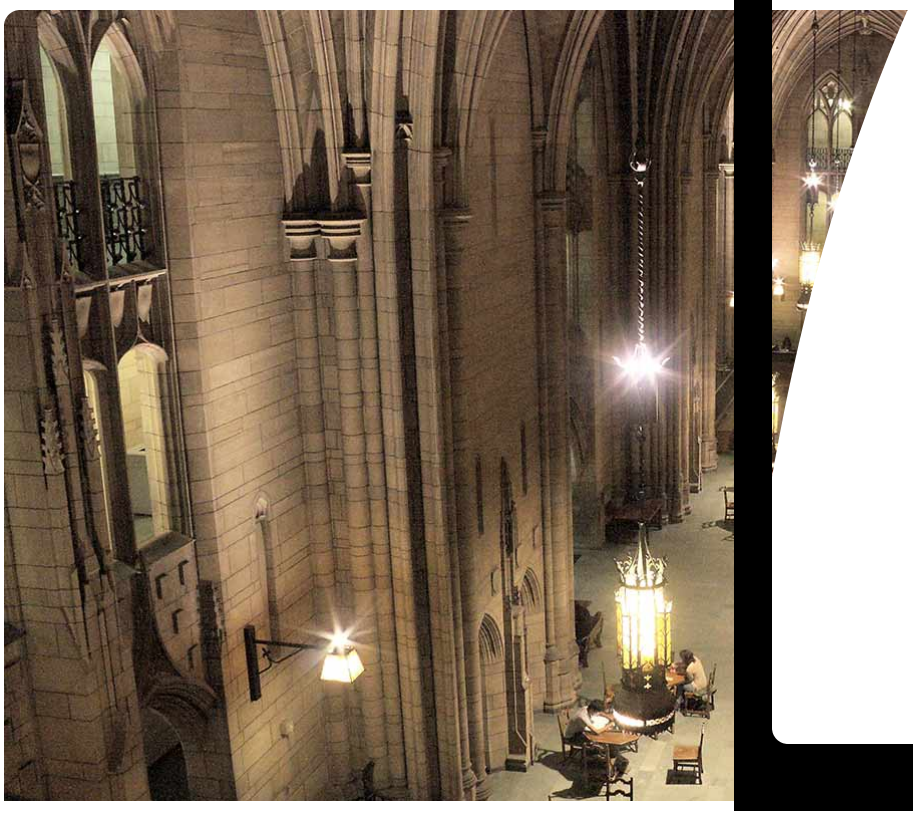
Using Generative AI to Buil Generative AI

Adding AI Chat to the Python Tutor ode Visualizer: A Co

By Philip Guo

February 25, 2025 • 23 minute read

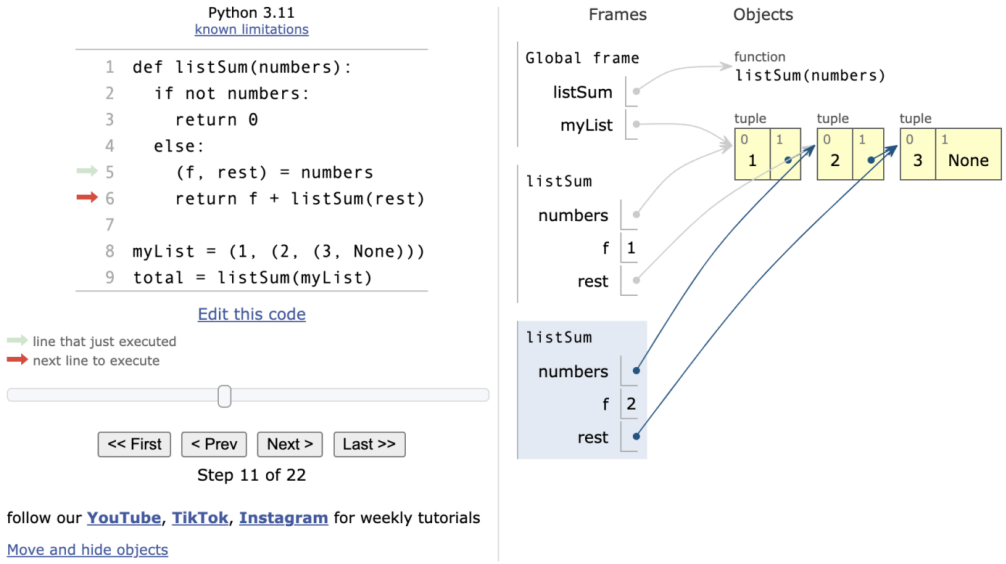


On May 8, O'Reilly Media will be hosting **Coding with AI: The End of Software Development as We Know It***—a live virtual tech conference spotlighting how AI is already supercharging developers, boosting productivity, and providing real value to their organizations. If you're in the trenches building tomorrow's development practices today and interested in speaking at the event, we'd love to hear from you by March 12. You can find more information and our call for presentations here. Just want to attend? Register for free here.*

Hi, I am a professor of cognitive science and design at UC San Diego, and I recently wrote posts on Radar about my experiences coding with and speaking to generative AI tools like ChatGPT. In this post I want to talk about using generative AI to extend one of my academic software projects—the Python Tutor tool for learning programming—with an AI chat tutor. We often hear about GenAI being used in large-scale commercial settings, but we don't hear nearly as much about smaller-scale not-for-profit projects. Thus, this post serves as a case study of adding generative AI into a personal project where I didn't have much time, resources, or expertise at my disposal. Working on this project got me really excited about being here at this moment right as powerful GenAI tools are starting to become more accessible to nonexperts like myself.

For some context, over the past 15 years I've been operating Python Tutor (https://pythontutor.com/), a free online tool that tens of millions of people around the world have used to write, run, and visually debug their code (first in Python and now also in Java, C, C++, and JavaScript). Python Tutor is mainly used by students to understand and debug their homework assignment code step-by-step by seeing its call stack and data structures. Think of it as a virtual instructor who draws diagrams to show runtime state on a whiteboard. It's best suited for small pieces of self-contained code that students commonly encounter in computer science classes or online coding tutorials.

Here's an example of using Python Tutor to step through a recursive function that builds up a linked list of Python tuples. At the current step, the visualization shows two recursive calls to the `listSum` function and various pointers to list nodes. You can move the slider forward and backward to see how this code runs step-by-step:

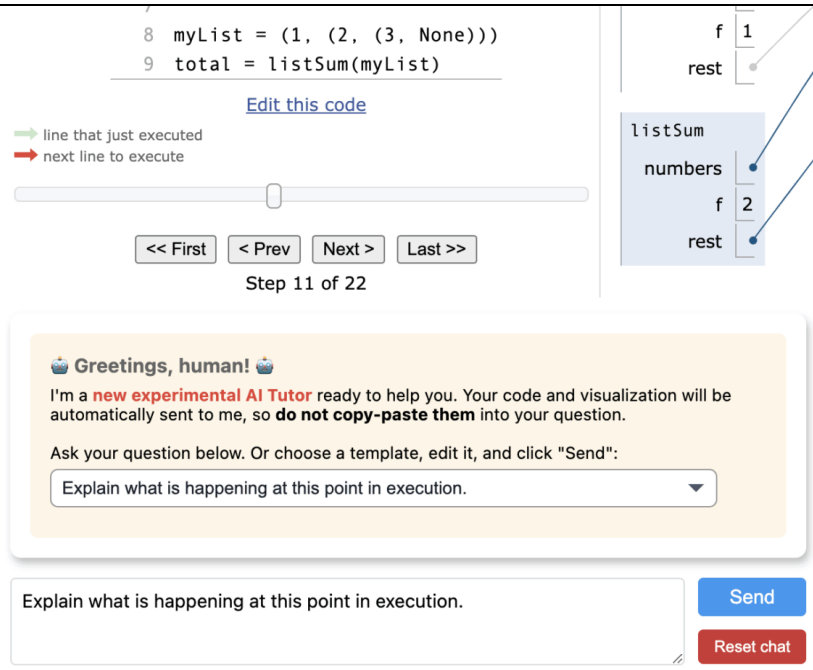


AI Chat for Python Tutor's Code Visualizer

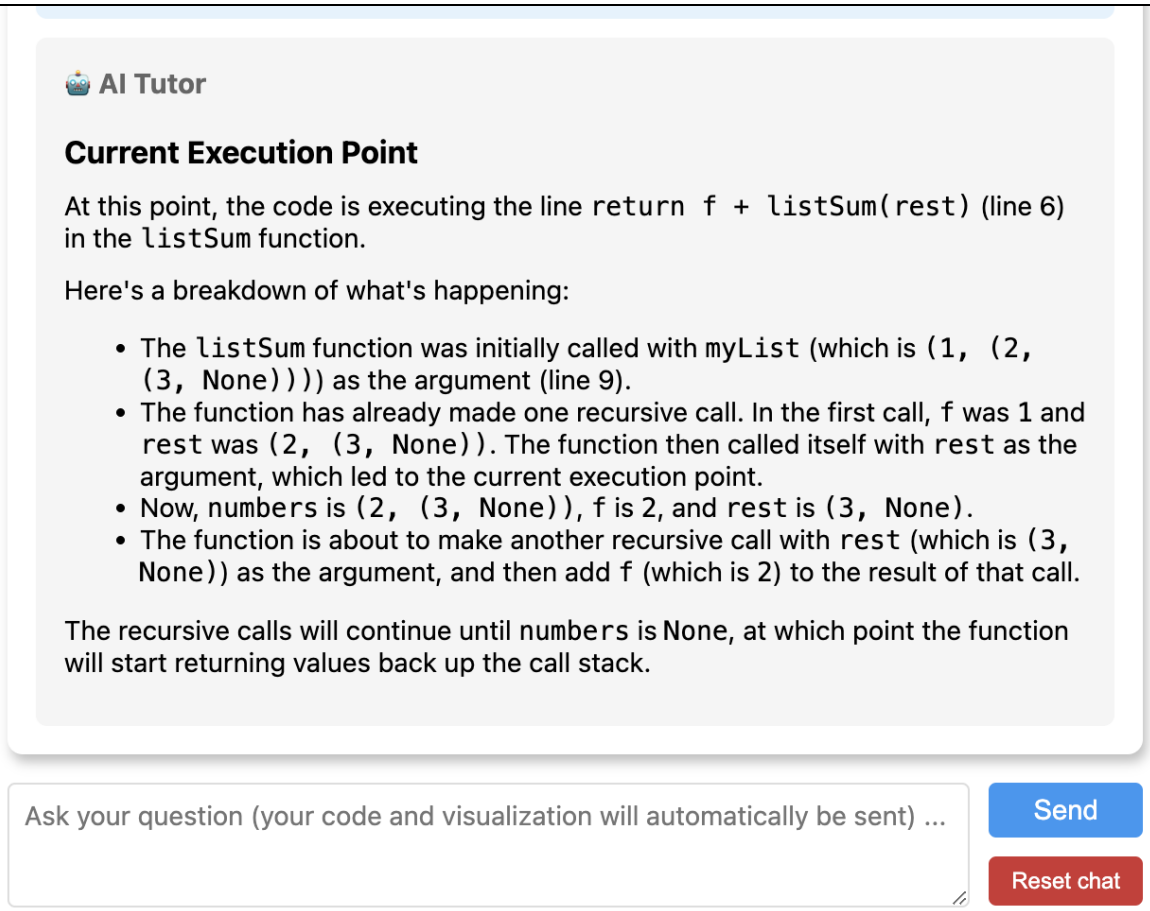
Way back in 2009 when I was a grad student, I envisioned creating Python Tutor to be an automated tutor that could help students with programming questions (which is why I chose that project name). But the problem was that AI wasn't nearly good enough back then to emulate a human tutor. Some AI researchers were publishing papers in the field of intelligent tutoring systems, but there were no widely accessible software libraries or APIs that could be used to make an AI tutor. So instead I spent all those years working on a versatile code visualizer that could be *used* by human tutors to explain code execution.

Fast-forward 15 years to 2024, and generative AI tools like ChatGPT, Claude, and many others based on LLMs (large language models) are now really good at holding human-level conversations, especially about technical topics related to programming. In particular, they're great at generating and explaining small pieces of self-contained code (e.g., under 100 lines), which is exactly the target use case for Python Tutor. So with this technology in hand, I used these LLMs to add AI-based chat to Python Tutor. Here's a quick demo of what it does.

First I designed the user interface to be as simple as possible. It's just a chat box below the user's code and visualization:

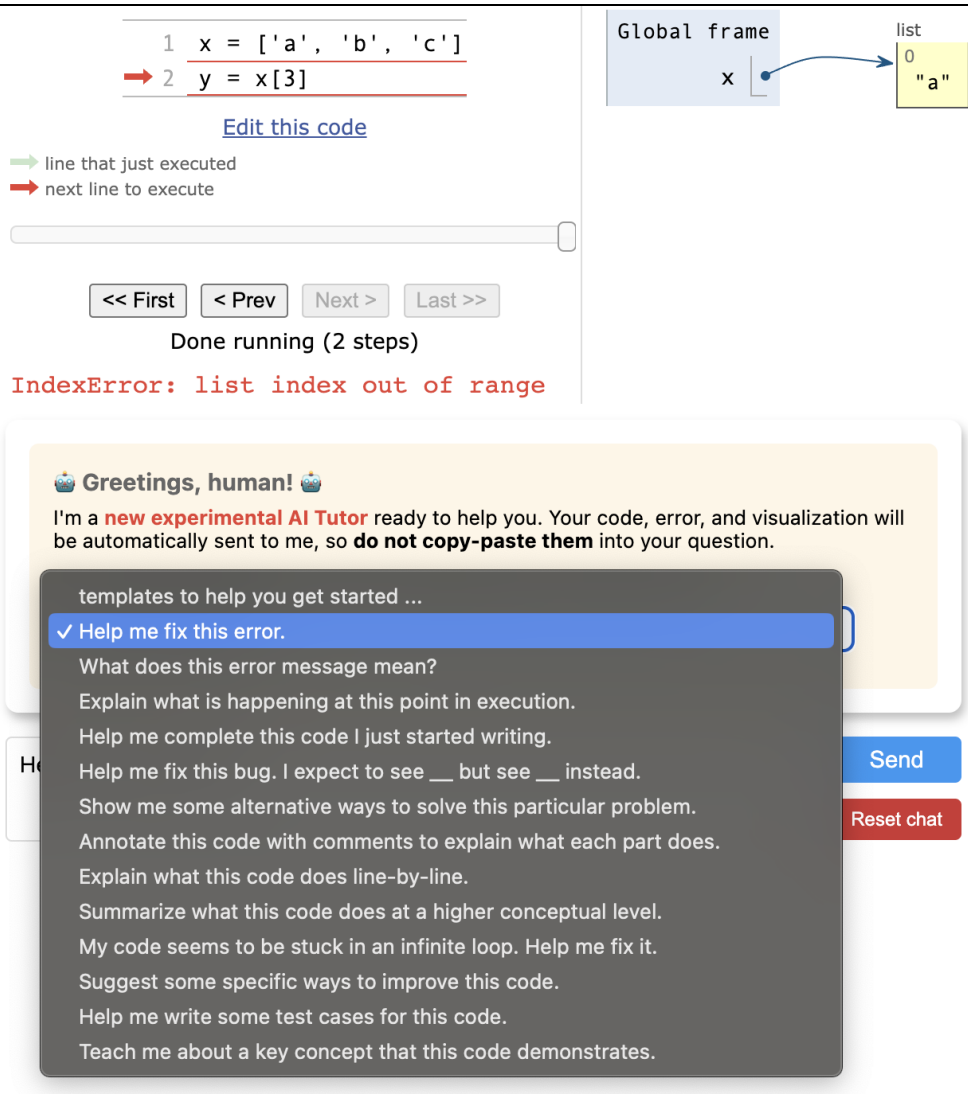


There's a dropdown menu of templates to get you started, but you can type in any question you want. When you click "Send," the AI tutor will send your code, current visualization state (e.g., call stack and data structures), terminal text output, and question to an LLM, which will respond here with something like:

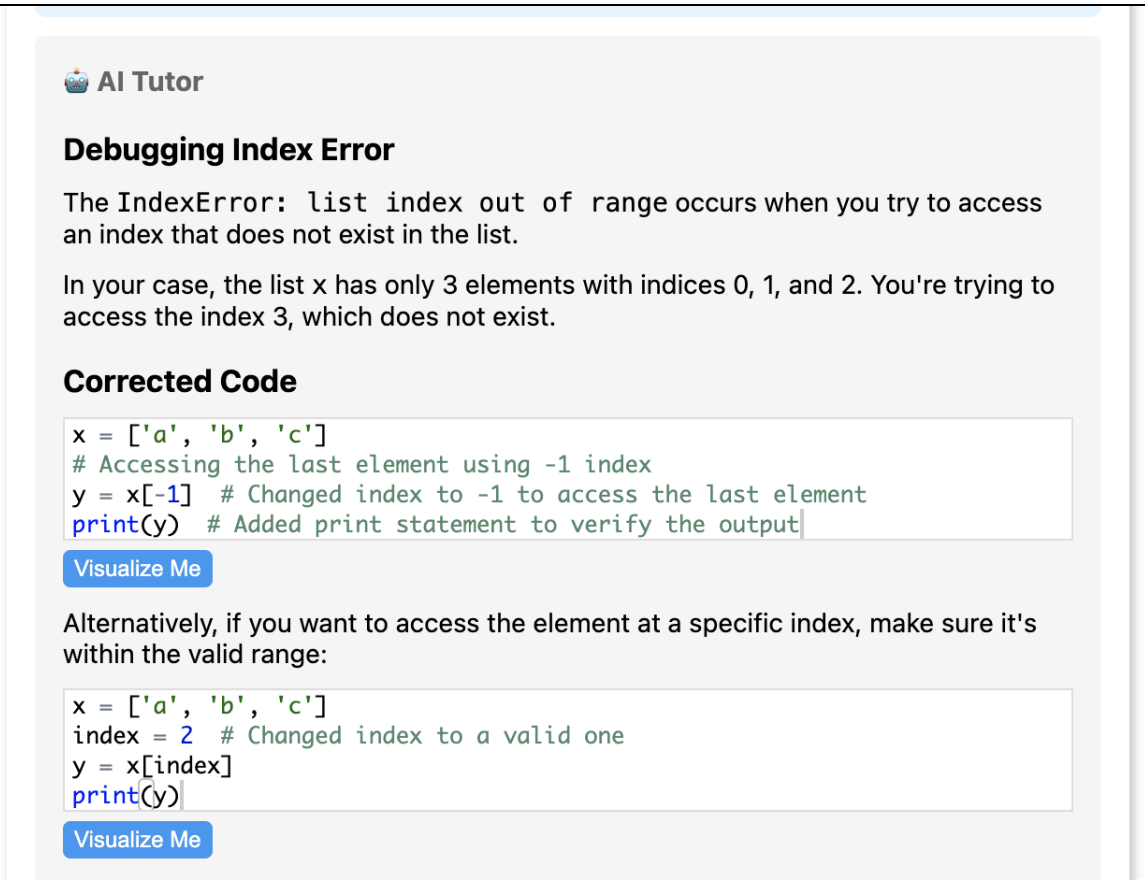


Note how the LLM can "see" your current code and visualization, so it can explain to you what's going on here. This emulates what an expert human tutor would say. You can then continue chatting back-and-forth like you would with a human.

In addition to explaining code, another common use case for this AI tutor is helping students get unstuck when they encounter a compiler or runtime error, which can be very frustrating for beginners. Here's an index out-of-bounds error in Python:

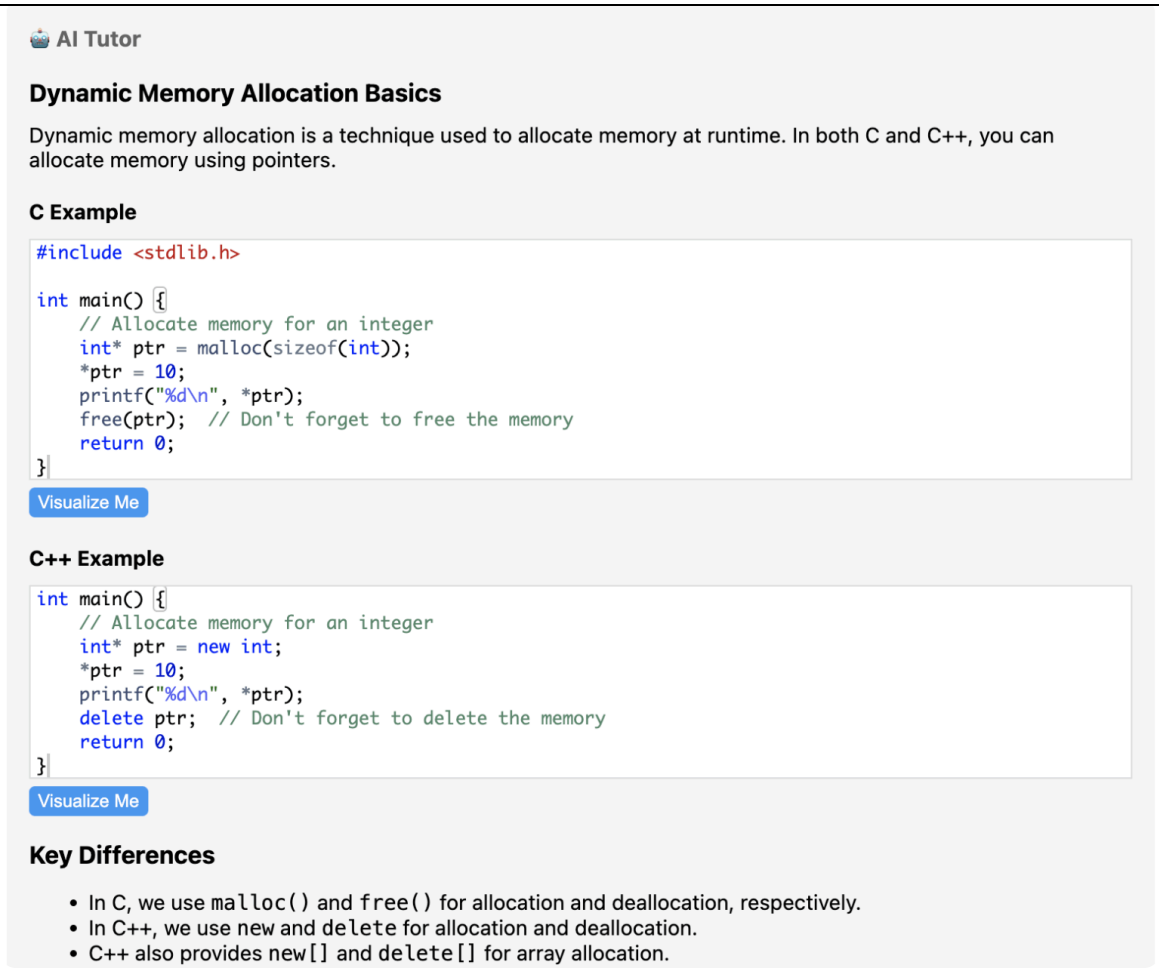


Whenever there's an error, the tool automatically populates your chat box with "Help me fix this error," but you can select a different question from the dropdown (shown expanded above). When you hit "Send" here, the AI tutor responds with something like:



Note that when the AI generates code examples, there's a "Visualize Me" button underneath each one so that you can directly visualize it in Python Tutor. This allows you to visually step through its execution and ask the AI follow-up questions about it.

Besides asking specific questions about your code, you can also ask general programming questions or even career-related questions like how to prepare for a technical coding interview. For instance:



... and it will generate code examples that you can visualize without leaving the Python Tutor website.

Benefits over Directly Using ChatGPT

The obvious question here is: What are the benefits of using AI chat within Python Tutor rather than pasting your code and question into ChatGPT? I think there are a few main benefits, especially for Python Tutor's target audience of beginners who are just starting to learn to code:

1) **Convenience** – Millions of students are already writing, compiling, running, and visually debugging code within Python Tutor, so it feels very natural for them to also ask questions without leaving the site. If instead they need to select their code from a text editor or IDE, copy it into another site like ChatGPT, and then maybe also copy their error message, terminal output, and describe what is going on at runtime (e.g., values of data structures), that's way more cumbersome of a user experience. Some modern IDEs do have AI chat built in, but those require expertise to set up since they're meant for professional software developers. In contrast, the main appeal of Python Tutor for beginners has always been its ease of access: Anyone can

2) **Beginner-friendly LLM prompts** – Next, even if someone were to go through the trouble of copy-pasting their code, error message, terminal output, and runtime state into ChatGPT, I've found that beginners aren't good at coming up with prompts (i.e., written instructions) that direct LLMs to produce easily understandable responses. Python Tutor's AI chat addresses this problem by augmenting chats with a system prompt like the following to emphasize directness, conciseness, and beginner-friendliness:

> You are an expert programming teacher and I am a student asking you for help with `${LANGUAGE}`.
> - Be concise and direct. Keep your response under 300 words if possible.
> - Write at the level that a beginner student in an introductory programming class can understand.
> - If you need to edit my code, make as few changes as needed and preserve as much of my original code as possible. Add code comments to explain your changes.
> - Any code you write should be self-contained and runnable without importing external libraries.
> - Use GitHub Flavored Markdown.

It also formats the user's code, error message, relevant line numbers, and runtime state in a well-structured way for LLMs to ingest. Lastly, it provides a dropdown menu of common questions and requests like "What does this error message mean?" and "Explain what this code does line-by-line." so beginners can start crafting a question right away without staring at a blank chat box. All of this behind-the-scenes prompt templating helps users to avoid common problems with directly using ChatGPT, such as it generating explanations that are too wordy, jargon-filled, and overwhelming for beginners.

3) **Running your code instead of just "looking" at it** – Lastly, if you paste your code and question into ChatGPT, it "inspects" your code by reading over it like a human tutor would do. But it doesn't actually run your code so it doesn't know what function calls, variables, and data structures really exist during execution. While modern LLMs are good at guessing what code does by "looking" at it, there's no substitute for running code on a real computer. In contrast, Python Tutor runs your code so that when you ask AI chat about what's going on, it sends the real values of the call stack, data structures, and terminal output to the LLM, which again hopefully results in more helpful responses.

Now that you've seen how Python Tutor's AI chat works, you might be wondering: Did I use generative AI to help me build this GenAI feature? Yes and no. GenAI helped me most when I was getting started, but as I got deeper in I found less of a use for it.

Using Generative AI to Create a Mock-Up User Interface

My approach was to first build a stand-alone web-based LLM chat app and later integrate it into Python Tutor's codebase. In November 2024, I bought a Claude Pro subscription since I heard good buzz about its code generation capabilities. I began by working with Claude to generate a mock-up user interface for an LLM chat app with familiar features like a user input box, text bubbles for both the LLM and human user's chats, HTML formatting with Markdown, syntax-highlighted code blocks, and streaming the LLM's response incrementally rather than making the user wait until it finished. None of this was innovative—it's what everyone expects from using an LLM chat interface like ChatGPT.

I liked working with Claude to build this mock-up because it generated live runnable versions of HTML, CSS, and JavaScript code so I could interact with it in the browser without copying the code into my own project. (Simon Willison wrote a great post on this Claude Artifacts feature.) However, the main downside is that whenever I request even a small code tweak, it would take up to a minute or so to regenerate all the project code (and sometimes annoyingly leave parts as incomplete [...] segments, which made the code not run). If I had instead used an AI-powered IDE like Cursor or Windsurf, then I would've been able to ask for instant incremental edits. But I didn't want to bother setting up more complex tooling, and Claude was good enough for getting my web frontend started.

A False Start by Locally Hosting an LLM

Now onto the backend. I originally started this project after playing with Ollama on my laptop, which is an app that allowed me to run LLMs locally for free without having to pay a cloud provider. A few months earlier (September 2024) Llama 3.2 had come out, which featured smaller models like 1B and 3B (1 and 3 billion parameters, respectively). These are much less powerful than state-of-the-art models, which are 100 to 1,000 times bigger at the time of writing. I had no hope of running larger models locally (e.g., Llama 405B), but these smaller 1B and 3B models ran fine on my laptop so they seemed promising.

Note that the last time I tried running an LLM locally was GPT-2 (yes, 2!) back in 2021, and it was TERRIBLE—a pain to set up by installing a bunch of Python dependencies, super slow to run, and producing nonsensical results. So for years I didn't think it was feasible to self-host my own LLM for Python Tutor. And I didn't want to pay to use a cloud API like ChatGPT or Claude since

But now, three years later, the combination of smaller LLMs and Ollama's ease-of-use convinced me that the time was right for me to self-host my own LLM for Python Tutor. So I used Claude and ChatGPT to help me write some boilerplate code to connect my prototype web chat frontend with a Node.js backend that called Ollama to run Llama 1B/3B locally. Once I got that demo working on my laptop, my goal was to host it on a few university Linux servers that I had access to.

But barely one week in, I got bad news that ended up being a huge blessing in disguise. Our university IT folks told me that I wouldn't be able to access the few Linux servers with enough CPUs and RAM needed to run Ollama, so I had to scrap my initial plans for self-hosting. Note that the kind of low-cost server I wanted to deploy on didn't have GPUs, so they ran Ollama much more slowly on their CPUs. But in my initial tests a small model like Llama 3.2 3B still ran ok for a few concurrent requests, producing a response within 45 seconds for up to 4 concurrent users. This isn't "good" by any measure, but it's the best I could do without paying for a cloud LLM API, which I was afraid to do given Python Tutor's sizable userbase and tiny budget. I figured if I had, say 4 replica servers, then I could serve up to 16 concurrent users within 45 seconds, or maybe 8 concurrents within 20 seconds (rough estimates). That wouldn't be the best user experience, but again Python Tutor is free for users, so their expectations can't be sky-high. My plan was to write my own load-balancing code to direct incoming requests to the lowest-load server and queuing code so if there were more concurrent users trying to connect than a server had capacity for, it would queue them up to avoid crashes. Then I would need to write all the sysadmin/DevOps code to monitor these servers, keep them up-to-date, and reboot if they failed. This was all a daunting prospect to code up and test robustly, especially because I'm not a professional software developer. But to my relief, now I didn't have to do any of that grind since the university server plan was a no-go.

Switching to the OpenRouter Cloud API

So what did I end up using instead? Serendipitously, around this time someone pointed me to OpenRouter, which is an API that allows me to write code once and access a variety of paid LLMs by simply changing the LLM name in a configuration string. I signed up, got an API key, and started making queries to Llama 3B in the cloud within minutes. I was shocked by how easy this code was to set up! So I quickly wrapped it in a server backend that streams the LLM's response text in real time to my frontend using SSE (server-sent events), which displays it in the mock-up chat UI. Here's the essence of my Python backend code:

```
client = openai.OpenAI(
    base_url="https://openrouter.ai/api/v1",
    api_key=<your API key>
)
completion = client.chat.completions.create(
    model=<name of LLM, such as Llama 3.2 3B>,
    messages=<your query to the LLM>,
    stream=True
)
for chunk in completion:
    text = chunk.choices[0].delta.content
    <stream text to web frontend using Server-Sent Events>
```

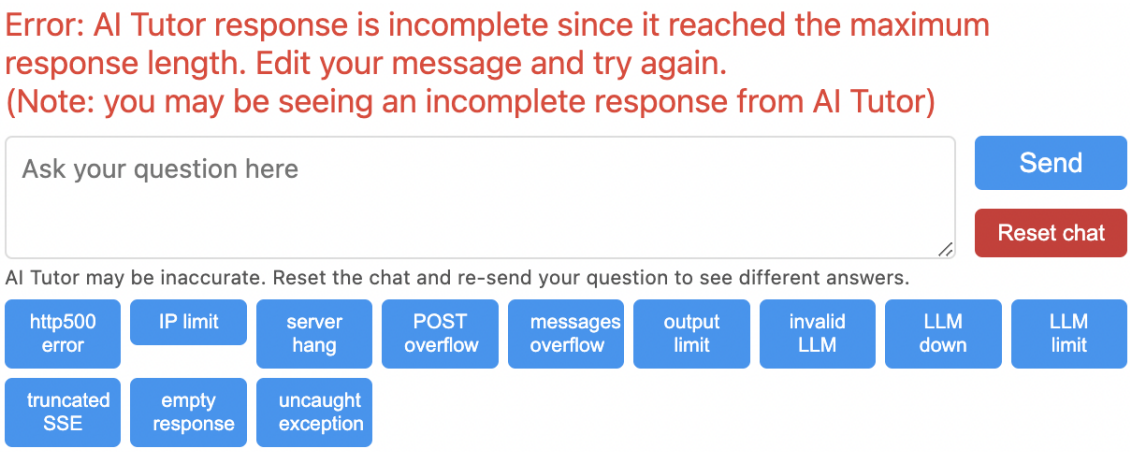
OpenRouter does cost money, but I was willing to give it a shot since the prices for Llama 3B looked more reasonable than state-of-the-art models like ChatGPT or Claude. At the time of writing, 3B is about $0.04 USD per million tokens, and a state-of-the-art LLM costs up to 500x as much (ChatGPT-4o is $20 and Claude 3.7 Sonnet is $18). I would be scared to use ChatGPT or Claude at those prices, but I felt comfortable with the much cheaper Llama 3B. What also gave me comfort was knowing I wouldn't wake up with a giant bill if there were a sudden spike in usage; OpenRouter lets me put in a fixed amount of money, and if that runs out my API calls simply fail rather than charging my credit card more.

For some extra peace of mind I implemented my own rate limits: 1) Each user's input and total chat conversations are limited to a certain length to keep costs under control (and to reduce hallucinations since smaller LLMs tend to go "off the rails" as conversations grow longer); 2) Each user can send only one chat per minute, which again prevents overuse. Hopefully this isn't a big problem for Python Tutor users since they need at least a minute to read the LLM's response, try out suggested code fixes, then ask a follow-up question.

Using OpenRouter's cloud API rather than self-hosting on my university's servers turned out to be so much better since: 1) Python Tutor users can get responses within only a few seconds rather than waiting 30-45 seconds; 2) I didn't need to do any sysadmin/DevOps work to maintain my servers, or to write my own load balancing or queuing code to interface with Ollama; 3) I can easily try different LLMs by changing a configuration string.

GenAI as a Thought Partner and On-Demand Teacher

partner by having GenAI help me come up with edge cases that I hadn't originally considered. I then created a debugging UI panel with a dozen buttons below the chat box that I could press to simulate specific errors in order to test how well my app handled those cases:



After getting my stand-alone LLM chat app working robustly on error cases, it was time to integrate it into the main Python Tutor codebase. This process took a lot of time and elbow grease, but it was straightforward since I made sure to have my stand-alone app use the same versions of older JavaScript libraries that Python Tutor was using. This meant that at the start of my project I had to instruct Claude to generate mock-up frontend code using those older libraries; otherwise by default it would use modern JavaScript frameworks like React or Svelte that would not integrate well with Python Tutor, which is written using 2010-era jQuery and friends.

At this point I found myself not really using generative AI day-to-day since I was working within the comfort zone of my own codebase. GenAI was useful at the start to help me figure out the "unknown unknowns." But now that the problem was well-scoped I felt much more comfortable writing every line of code myself. My daily grind from this point onward involved a lot of UI/UX polishing to make a smooth user experience. And I found it easier to directly write code rather than think about how to instruct GenAI to code it for me. Also, I wanted to understand every line of code that went into my codebase since I knew that every line would need to be maintained perhaps years into the future. So even if I could have used GenAI to code faster in the short term, that may come back to haunt me later in the form of subtle bugs that arise because I didn't fully understand the implications of AI-generated code.

That said, I still found GenAI useful as a replacement for Google or Stack Overflow sorts of questions like "How do I write X in modern JavaScript?" It's an incredible resource for learning technical details on the fly, and I sometimes adapted the example code in AI responses into my

Finishing Touches and Launching

I wanted to launch by the new year, so as November rolled into December I was making steady progress getting the user experience more polished. There were a million little details to work through, but that's the case with any nontrivial software project. I didn't have the resources to evaluate how well smaller LLMs perform on real questions that users might ask on the Python Tutor website, but from informal testing I was dismayed (but not surprised) at how often the 1B and 3B models produced incorrect explanations. I tried upgrading to a Llama 8B model, and it was still not amazing. I held out hope that tweaking my system prompt would improve performance. I didn't spend a ton of time on it, but my initial impression was that no amount of tweaking could make up for the fact that a smaller model is just less capable—like a dog brain compared to a human brain.

Fortunately in late December—only two weeks before launch—Meta released a new Llama 3.3 70B model. I was running out of time, so I took the easy way out and switched my OpenRouter configuration to use it. My AI Tutor's responses instantly got better and made fewer mistakes, even with my original system prompt. I was nervous about the 10x price increase from 3B to 70B ($0.04 to $0.42 per million tokens) but gave it a shot anyhow.

Parting Thoughts and Lessons Learned

Fast-forward to the present. It's been two months since launch, and costs are reasonable so far. With my strict rate limits in place Python Tutor users are making around 2,000 LLM queries per day, which costs less than a dollar each day using Llama 3.3 70B. And I'm hopeful that I can switch to more powerful models as their prices drop over time. In sum, it's super satisfying to see this AI chat feature live on the site after dreaming about it for almost 15 years since I first created Python Tutor long ago. I love how cloud APIs and low-cost LLMs have made generative AI accessible to nonexperts like myself.

Here are some takeaways for those who want to play with GenAI in their personal apps:

- I highly recommend using a cloud API provider like OpenRouter rather than self-hosting LLMs on your own VMs or (even worse) buying a physical machine with GPUs. It's infinitely cheaper and more convenient to use the cloud here, especially for personal-scale projects. Even with thousands of queries per day, Python Tutor's AI costs are tiny compared to paying for VMs or physical machines.
- Waiting helped! It's good to not be on the bleeding edge all the time. If I had attempted to do this project in 2021 during the early days of the OpenAI GPT-3 API like early adopters did, I

know how to help me code using those APIs since the necessary docs weren't available for them to train on. By simply waiting a few years, I was able to work with high-quality stable cloud APIs and get useful technical help from Claude and ChatGPT while coding my app.

- It's fun to play with LLM APIs rather than using the web interfaces like most people do. By writing code with these APIs you can intuitively "feel" what works well and what doesn't. And since these are ordinary web APIs, you can integrate them into projects written in any programming language that your project is already using.
- I've found that a short, direct, and simple system prompt with a larger LLM will beat elaborate system prompts with a smaller LLM. Shorter system prompts also mean that each query costs you less money (since they must be included in the query).
- Don't worry about evaluating output quality if you don't have resources to do so. Come up with a few handcrafted tests and run them as you're developing—in my case it was tricky pieces of code that I wanted to ask Python Tutor's AI chat to help me fix. If you stress too much about optimizing LLM performance, then you'll never ship anything! And if you find yourself yearning for better quality, upgrade to a larger LLM first rather than tediously tweaking your prompt.
- It's very hard to estimate how much running an LLM will cost in production since costs are calculated per million input/output tokens, which isn't intuitive to reason about. The best way to estimate is to run some test queries, get a sense of how wordy the LLM's responses are, then look at your account dashboard to see how much each query cost you. For instance, does a typical query cost 1/10 cent, 1 cent, or multiple cents? No way to find out unless you try. My hunch is that it probably costs less than you imagine, and you can always implement rate limiting or switch to a lower-cost model later if cost becomes a concern.
- Related to above, if you're making a prototype or something where only a small number of people will use it at first, then definitely use the best state-of-the-art LLM to show off the most impressive results. Price doesn't matter much since you won't be issuing that many queries. But if your app has a fair number of users like Python Tutor does, then pick a smaller model that still performs well for its price. For me it seems like Llama 3.3 70B strikes that balance in early 2025. But as new models come onto the scene, I'll reevaluate those price-to-performance trade-offs.

Acknowledgments: This material is based upon work supported by the National Science Foundation under Grant No. NSF IIS-1845900.

Post topics: AI & ML • Artificial Intelligence

Follow us

About O'Reilly

Teach/Write/Train

Careers

O'Reilly News

Media Coverage

Community Partners

Affiliate Program

Submit an RFP

Diversity

Content Sponsorship

Support

Contact Us

Newsletters

Privacy Policy

AI Policy

International

Australia & New Zealand

Japan

Download the O'Reilly App

Take O'Reilly with you and learn anywhere, anytime on your phone and tablet.

Download on the App Store

GET IT ON Google Play

Watch on Your Big Screen

O'REILLY®