Beginner Programming with Large Language Models

Seth J. Chandler, with help from AI

September 27, 2025

Chapter 1

You can code

Introduction: A Changing Expectation

For most of the history of legal education, the law student's intellectual toolkit has been confined to words, arguments, and a familiarity with doctrine. Courts and legislatures did the work of crafting rules, lawyers the work of interpreting and applying them, and the only "technology" required was the ability to look up authorities in books and later in digital databases like Westlaw and Lexis. Programming was seen as the province of engineers and computer scientists, a foreign domain of mysterious syntax and arcane tools.

That boundary is dissolving. Large language models (LLMs) such as ChatGPT and Claude can now translate ordinary English instructions into working computer programs. This does not mean every law student should become a full-fledged software engineer. But it does mean that the bar to entry for building useful, small programs is far lower than it was even a few years ago. The lawyer of the near future will increasingly find themselves using code to automate research tasks, analyze data, or even generate teaching materials. The skill that seemed alien yesterday is becoming accessible today.

This essay is written for students without technical backgrounds. Its purpose is to encourage experimentation, while also warning against illusions of simplicity. Generating code with LLMs is often the easiest task these systems can do, but as our own attempts will show, the process requires patience, iteration, and the willingness to learn a few basics of how computers actually run programs.

Our Experiment: Four Mini-Programs

Let's begin with a concrete narrative drawn from our own conversations. I asked LLMs to produce four simple programs of interest to law students:

1. **A Python script to extract case citations** from the text of a court opinion.
2. **A JavaScript web page with flashcards** on famous cases.
3. **A Go command-line tool to count word frequencies** in an opinion.
4. **An Excel spreadsheet with a functioning neural network** for pedagogical demonstration.

The first three are less than fifty lines long, each runs on a different language platform, and each addresses a task relevant to legal education. The fourth—built with Claude Desktop—demonstrates how even complex computational models can now be created in familiar tools like Excel. Together they show both the promise and the pitfalls of programming with LLMs. *For traditional code, programs ended up taking less than 75 lines, and as a practical rule of thumb, debugging problems tend to be roughly proportional to lines of code—brevity helps.*

Example One: The Case Citation Extractor (Python)

The first program was a Python script that takes a block of text and uses a regular expression (a kind of pattern-matching formula) to identify citations such as *Brown v. Board of Education, 347 U.S. 483 (1954).*

The very first version was plausible but flawed. It hung waiting for input when I simply ran the program. The model had written code that assumed I would pipe text into the program via the command line, but as a law professor with limited programming background, that was not how I naturally tried to use it. I simply typed `python find_cases.py` and expected it to do something. Instead it froze, waiting for me to provide input I didn't know I had to give.

This is a key lesson: *LLMs often write code that works in theory, but not in practice for beginners.*

When I pointed this out, ChatGPT revised the script to use sample text when no input was provided. That was better, but the first attempt still did not work correctly. I had to emphasize that I wanted the simplest possible behavior: just run the script and see something happen, unless I explicitly gave it a file or text. Only after this clarification did we arrive at a minimal version that met that need.

What students should notice here is not just the final success but the process. I had to iterate, clarify, and push back against overengineering. The first instinct of the model was to show off how "Unix-like" it could be, rather than to serve the goal of an introductory teaching demo. Without persistence, a novice might have given up.

Final code. (Inserted exactly as provided.)

```
#!/usr/bin/env python3
import re, sys

PATTERN = re.compile(r"""(
    [A-Z][A-Za-z0-9.&''\- ]+\s+v\.\s+[A-Z][A-Za-z0-9.&''\- ]+,\s+
    \d{1,4}\s+[A-Z][A-Za-z.\d\s]*?\s+(?:\d{1,4}|___)\s*\(
    [A-Za-z.\d\s]*?\d{4}\)
)""", re.VERBOSE)

SAMPLE = (
    "In Brown v. Board of Education, 347 U.S. 483 (1954), the Court...\n"
    "See Miranda v. Arizona, 384 U.S. 436 (1966) and United States v. Texas, 599 U.S. ___ (2023)
    "Fifth Circuit example: Texas v. United States, 809 F.3d 134 (5th Cir. 2015)."
)

def read_text():
    if not sys.stdin.isatty():              # piped input
        return sys.stdin.read()
    if len(sys.argv) > 1:                   # file path as first arg
        with open(sys.argv[1], "r", encoding="utf-8") as f:
            return f.read()
    return SAMPLE                            # default

def main():
    text = read_text()
    seen = set()
    for m in PATTERN.findall(text):
        if m not in seen:
            seen.add(m)
            print(m)

if __name__ == "__main__":
    main()
```

Example Two: The Flashcards (JavaScript)

The second program was more straightforward. I asked for a JavaScript page that would display flashcards with law-related questions. The model produced a simple HTML file that

could be opened directly in a web browser. The code contained a list of cases and holdings (e.g., *Marbury v. Madison*, *Brown v. Board*, *Miranda v. Arizona*). A "Show Answer" button revealed the text, and a "Next" button cycled through the questions.

This example shows the upside of programming with LLMs: *it really can be this easy.* I copied and pasted the code into a text file named `flashcards.html`, opened it in my browser, and had an instant working tool. The script was short, readable, and even had a bit of style. A student could add their own questions by editing a few lines in the array of cards.

Here, the model's output required no debugging. The risk of overengineering was avoided, and the experience was empowering: with only a single command ("write a JavaScript program with flashcards"), I had a working educational app.

Final code. (Inserted exactly as provided.)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Law Flashcards</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <style>
    body { font-family: system-ui, Arial, sans-serif; margin: 2rem; }
    .card { max-width: 700px; border: 1px solid #ccc; border-radius: 8px; padding: 1rem; }
    .q { font-size: 1.1rem; margin: 0 0 0.75rem; }
    .a { margin: 0.5rem 0 1rem; display: none; }
    .controls { display: flex; gap: 0.5rem; }
    button { cursor: pointer; padding: 0.5rem 0.8rem; border-radius: 6px; border: 1px solid #888
  </style>
</head>
<body>
  <h1>Law Flashcards</h1>

  <div class="card">
    <p class="q"></p>
    <p class="a"></p>
    <div class="controls">
      <button id="show">Show Answer</button>
      <button id="next">Next</button>
    </div>
    <p id="progress" style="font-size:0.9rem; color:#555;"></p>
```

```
</div>

<script>
  const cards = [
    {
      q: "What did Marbury v. Madison (1803) decide?",
      a: "Established judicial review: the Supreme Court may declare laws unconstitutional."
    },
    {
      q: "What principle comes from Erie Railroad Co. v. Tompkins (1938)?",
      a: "Federal courts in diversity apply state substantive law and federal procedural law."
    },
    {
      q: "What was the holding of Brown v. Board of Education (1954)?",
      a: "Racially segregated public schools are unconstitutional under the Equal Protection C
    },
    {
      q: "What warning is required by Miranda v. Arizona (1966)?",
      a: "Suspects must be informed of rights to remain silent and to counsel before custodial
    },
    {
      q: "What does New York Times Co. v. Sullivan (1964) require for public officials?",
      a: "They must prove actual malice to recover for defamation."
    }
  ];

  let i = 0, shown = false;
  const qEl = document.querySelector('.q');
  const aEl = document.querySelector('.a');
  const showBtn = document.getElementById('show');
  const nextBtn = document.getElementById('next');
  const progEl = document.getElementById('progress');

  function render() {
    qEl.textContent = cards[i].q;
    aEl.textContent = cards[i].a;
    aEl.style.display = shown ? 'block' : 'none';
    showBtn.textContent = shown ? 'Hide Answer' : 'Show Answer';
    progEl.textContent = `Card ${i + 1} of ${cards.length}`;
  }
```

```
    showBtn.addEventListener('click', () => { shown = !shown; render(); });
    nextBtn.addEventListener('click', () => { i = (i + 1) % cards.length; shown = false; render(

    render();
  </script>
</body>
</html>
```

Example Three: The Word Frequency Counter (Go)

The third program was written in Go, a modern systems language designed to be fast and simple to compile. The task was to count the most common words in a legal opinion.

The initial version worked, but the output was not very useful. Words like "s," "a," "that," and "ct" dominated the results. In other words, the lack of a good stopword list made the analysis unhelpful. Once I reported this, the model expanded the list and filtered out very short words. The program became more functional, though we could keep refining it indefinitely.

The lesson here is that *you don't always know what is wrong until you run the program.* The need for better stopwords only became apparent once I saw the output. Students should expect this pattern: write code, run it, learn from the shortcomings, revise, and repeat.

Final code. (Inserted exactly as provided.)

```
package main

import (
        "bufio"
        "fmt"
        "os"
        "regexp"
        "sort"
        "strings"
)

func main() {
        if len(os.Args) < 2 {
                fmt.Println("Usage: go run main.go FILE")
                return
        }
```

```
file, err := os.Open(os.Args[1])
if err != nil {
        panic(err)
}
defer file.Close()

// Expanded stopwords: common English + legal filler
stopwords := map[string]bool{
        "a": true, "an": true, "and": true, "are": true, "as": true, "at": true,
        "be": true, "but": true, "by": true, "for": true, "from": true,
        "has": true, "have": true, "in": true, "is": true, "it": true,
        "of": true, "on": true, "or": true, "s": true, "that": true,
        "the": true, "this": true, "to": true, "was": true, "were": true,
        "with": true, "ct": true, "u": true, "v": true,
}

counts := make(map[string]int)
re := regexp.MustCompile(`[A-Za-z]+`)

scanner := bufio.NewScanner(file)
for scanner.Scan() {
        words := re.FindAllString(scanner.Text(), -1)
        for _, w := range words {
                word := strings.ToLower(w)
                if stopwords[word] {
                        continue
                }
                if len(word) < 3 && word != "us" {
                        continue
                }
                counts[word]++
        }
}

type kv struct{ Word string; Count int }
var pairs []kv
for w, c := range counts {
        pairs = append(pairs, kv{w, c})
}
sort.Slice(pairs, func(i, j int) bool { return pairs[i].Count > pairs[j].Count })
```

```
        fmt.Println("Top 10 words:")
        for i := 0; i < len(pairs) && i < 10; i++ {
                fmt.Printf("%s: %d\n", pairs[i].Word, pairs[i].Count)
        }
}
```

Example Four: The Neural Network in Excel (with Claude Desktop)

The fourth example represents a different kind of programming altogether—one that doesn't require learning new syntax or command-line tools. Using Claude Desktop, we created a fully functional neural network entirely within Microsoft Excel. This example demonstrates how LLMs can now help create sophisticated computational models using tools that law students already know.

The task was pedagogical: build a minimal neural network that could demonstrate machine learning concepts through the familiar interface of a spreadsheet. The result was a complete implementation with 20 training samples, a 5-neuron hidden layer, forward propagation, backpropagation, and gradient descent—all using Excel formulas.

What makes this example particularly relevant for law students is the accessibility. Unlike Python, JavaScript, or Go, Excel requires no installation of programming environments, no command-line navigation, and no debugging of syntax errors. The formulas are visible, the calculations transparent, and the learning process observable in real-time.

The development process, however, was far from trivial. It required several iterations:

1. The first version had the basic structure but contained circular references that made manual training impossible.

2. We then attempted to use Excel's iterative calculation feature, but this proved too complex for pedagogical purposes.

3. The breakthrough came with a "copy-paste" design: green cells with formulas calculate the next iteration's weights, which students manually copy as values into yellow cells containing current weights.

4. Even this required debugging—fixing cell references, correcting range errors, and ensuring the gradient calculations were properly structured.

This example illustrates several key advantages of Excel as a programming platform for beginners:

Immediate visibility: Every calculation is visible. Students can see exactly how each weight affects the output, how errors propagate backward, and how gradients drive learning.

No syntax barriers: Excel formulas use a syntax that many students already understand. `=AVERAGE(B2:B10)` is more intuitive than `np.mean(data[1:10])`.

Interactive learning: Students can change learning rates, modify weights manually, and immediately see the effects. This tactile interaction aids understanding.

Universal availability: Excel is installed on virtually every law school computer. There's no need to configure Python environments or install compilers.

The pedagogical innovation here was making the training process manual. Students copy calculated weight updates from green cells to yellow cells, physically performing gradient descent. This transforms an abstract algorithm into a concrete, repeatable action. Each copy-paste operation is one training step, and students can watch the mean squared error decrease with each iteration.

[Screenshot of Excel Neural Network will be inserted here]

Yet this example also shows the continued need for human expertise. Despite Claude's sophisticated capabilities, creating this spreadsheet required:

- Understanding that circular references would break the manual training flow
- Recognizing that a copy-paste mechanism would be more educational than automation
- Debugging numerous formula errors related to cell ranges and references
- Iterating through multiple designs before finding one that was both functional and pedagogical

The lesson is that while LLMs can generate sophisticated Excel programs, *human judgment remains essential for design decisions and debugging.* The model could write the formulas, but it took human insight to recognize that manual copying would make the learning process tangible for students.

What It Takes to Run the Code

One of the most important points to stress is that generating code is only the first step. To make these programs run, I had to copy and paste them into actual files and execute them in real environments. For Python, I ran `python find_cases.py`. For JavaScript, I opened the HTML file in a browser. For Go, I ran `go run main.go opinion.txt`. For Excel, I opened the file and enabled macros if necessary.

This required installing interpreters or compilers (except for Excel), learning how to navigate the command line, and mastering small but non-obvious details like putting filenames with spaces inside quotes. Even installing Visual Studio Code (or another editor) and creating files in the right place is not trivial for someone with no prior experience.

The message to students is: *there is a learning curve.* Do not be discouraged when you stumble over details that feel embarrassingly small. Everyone who programs has fought with quotes, missing colons, or "file not found" errors. The difference today is that LLMs make it possible to bypass the hardest part—the act of writing working logic—and focus on learning the mechanics of running it.

Excel stands out as an exception here: the "installation" is already done, the "execution environment" is a familiar application, and "running" the program means clicking on cells. This accessibility makes it an ideal starting point for law students venturing into programming.

Why LLMs Are Good at Generating Code

It is worth pausing to ask: why are LLMs especially strong at generating code?

The answer has two parts:

1. **Training data.** Models like ChatGPT and Claude are trained on massive corpora of publicly available text, and software code is one of the most common and high-value text types on the internet. There are millions of GitHub repositories, tutorials, Q&A posts, and documentation pages. The model has ingested a huge share of this material.

2. **Self-interest.** The people building these models are themselves programmers. They are naturally motivated to ensure the models excel at writing code, because they want tools that help them in their daily work. As a result, generating code is one of the tasks where LLMs are most advanced relative to other skills.

The practical effect is that *producing working code is probably what LLMs currently do best.* That is why this is such a good entry point for law students. You are not asking the model to do something speculative; you are asking it to do what it is already optimized to do.

The Tools: From ChatGPT to Integrated Environments

So far I have described how to use LLMs like ChatGPT and Claude to generate small programs. But for more complex projects, you will likely need more integrated tools. A few worth naming:

- **GitHub Copilot (built on Codex)**: integrates directly into Visual Studio Code and suggests code as you type.
- **Claude Desktop**: can see your screen, manipulate files directly, and create complex programs in familiar tools like Excel.
- **Claude Code (from Anthropic)**: optimized for large codebases and context-aware edits.
- **Gemini CLI (Google)**: command-line interface that can generate and refactor code quickly.
- **Cursor**: a popular AI-first code editor that blends model suggestions with standard development tools.
- **Replit**: an online environment that lets you run code in the browser, with strong AI integrations.
- **JetBrains IDEs (like PyCharm or GoLand)**: increasingly offer AI assistants alongside robust language support.

The pattern is clear: *vanilla prompting is great for small demos, but serious work benefits from environments where the AI can see your files, your errors, and your context all at once.*

The Future: Legal Arguments as Code

The deeper implication for law students is not just about programming. It is about imagining a world two years from now when LLMs have the same fluency in building legal arguments that they currently have in building code.

Consider how Midpage.ai is already experimenting with "AI-first" approaches to legal research and drafting. Imagine an LLM that, when asked, produces a well-cited argument for why a motion should be granted, just as today it produces a working Python function. The analogy to code is instructive: first the system will be clumsy, then it will become reliable, and eventually it may be second nature.

This means that the exercises we are doing today with code are not just about software. They are *previews of how lawyers will practice tomorrow.* The patience you develop in debugging a Go program is the same patience you will need in refining AI-generated briefs. The insistence on minimal examples in programming—start simple, then iterate—will be just as crucial in legal practice with AI.

Advice for Beginners: Four Starter Approaches

Students often ask which languages or tools are best to learn. My own view is that a subset of four covers the ground:

1. **Python** – ubiquitous, easy to read, with countless legal and scientific libraries. Downsides: dependency hell, sometimes slow.
2. **JavaScript** – the language of the web. If you want interactive flashcards, data visualizations, or browser-based apps, this is the one to know.
3. **Go** – fast, simple to compile, avoids many dependency problems, and is growing in popularity. Especially good for text processing and server tools.
4. **Excel with LLMs** – the most accessible entry point. No installation required, familiar interface, immediate visual feedback. Perfect for understanding algorithms before moving to traditional code.

Mastery of all four is not required. But experimenting with one from each family will give you a sense of how code can be used in law. Starting with Excel might be the gentlest introduction before moving to traditional programming languages.

What Can Go Wrong

To keep expectations realistic, let's list the ways things went wrong in our experiments:

- The Python citation extractor initially froze waiting for input.
- Even after fixing that, the first fallback logic was not correct until simplified further.
- The Go frequency counter produced useless results until stopwords were improved.
- Small details like quotes around filenames can cause confusing errors.
- The Excel neural network had circular reference problems requiring a complete redesign.
- Even with Claude Desktop's capabilities, multiple iterations were needed to get formulas working correctly.

These are not fatal flaws. But they show that *LLMs do not deliver perfect solutions on the first try.* You must be prepared to iterate, learn by doing, and adjust your instructions.

Why Minimal Examples Matter

The most important pedagogical lesson is this: *always start with minimal examples.*

Had I asked for a "robust legal text analyzer," the model would have produced hundreds of lines of code with multiple libraries. It almost certainly would not have worked on the first try. By asking for "a program under 50 lines," we forced simplicity, and simplicity made debugging feasible.

For students, the takeaway is clear: start small. A script that prints the top ten words in an opinion is more useful as a learning tool than a half-working "comprehensive research platform." Once you have the basics, you can grow. Even our Excel neural network, though complex in concept, was kept minimal—just enough neurons and training samples to demonstrate the concept without overwhelming complexity.

Conclusion: Encouragement Without Illusion

The message of this essay is encouragement, but not hype. Law students without technical backgrounds can, with today's LLMs, build working programs. The barrier to entry is far lower than ever before. Whether copying and pasting code into Visual Studio or creating formulas in Excel, programming is within reach for anyone.

At the same time, there are limits. Learning even the basics of running code requires patience. You need to get somewhat comfortable working in "terminal mode." You need to learn Programs will have bugs. LLMs will overcomplicate instructions or misinterpret your goals. You must iterate, simplify, and keep expectations modest. Even with familiar tools like Excel, creating sophisticated programs requires careful design and debugging.

Yet this modesty is precisely what makes the enterprise worthwhile. By starting with small successes—a flashcard page, a citation extractor, a word counter, or a spreadsheet that learns—you build confidence. And in doing so, you prepare yourself for a future in which the same ease will apply not just to code but to law itself. Two years from now, when LLMs generate legal arguments with the fluency they now generate Python or Excel formulas, you will be glad you practiced on the easy side first.

Programming with LLMs is not a distraction from legal education. It is a preview of its future.

Chapter 2

Working through frustration

2.1 Why this matters now

Everyone is encouraging vibe coding —describing what you want to an LLM and iterating. And it can indeed be a gentle on-ramp to computing or a tool from which even advanced programmers can benefit. You can copy short programs into an editor, run them, and get useful results in minutes. There is a learning curve, and some new vocabulary, but most obstacles are solvable with a few practical habits: keep examples minimal, run early and often, read the first clear error message, and iterate. The goal is not to become a software engineer; it is to acquire just enough operational skill to turn ideas into small working tools.

2.2 What vibe coding is (and is not)

Vibe coding is not "magic code without learning." It is a cooperative workflow: you state the outcome, the LLM proposes code, you run it, observe what fails, and refine the prompt or the code. The power comes from rapid feedback, not from skipping every prerequisite. With the right guardrails, it is realistic for absolute beginners.

2.3 Ten hidden assumptions that quietly derail beginners

The frustration most newcomers feel comes from silent prerequisites. The absence of even one of these can lead to emotional distress. When writers or coders neglect multiple silent prerequisites, the results can range from sadness, to unwarranted feelings of inadequacy, to pulling one's hair out. Here are eight common unspoken prerequisites and what to do about them. The key is to recognize when you are missing a key piece and how to swiftly solve the problem.

1. Terminal fluency is often assumed. Without cd, ls/dir, pwd, and how to run a script, nothing starts. Spend some time in study and learn mode with this curriculum.

    a Ask to be taught the ten most common terminal commands a vibe coder will need.

    b Ask about relative vs. absolute paths and how failure to understand them can cause problems.

    c Ask how to move long file names and paths into your terminal session in your operating system.

    d Ask about how to use arrow keys to avoid excessive typing. Discuss various shells.

2. It is assumed you know about Python "dependency hell": what are its signs and how to escape it. This one is super frustrating because it can happen frequently, particularly if you are trying to use frontier code, and because, without the requisite background, it can be a very difficult problem to solve. Here is a concise curriculum outline for an AI learning assistant, focusing on the core concepts of Python dependency management.

2.3.1 Python Dependency Management Curriculum

1. Introduction to Dependencies

- **Core Concept:** Dependencies are the external packages a project requires.
- **The pip Problem:** Standard pip installs packages globally or into a simple virtual environment, but it has a naive dependency resolver that can easily lead to version conflicts (“dependency hell”).
- **The Solution:** Use a modern dependency manager like **Poetry** or **Pipenv**. These tools provide superior dependency resolution and create project-specific, isolated environments with a **lockfile** (poetry.lock or Pipfile.lock) to guarantee deterministic and reproducible builds.

2. Recognizing Problems

- **Primary Symptom:** A ModuleNotFoundError is the most common indicator of a missing or incorrect dependency.
- **Proper Diagnosis:** Avoid pip list. Instead, use poetry show –tree or pipenv graph to visualize the complete dependency tree and understand the relationships and versions required by each package.

3. The Modern Workflow

- **Initialization:** Start a project by creating its environment (poetry new ⟨project⟩ or pipenv –python ⟨version⟩).
- **Adding Dependencies:** Install packages using the manager (poetry add ⟨package⟩ or pipenv install ⟨package⟩). The tool will resolve the correct versions of all sub-dependencies and update the lockfile.
- **Execution:** Run all commands and scripts through the manager (poetry run ⟨command⟩ or pipenv run ⟨command⟩) to ensure they execute within the correct isolated environment.

3. Project layout and imports are assumed. Files in the wrong place cause "module not found."

4. $PATH configuration is assumed. If your shell cannot find executables, commands fail. And if you don't even know what this means, it can make the problem even more frustrating. Use study and learn mode for a 10 minute lesson on how to deal with $PATH in your operating system.

5. Error reading is often assumed. You may be presented with a "stack trace" and have no idea what it is even is or how to read it. Stack traces have a structure; the first actionable line is usually near the bottom. Solution: give the error messages to an LLM and ask for help. Do this by cut and paste of text or a screen capture. LLMs are often great at dealing with error messages. Just give it enough context to understand what was going on when the problem arose.

6. Quoting and spaces are assumed. Unquoted file paths break commands; learn to use quotes and tab-completion. Ask study and learn mode for a brief lesson on how to get file paths into the terminal when using your operating system. Ask about drag and drop. You can learn this in 5 minutes and avoid a lot of misery.

7. Version control is assumed. Without Git, experiments feel risky and recovery is painful. Solution: spend 30 minutes in study mode learning how to use Git and Github.

Transition. Once you know these pitfalls, you can adopt a few guardrails that make vibe coding feel coherent instead of fragile.

2.4 Keep it simple

2.4.1 Start without frameworks or scaffolding

Many vibe coders and large language models will encourage you to use "frameworks" and various forms of "scaffolding." And if you were not a beginner, that advice would make a lot of sense; but it can definitely become "overengineering" for beginners. (LLMs love to overengineer). A framework is a large, prebuilt set of conventions and libraries that chooses much of your project's structure for you (Django or Flask for Python; React, Next.js, or Vue for browser apps; Ruby on Rails for full-stack sites). Scaffolding tools are generators that create many files and settings for a new project (django-admin startproject, npm create vite@latest, create-react-app, Rails generators, Yeoman). These features are valuable once you understand the moving parts, but for first projects they introduce indirection and additional commands that can fail for opaque reasons. Practical rule for beginners: one file, no build step. A single HTML file for browser JavaScript, a single .py for Python, or a single main.go for Go. Tell the LLM you are a beginner and ask it to use the most basic functionality possible. You can make the LLM feel better by telling it that later on, they can "refactor" the code to use a favored framework.

2.4.2 Defer Docker

If an AI answer says “just use Docker...,” run. Docker “containers” are excellent for reproducibility and deployment, but they add images, containers, volumes, networks, and Dockerfiles—more layers, more seemingly incomprehensible vocabulary, esoteric distinctions between images and containers, more places to break. Prefer native installs at first. For Python, use a simple virtual environment or Poetry. For Go, compile or run directly. Return to Docker later when you can articulate a deployment need it uniquely solves. Spend at least an hour in Study Mode and watch several hours of YouTube videos to figure out how the world's biggest kludge works.As you may have gathered, I hate Docker and avoid it at all cost. It is a giant bandaid over fundamental architectural problems.

2.4.3 Pick low-friction language targets for tiny projects

When you do not plan to learn the language in depth yet, reduce setup:

- JavaScript in the browser: no installs; put a script tag in an .html file and open it.
- Go: strong standard library, fast, and compiles to a single binary; few dependency pitfalls.
- Python: excellent and ubiquitous; keep early projects dependency-free or manage them with Poetry.

There is a trade-off to acknowledge: JavaScript and Go may have fewer ready-made packages for niche legal tasks, but that may well be acceptable for first projects whose purpose is confidence and momentum.

2.4.4 Python without the pain: Poetry for clean, reproducible projects

Python's chief beginner hazard is dependency hell (conflicting versions, global installs, broken environments).[1] Poetry mitigates this with project isolation, precise version locking, and a memorable workflow. Minimal Poetry flow: 1. pipx install poetry (or install per docs) 2. poetry new myproject (or poetry init in an existing folder) 3. poetry add requests (example package) 4. poetry run python main.py (runs inside the project's environment) Optional: poetry shell to activate the env for multiple commands. Use Poetry the moment you add even one third-party package, or whenever you want classmates to reproduce your setup.

2.4.5 Learn just enough Git and GitHub

On a gain-to-pain ratio, learning Git and Github is a winnder. A five-command subset changes your experience: • git init (start a repository) • git add -A and git commit -m "message" (save a snapshot) • git log (see history) • git restore –source – (recover a prior version) Optional: git branch -c feature-x (try ideas safely). Benefits you feel immediately: easy undo, fearless experiments, off-device backup via GitHub, and simple sharing. Spend 40 minutes learning git and Github. Great investment. Free for most plans.

2.4.6 Use Study and Learn mode to close gaps just-in-time

Modern study modes in large language models can turn friction into short, targeted lessons. When you run into problems, go into study and learn mode and tell it what issues you are encountering. Explain that you are a beginner and are probably missing some piece of "obvious" background. Ask it to figure out the pieces of the puzzle you are missing and design a curriculum. Works really well![2]

[1] Ironically, Python's dependency hell arose from its massive success and longevity. As an old language, Python's ecosystem grew rapidly, with thousands of packages on PyPI. Inconsistent versioning, lack of strict dependency management in early tools like pip, and backward-incompatible updates created conflicts, making dependency resolution complex and error-prone.

[2] Befriending a human expert can often be useful, though they can sometimes become frustrated at how little you know. If they suggest Docker, unfriend them.

2.5 Final counsel

LLMs are fantastic at programming. For beginners, however, it is important to manage expectations and to recognize that enthusiasts and LLMs often forget that there are very "basic" things that are absolutely critical, not that difficult to understand, but that you don't yet know. Keep projects small; treat line count as a risk indicator. Avoid frameworks and scaffolding until you have a couple of working wins. Defer Docker until you know why you need it. Prefer low-friction environments (browser JS, Go, or Python with Poetry). Learn just enough Git to feel safe experimenting. Vibe coding builds confidence when you reduce operational complexity and let the LLM help with what it already does best: producing small, working programs you can actually run.