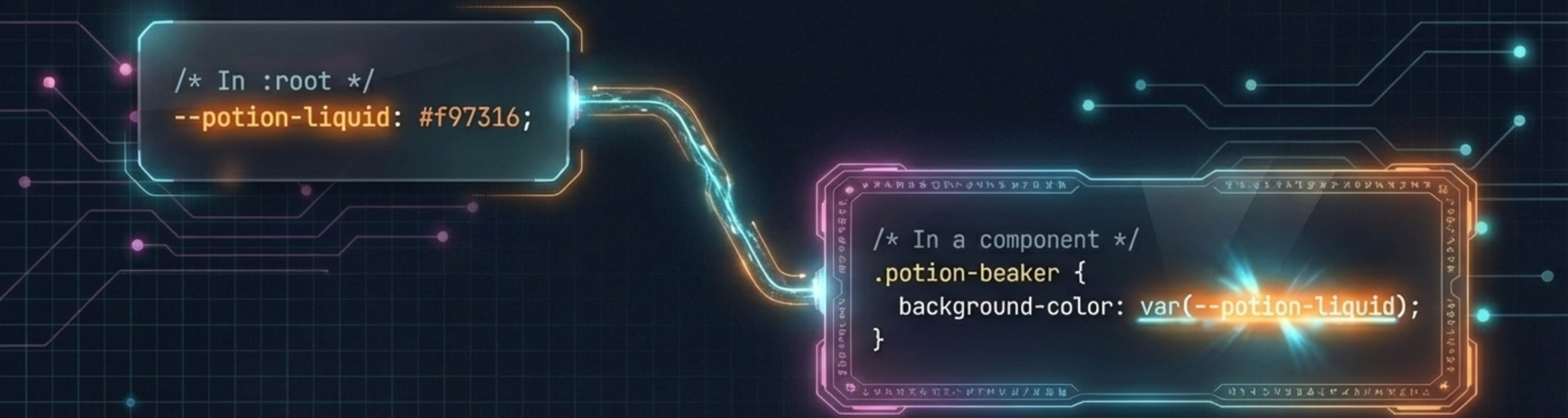
Declaring Global Runes in `:root`

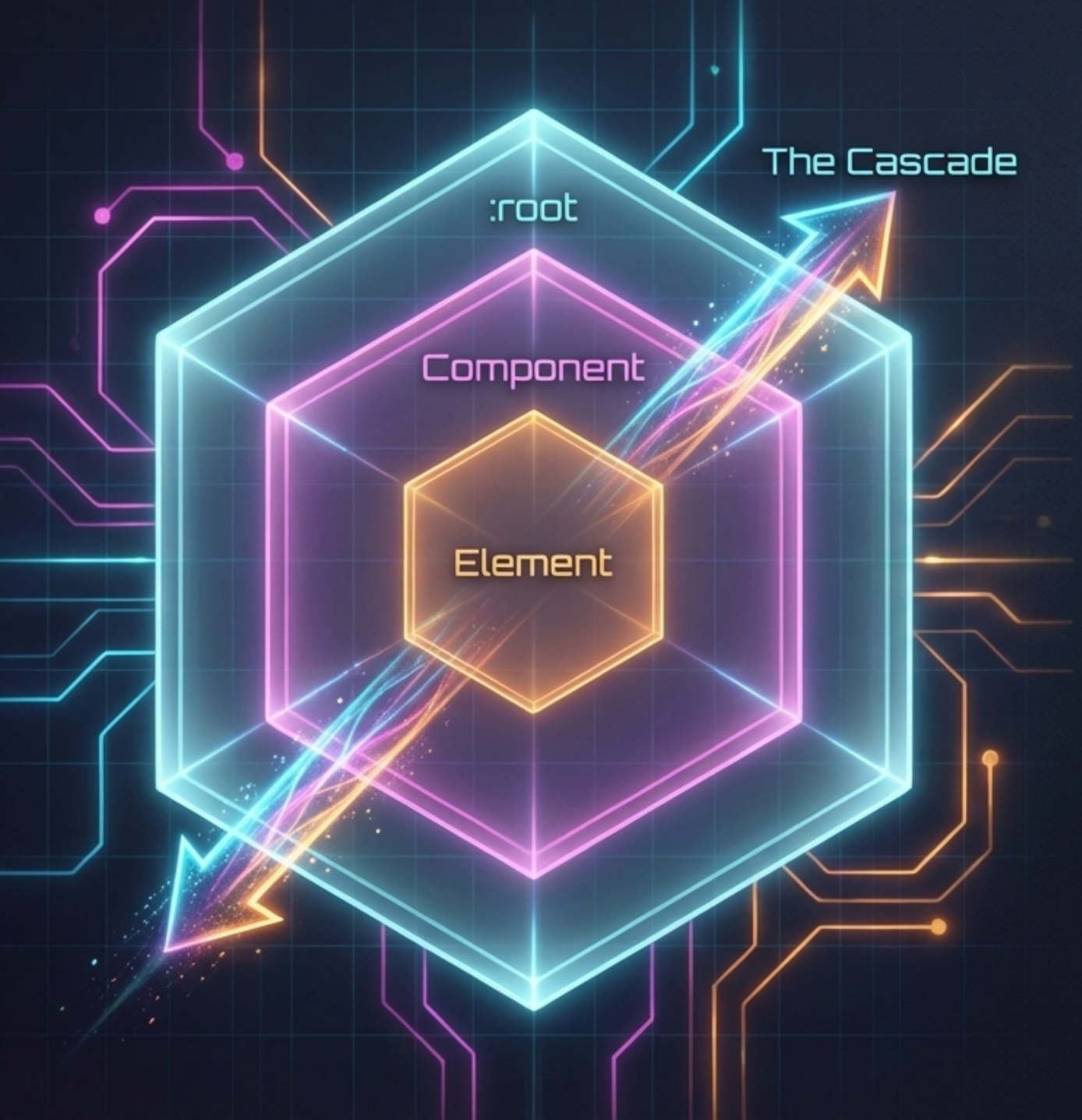
Define variables in the `:root` selector to create global tokens accessible to your entire application. This is the central source of truth for your design system.

```
:root {
  --page-bg: #0f172a;
  --arcane-accent: #f97316;
  --glyph-border: #fde68a;
  --rune-text-strong: #ffffff;
  --potion-liquid: #f97316;
  --potion-glow: 0 0 32px rgba(249, 115, 22, 0.65);
}
```

Using Variables with `var()`

The `var()` function retrieves the value of a custom property. It's the incantation used to summon a rune's power. It literally means: "grab the computed value of this variable from its cascade."





Scope & The Cascade

Variables don’t dodge the cascade—they **use** it. Values are inherited from parent elements unless a more specific (local) value is defined.

Hierarchy of Power

* **1. `root` (Global):** The base design tokens for the entire document.
* **2. Component/Section (Local):** Overrides global values for a specific context, like a `.dark-theme` wrapper.
* **3. Element (Most Specific):** The final override, closest to the point of use. The deepest spell wins.

Building a Color System

Use variables to define a clear, consistent, and maintainable color palette. Changing a rune once can shift the entire lab's mood.

Local Overrides

Override a global variable inside a specific selector to change its value only within that scope. Perfect for creating component variants without writing new properties.

Global Value (Inherited)

Beaker uses the default `-potion-liquid: #22d3ee;` from its parent lab.

Local Override (Scoped)

```
/* Bottom-shelf variant: override only the liquid */
.potion-card--lower-override {
  --potion-liquid: #ec4899; /* neon pink */
}
```

Theme Switching with JavaScript

Hand the wand to JavaScript to rewrite variable values at runtime.
JavaScript only updates the tokens; CSS and the cascade handle the rest.


```
// JS defines theme values
const THEMES = {
  sun: { liquid: '#f97316', glow: '...' },
  violet: { liquid: '#ec4899', glow: '...' }
};

// And sets them on the root element
function applyTheme(key) {
  const theme = THEMES[key];
  document.documentElement.style.setProperty(
    '--potion-liquid', theme.liquid
  );
}
```

Variables + Functions

Combine variables with CSS functions to create dynamic and responsive values.
Use them as unitless ‘intensity knobs’ for motion, shadows, and spacing.

calc()

clamp()

min()

```
:root {
  --lift-distance: 14px; /* The intensity dial */
}

.card:hover {
  /* The spell */
  transform: translateY(calc(-1 * var(--lift-distance));
}
```

Before vs. After Refactor

Refactoring to variables makes your CSS cleaner, more readable, and infinitely more maintainable. You stop hardcoding colors like it's 2009 and start wielding design tokens.

Before: Hard-Coded Values

```css
.button-primary {
  background: #ec4899;
}

.modal-header {
  border-bottom: 2px solid #ec4899;
}

.link:hover {
  color: #ec4899;
}
```

After: Variable-Driven

```css
:root {
  --brand-accent: #ec4899;
}

.button-primary {
  background: var(--brand-accent);
}

.modal-header {
  border-bottom: 2px solid
    var(--brand-accent);
}

.link:hover {
  color: var(--brand-accent);
}
```

Common Mistakes to Avoid

Forgetting Fallbacks

Not providing a fallback value in `var()` can break styles if a variable is missing.

```
color: var(--missing-var, black);
```

Unclear Naming

Vague names like `--c1` or `--main-red` create confusion. Use semantic names like `--color-brand-accent` or `--color-status-danger`.

Storing Properties, Not Values

A variable stores only the value part, not an entire declaration.

```
Incorrect:  --fancy-border: border: 3px solid hotpink;
Correct:  --fancy-border-color: hotpink;
```

Best Practices for Potent Magic

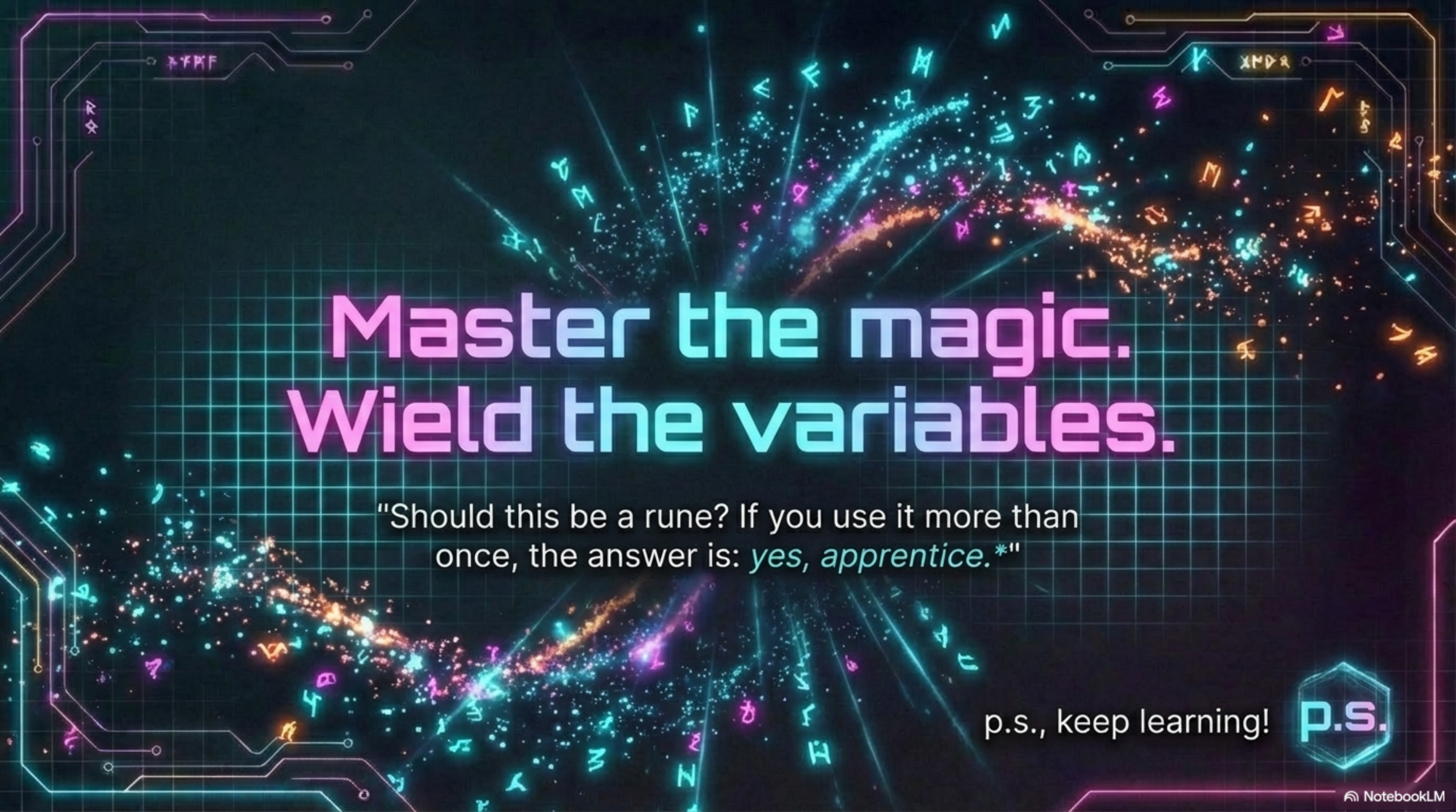
Consistent Naming Conventions: Use a clear system, such as `[category]-[property]-[variant]`.

Example: `--color-brand-primary`, `--font-size-heading-lg`

Global Palette, Local Consumption: Define foundational brand tokens in `:root`. Components should consume these tokens, not redefine them.

Component Defaults: Give components safe, self-contained default values. This allows them to look correct even out of context, while parents can still override them.

Example:
```
.card { --card-bg: #0f172a;
        background: var(--card-bg); }
```


Master the magic.
Wield the variables.
"Should this be a rune? If you use it more than once, the answer is: *yes, apprentice.*"
p.s., keep learning!
p.s.
NotebookLM