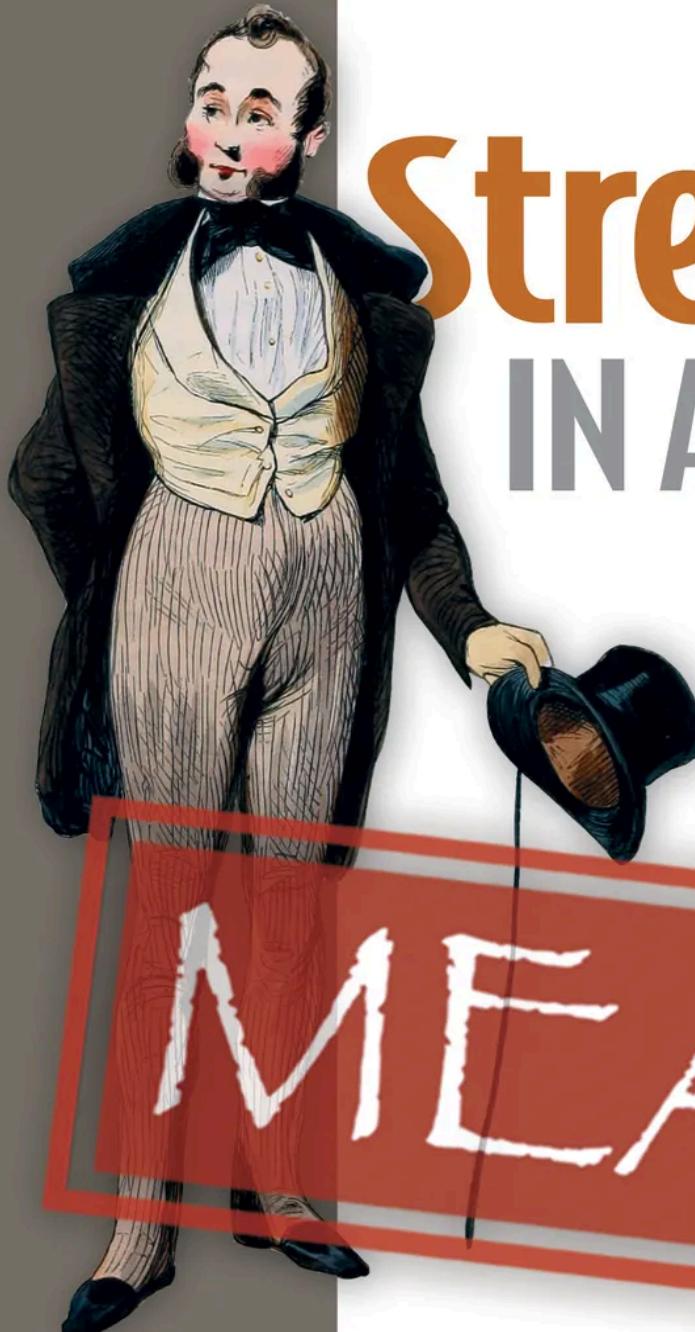


Pure Python web apps in minutes



Streamlit IN ACTION

Aneev D. Kochakadan



MEAP



MEAP Edition
Manning Early Access Program

Streamlit in Action
Pure Python web apps in minutes
Version 6

Copyright 2025 Manning Publications

For more information on this and other Manning titles go to manning.com.

welcome

Thank you for purchasing the MEAP version of *Streamlit In Action*!

If you're like me, you were probably drawn to Python because of its intuitive, yet powerful, syntax that reads almost like English. It's one of the easiest languages to pick up, making it a favorite among beginners and experienced programmers alike. However, I've always found one major flaw with it: creating something visual and accessible on the web in Python can be incredibly challenging.

Traditionally, you'd have to juggle HTML, CSS, and JavaScript or a framework like React for the frontend while using Python for the backend—a complex and time-consuming process that not everyone has the bandwidth to learn or desire to master.

Streamlit bridges this gap beautifully. It's built with the same ease and intuitiveness that makes Python so beloved. What might otherwise take weeks to develop, painstakingly wiring up a frontend and backend written in different technologies, can often be accomplished in hours or sometimes even *minutes* with Streamlit.

Streamlit has been a game-changer for many people, empowering data scientists, analysts, and even semi-technical folks with some Python knowledge to deliver business value quickly. For others such as software engineers, it can be invaluable as a rapid prototyping tool. Wherever you are on the skill spectrum, as long as you have some knowledge of Python, this book will equip you to become a full-fledged application developer, wielding Streamlit in powerful and creative ways.

Streamlit in Action is packed with practical projects. It teaches topics in an organic way, similar to how you would encounter and learn them in the course of working on real problems. In fact, I've worked hard to make this book a condensed version of years of experience that you would otherwise have had to acquire independently. Here's what you can expect:

Part 1 covers the basics of Streamlit, immediately arming you to be effective with Streamlit. Even if you *only* complete this part, this book will leave you in a position to start creating full-fledged apps.

Part 2 deepens your experience, guiding you through building several complex applications.

Part 3 dives into large language models (LLMs) and AI applications, a niche where Streamlit truly shines.

Part 4 focuses on all of the operational aspects surrounding Streamlit, like deployment to various environments and testing.

Part 5 delves into advanced development, including how to extend Streamlit with your own custom components.

Your feedback is crucial in shaping the best book possible. Please share your thoughts and questions in the [liveBook Discussion forum](#) where they'll help me understand if my approach is working and re-adjust it if I need to. I hope you have as much fun reading Streamlit in Action as much as I did in writing it, and that it helps you unlock your potential as a developer.

—Aneev Kochakadan

brief contents

PART 1: HITTING THE GROUND RUNNING

- 1 Introduction to Streamlit*
- 2 Getting started with Streamlit*
- 3 Taking an app from concept to code*
- 4 Streamlit's execution model*
- 5 Sharing your apps with the world*

PART 2: DIVING INTO REAL STREAMLIT DATA APPS

- 6 A dashboard fit for a CEO*
- 7 The CEO strikes back: Supercharging our dashboard*
- 8 Building a CRUD app with Streamlit*

PART 3: STREAMLIT FOR LLMS

- 9 An AI-powered trivia game*
- 10 A customer support chatbot with LangGraph and Streamlit*

PART 4: TESTING AND DEPLOYING STREAMLIT APPS

- 11 Testing Streamlit apps*
 - 12 Packaging and deploying Streamlit apps*
- Appendix A. Installing Python and Streamlit*

1 Introduction to Streamlit

This chapter covers

- Why you'd want to build web apps
- What exactly Streamlit is, and why it's so popular
- How Streamlit's ease-of-use, LLM-friendliness, and other factors make it popular
- How Streamlit is different from other similar technologies
- What you can (and can't) build with Streamlit

Welcome to the exciting realm of Streamlit! By picking up this book, you've joined the ranks of thousands of developers who have discovered Streamlit over the past several years. These developers have become enamored with what Streamlit makes possible: web apps coded entirely in Python in mere minutes!

Take a second to think about why you were drawn to this book. Maybe you have an idea buzzing in your head for an application that will save your coworkers hours of mundane, easy-to-automate tasks, and you want the fastest way to turn it into reality. Maybe you're aiming to land a job in tech and want to close a gap in your skillset by adding frontend development. Perhaps you're a data analyst or scientist wanting to present your findings in interactive dashboards for higher-ups. Or you might be a software engineer needing a quick way to prototype your apps. Perhaps you simply heard the buzz around Streamlit and AI and were curious.

Whatever your story, Streamlit can bring your ideas to life quickly and easily. This book will be your guide, taking you step-by-step through the process of creating powerful, interactive web applications. You'll learn how to harness the simplicity and elegance of Python to build and deploy apps that can wow your audience, solve real problems, and advance your career.

Experience is the best teacher, so you'll learn Streamlit through real-world projects. By the end of this book, you'll have built a versatile portfolio, ranging from an interactive mortgage calculator to a chatbot powered by generative AI—all in record time!

While each chapter will immerse you deeper into Streamlit's capabilities, you'll also learn the overarching process of developing apps, including how to think about UI design and how to organize your code for maintainability.

Whether you're a seasoned industry veteran or a complete beginner, I am convinced that you'll find value in this book.

Eager to dive in? Then let's get started with the basics.

1.1 Building web apps

Before we get to Streamlit itself, let's talk about web apps and why you would want to build one, which is what Streamlit is for.

A *graphical app* is an application with a *graphical user interface (GUI)* that you can interact with by clicking a mouse or touching on-screen controls. Contrast this with a *command-line app*, which has a *command-line user interface* where you type text inputs into a terminal and get the results back in the same terminal. You've likely written a few of the latter using Python.

A *web application* (or *web app*) is a graphical app that you access through a web browser, such as when you access Gmail or Netflix using the Google Chrome browser, as opposed to a *desktop app*, which is a program that runs directly on your computer (e.g., Photoshop or Notepad running on your PC), or a *native mobile app*, which runs directly on your mobile phone (e.g., Uber or when you access Gmail through the iPhone Gmail app).

Streamlit helps you build web apps, not command-line, desktop or native mobile apps. But why would you want to do that?

1.1.1 Why build a graphical app in the first place?

If you've learned Python, chances are that you're comfortable with creating command-line programs. These are the easiest types of apps to create, and Python provides native support for them.

The trouble is, outside of techies, no one likes to use command-line programs for anything significant. Graphical apps are much less intimidating and give users a more intuitive experience.

This is true generally, but also in the workplace. If you want to automate something at work for a non-technical audience and expect people to actually use what you've built, you'll need to make a graphical interface to go with it. You might have created the perfect solution to a genuine problem that cuts the effort required to accomplish something from hours to minutes, but the minute you tell people they're going to have to open up a terminal and type commands into it, you'll have lost them.

1.1.2 Why build apps for the web?

Graphical apps provide a clear usability advantage, but why build a web app rather than a desktop app or a native mobile app?

Over the past two decades, web apps have become increasingly popular for a whole host of reasons. Here are just a couple:

- Web apps can run on pretty much any device that has a web browser. This means you can code your app once and have users be able to run your program on their computer, tablet or phone with no extra effort.
- Users don't have to install or update web apps manually; by visiting the right web address, they are always using the latest version.

Web apps are especially popular within companies. Most companies today have an *intranet*, which is an internal network of web pages only open to their employees. Since all employees are used to accessing the intranet, it often makes a lot of sense for a company to host their internal programs on this intranet as web apps rather than have people go through the trouble of installing desktop or mobile apps.

Of course, there are certainly valid reasons to build desktop or mobile apps instead, such as when performance or offline access is critical. However, for many use cases, the benefits of web apps make them the preferred choice for both developers and users.

1.1.3 What do you need to build a web app?

Hopefully you've bought into the idea of creating web apps enough to turn to the next question: how?

Generally speaking (and simplifying somewhat), a web app has two big parts: a *frontend* and a *backend*. The frontend contains the onscreen things that people interact with, such as buttons, textboxes, menus, and what have you. The backend has the actual logic that fulfills the app's purpose, such as crunching numbers or looking something up in a database.

The skills and languages required to create the backend are quite different from those required for the frontend. You can write your backend logic in Python (which this book assumes you're familiar with), primarily focusing on getting your business logic to work.

Writing the frontend, on the other hand, involves focusing on the user experience where there are fewer "right" answers. And importantly, it has traditionally required familiarity with a different set of languages. Learning these languages requires at least as much, and probably more, effort as it took to learn Python from scratch.

This problem has stymied many busy Python developers without the time to invest in acquiring a whole new skillset, and prevented them from building full-fledged web apps. Luckily for you, we have a solution today: Streamlit.

1.2 What is Streamlit?

Streamlit is a pure Python frontend development library that lets you create web apps called *Streamlit apps* quickly and easily.

As we've seen, a traditional challenge in Python development is that you need to either use a command line to execute your script or write non-Python code to create a visual interface if it's for the web. Streamlit disrupts this by allowing you to write a web-based UI in Python.

In fact, one of my favorite ways to describe Streamlit apps to people new to them is "think of them as Python scripts where you can click buttons and stuff."

Originally launched in 2019, Streamlit has exploded in popularity and usage in the last couple of years, bolstered by—among other things we'll explore shortly—its easy-to-grok syntax, its value in data science, and its support for creating LLM-based chatbots. So much so that in 2022, it was acquired by Snowflake Inc. for \$800M.

To give you a sense of Streamlit's rise, figure 1.1 shows a Google Trends chart:

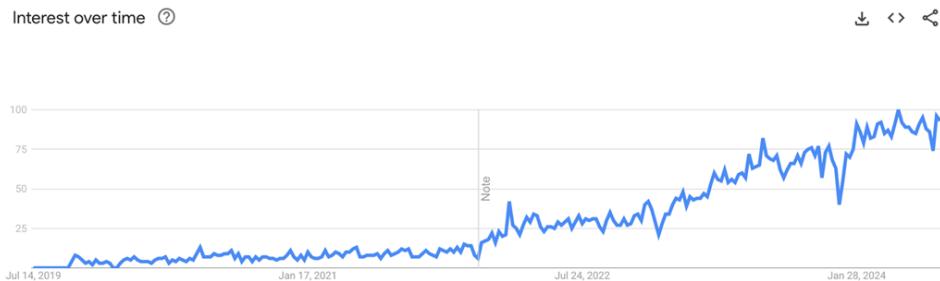


Figure 1.1 A Google Trends chart that shows the popularity of Streamlit over time (note: the dip near the end of 2023 is the week between Christmas and New Year's Day, when I assume relatively few people were working in the US)

1.3 10 reasons Streamlit is so popular

As figure 1.1 shows, Streamlit has enjoyed a fairly steady rise in popularity, especially beginning in 2022. Besides its vibrant community of individual users, the roster of companies that use Streamlit internally includes names such as Netflix, Airbnb, Stripe, and Square.

There are many reasons that Streamlit has gained such acceptance, chief among them being its pure-Python nature and its ease-of-use for data science and AI applications in particular. In this section, we'll explore ten of these reasons.

1.3.1 Streamlit is pure Python

Any code you write with Streamlit is Python code.

Traditionally, creating a web-based interface has required developers to write HTML, CSS and Javascript, the three stalwart languages of the web. *HyperText Markup Language* (HTML) is used for page structure, *Cascading StyleSheets* (CSS) for appearance and layout, and *Javascript* for functionality.

The trouble is, these languages (especially CSS and Javascript) can be hard to master if you want to create something relatively complex. There are frameworks built on top of these languages that help, but more often than not these have a learning curve, too. In any case, you'd still need to know HTML, CSS, and Javascript to use them effectively.

Python, due to factors such as its ease-of-use and its rich ecosystem of libraries for data wrangling and analysis, is popular among data scientists, hobbyists, and even semi-technical people who learned it to help with their day jobs (and because it's fun). These groups are illustrated in figure 1.2. Their skillset may not extend to the three web languages, or they may only have a passing knowledge of them—usually not enough to create complex applications.

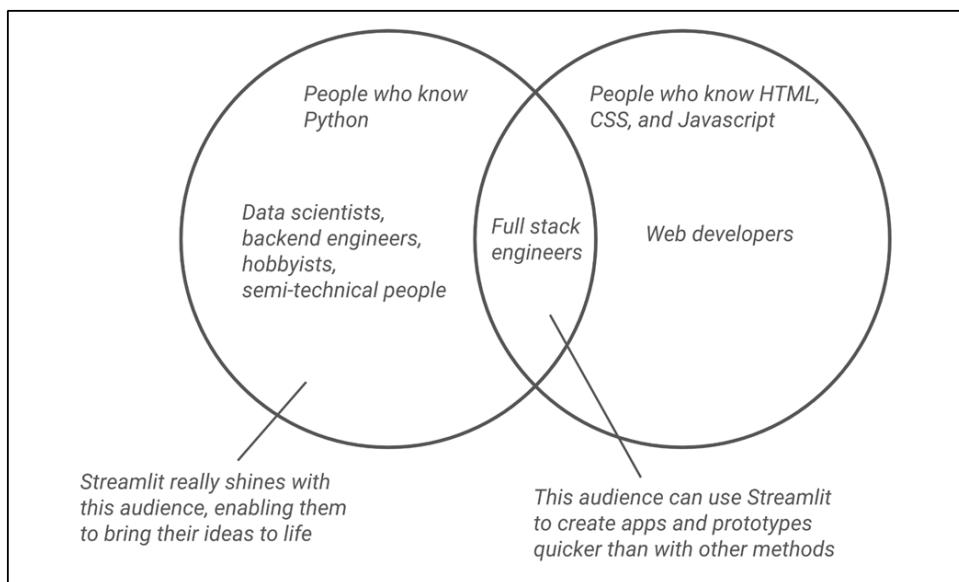


Figure 1.2 Streamlit unlocks web app development for anyone who knows Python and helps even full stack developers prototype and build faster

Streamlit is a boon to these people because it lowers a huge barrier for them; it lets them create rich applications for the web without first having to invest time in learning another stack of languages.

1.3.2 Streamlit lets you go from idea to app in minutes

When you start using Streamlit, I can guarantee that you'll be impressed by how little time it takes to create a working application.

Due to its intuitiveness and the fact that it gives you sensible defaults for most things (which means you don't have to customize or configure them manually to look and work great), apps created in Streamlit are really quick to develop.

Indeed, often, the time it takes to go from an idea to a fully working app can be measured in minutes rather than hours or days.

1.3.3 Streamlit makes beautiful apps

Even if you do know some of the web-based languages I mentioned earlier, *knowing* them is very different from being skilled at *using* them to create web pages that look good.

In Streamlit, the apps and pages you create are beautiful by default. That's because the elements (individual parts of an app, such as buttons, checkboxes or tabs) that Streamlit creates for you have been pre-designed to look good. All you have to do is put them together.

As anyone who has used CSS to style a web page by hand (or tried to, at any rate) can tell you, it can be frustratingly hard to get just the right spacing you want between the edges of a button and its text, or the unique shadow effect you've dreamt up. And if, through force of will, you manage to implement exactly what you had in mind, it's *still* not guaranteed to be pretty, because UI design is as much art as it is science.

Streamlit doesn't necessarily solve all of those problems for you, but it *does* make it so you have to go out of your way to create something that *doesn't* look nice.

1.3.4 Streamlit lets you focus on your app, not UI details

Giving you pre-designed elements you can then combine to create apps has another advantage; it frees up your time to focus on the part you know best: your app's logic.

Streamlit intentionally limits the UI choices you can make by choosing *for* you.

As an example, take the snippet of code shown in Listing 1.1

Listing 1.1 Using tabs in Streamlit

```
import streamlit as st

tab1, tab2, tab3 = st.tabs(["Mission", "About us", "Careers"])
with tab1:
    st.header("Our Mission")
    st.write("Our mission is to teach people to make web apps in Python.")

with tab2:
    st.header("About Us")
    st.write("We are a group of Python enthusiasts.")

with tab3:
    st.header("Careers")
    st.write("We are hiring! Apply today!")
```

You don't need to know how this works just yet, but it produces the tabbed page shown in figure 1.3.



Figure 1.3 Tabs in Streamlit, illustrating how Streamlit makes UI choices for you

Turn your attention to the tab bar at the top. Notice the orange line beneath the tab we're currently on and the way the tab we're hovering over on is highlighted. Also, though you can't see it in a screenshot, the transition from one tab to another has a small animation where the orange line moves under the new tab.

The line that produces the tab bar is the following:

```
tab1, tab2, tab3 = st.tabs(["Mission", "About us", "Careers"])
```

Notice how nothing in this line says anything about the styling we just discussed. We essentially just said "tabs" and Streamlit took care of the details for us.

This is because Streamlit correctly recognizes that most developers don't want to design these UI-related minutiae and would much rather spend that time implementing their business logic.

The result is that Streamlit app developers are highly productive, and able to churn out sensible interfaces that complement rather than detract from their functionality.

That said, there is ultimately a tradeoff here between hassle-free UI development and fine-grained control. Streamlit works great for you if you care about the former, but not the latter. If you want to be able to exercise a lot of control over the finer aspects of your interface, Streamlit may not be the tool for you.

For instance, as of the time of writing, if you wanted to put a shaded box around the current tab instead of a line underneath, you can't do that easily or without knowing HTML and CSS.

1.3.5 Streamlit's syntax is simple, concise, and intuitive

One of the things I've always admired about Streamlit is how readable the syntax is. Like Python itself, Streamlit code is self-documenting and you'll often find what it does obvious.

For example, let's say you want to simulate the roll of a die and plot the results in a graph. Consider the four-line snippet shown in Listing 1.2 that does this.

Listing 1.2 A die roll simulator in Streamlit

```
import streamlit as st
import random

st.title("Die Roll Simulator")
num_rolls = st.slider('Number of die rolls', min_value=10, max_value=100)
if st.button('Plot Graph'):
    die_rolls = [random.randint(1, 6) for _ in range(num_rolls)]
    st.line_chart(data=die_rolls)
```

You can see the output in figure 1.4.

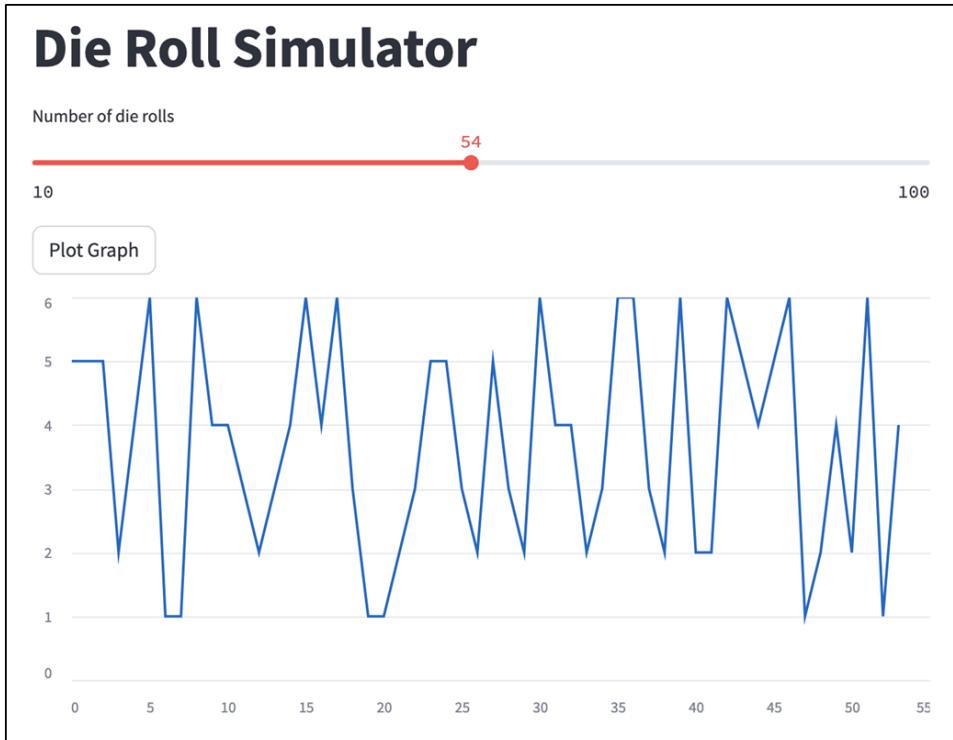


Figure 1.4 Output of a die roll simulator in Streamlit

Notice how you can likely read the code in Listing 1.2 and make sense of it without having seen a single line of Streamlit code before: we show a title, display a slider for the user to pick the number of die rolls to plot (between 10 and 100), show a "Plot graph" button, and when it's clicked, we generate the die rolls (random numbers between 1 and 6) and plot them in a line chart.

There's none of the "setup" code that you would see with other languages or libraries, such as defining a handler to listen for a button-click event or defining a laundry list of attributes for the slider. It's short and simple.

1.3.6 Streamlit is great with LLMs

For better or for worse, technology had a watershed moment in 2022 with the launch and subsequent reception of OpenAI's generative AI chatbot, ChatGPT.

To understand how the rise of generative AI has resulted in an explosion in popularity for Streamlit, consider the following:

- AI has captured the world's imagination.
- Python is the most popular language for AI development because of its wide adoption across the industry and its rich ecosystem of AI-related libraries such as TensorFlow, PyTorch, scikit-learn, and LangChain.
- Streamlit is the fastest way to write visual apps in Python.

The intersection of these facts has meant that developers of all stripes have been flocking to Streamlit to develop AI apps.

Streamlit itself was quick to capitalize on the sudden popularity of Large Language Models (LLMs) like GPT.

For instance, Streamlit makes writing conversational chatbots a cinch through the introduction of chat elements.

Listing 1.3 shows a complete working AI chatbot I built using Streamlit chat elements in less than 30 lines of code:

Listing 1.3 A working AI chatbot in less than 30 lines of code

```
import os
import streamlit as st
from openai import OpenAI

os.environ["OPENAI_API_KEY"] = "sk-..." # Replace with your own API key
openai = OpenAI()

human_message = lambda m: {"role": "user", "content": m}
ai_message = lambda m: {"role": "assistant", "content": m}

def talk_to_ai(question, history):
    return openai.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=history + [human_message(question)],
    ).choices[0].message.content

st.session_state.history = st.session_state.get("history", [])
history = st.session_state.history

for message in history:
    st.chat_message(message["role"]).markdown(message["content"])

if prompt := st.chat_input("Chat with me!"):
    st.chat_message("human").markdown(prompt)
    response = talk_to_ai(prompt, history)
    history.extend([human_message(prompt), ai_message(response)])
    st.chat_message("ai").markdown(response)
```

NOTE To get this to work, you need to make an OpenAI account, generate an API key and plug it into the line `os.environ["OPENAI_API_KEY"] = "sk-...."`. Importantly, the above code is for demonstration purposes only. In practice, you should never include API keys in any code that you distribute or share. Later in this book, we'll explore how to work around this.

Figure 1.5 shows the output of our code. You don't have to understand the code just yet, but you can hopefully see that Streamlit created a fully functional chat interface for us, complete with cute little bot and user avatars.

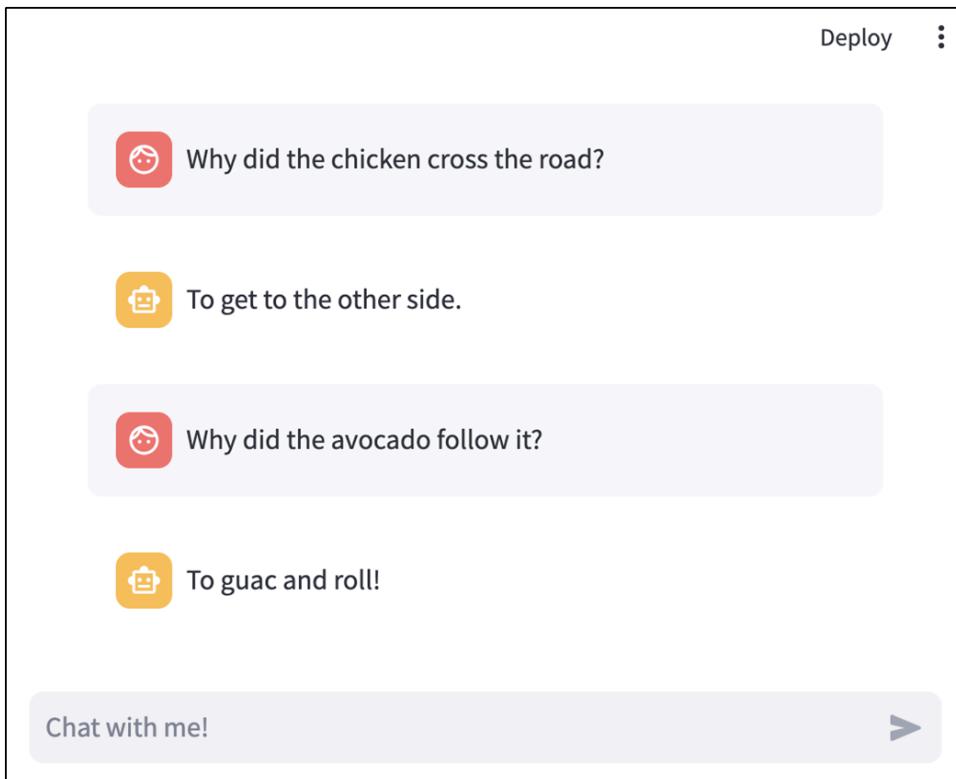


Figure 1.5 A complete AI chatbot in Streamlit

Time will tell if the hype around generative AI proves to be well-founded. In the meantime, if you're considering writing an AI app, Streamlit has you covered.

1.3.7 You can share your Streamlit apps for free, in record time

Building a web app is one thing, but making it available to people is quite another. For a public-facing web app that doesn't use Streamlit, usually that involves finding a way to host it (such as using a cloud provider like AWS or even acquiring and managing your own servers).

All of this can seem intimidating (not to mention expensive) to the time-strapped data scientist or casual hobbyist, especially if the app they're trying to distribute is relatively simple and doesn't need to cater to thousands of users.

Enter Streamlit Community Cloud, a completely free and fast way to deploy an unlimited number of public-facing Streamlit apps, designed with the same simplicity that you'll find in Streamlit itself. Using it is as simple as linking your GitHub repository (a way to share your code and manage different versions) to it.

In fact, most of the projects and examples used in this book, besides being available on GitHub, have also been published to Streamlit Community Cloud where you can access and play with them.

Community Cloud does come with limitations, so it may not be for everyone, but it is a nice, hassle-free way to share your creations.

We discuss deploying to Community Cloud in detail in Chapter 5, and other options in Chapter 12 if Community Cloud doesn't meet your specific needs.

1.3.8 Streamlit has a huge, friendly community

Streamlit's user base is growing every day, and the more it grows, the more questions people have. Luckily, Streamlit's forums are friendly and the members (including the team behind Streamlit) are quite responsive.

In researching this book, for instance, I needed to find out more about how Streamlit works under the hood. A forum comment by a Streamlit engineer helped me identify the right place to check in the source code.

If you've exhausted your Google-fu and still have a question that stumps you, help is at hand.

1.3.9 Streamlit has excellent support for data science and visualizations

Streamlit's original design prioritized data scientists, so you may be unsurprised to learn that it has great support for all sorts of data visualizations, powered by Python's already-rich set of visualization libraries.

This means that you can use your favorite visualization library to create charts (Matplotlib, Plotly, Altair etc.), graphs (GraphViz) or 3D renders (PyDeck), and display them in Streamlit.

Figure 1.6 shows a histogram rendered in Streamlit using the Matplotlib library.

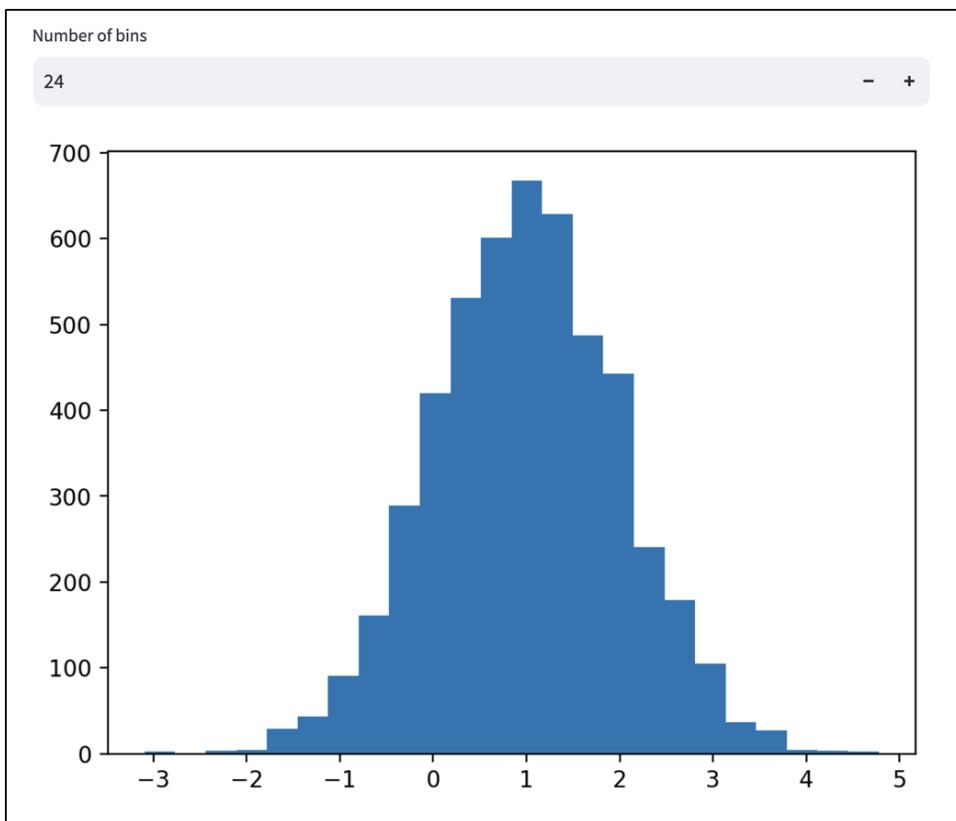


Figure 1.6 A histogram in Streamlit created using the popular Matplotlib library

Streamlit also works great with Pandas, which is an extremely popular library that makes working with tabular data easy via its central concept of dataframes.

A dataframe is a table-based data structure that enables developers to ingest, wrangle and analyze data in various ways. If you're a data scientist, there's a good chance that you use it regularly in your work.

Streamlit has first-class support for dataframes with the ability to display them directly and even edit them visually.

Figure 1.7 shows an example of this where we enable the user to edit a Pandas dataframe live in the Streamlit app.

	Name	Age	City
	Alice	21	New York
	Bob	42	
	Carol	19	Chicago
	David	26	Houston
	Eve	30	Phoenix

Average age: 27.6

Figure 1.7 An editable Pandas dataframe as displayed in Streamlit

We'll explore Pandas and dataframes in more detail in a later chapter.

1.3.10 You can extend Streamlit with third-party components or build your own

As we saw earlier, Streamlit saves you time by providing pre-built UI elements and limits the amount of customization you can perform on them.

Most of the time, this is actually a good thing since it gives you space to focus on your logic. Other times, it can feel restrictive because the specific experience you have in mind may be challenging to build out of the box with Streamlit's building blocks.

In these situations, Streamlit provides a way out in the form of Streamlit Components, which are modules that third-party developers can create to extend its functionality. Streamlit Components can range from things that fill perceived gaps in Streamlit's natively-available elements, such as a search box with autocomplete, to entire mini-apps (an audio recorder, say) that you can embed in your app.

Streamlit publishes popular components in a gallery on its website where you can view how they look and work. Installing a component is as easy as installing any other Python library.

If you have some frontend development experience, you can even create your own components. This does require knowledge of the web languages we mentioned earlier (HTML, CSS, and Javascript), but can enable you to fine-tune the experience you create for your users. We explore how to create our own Streamlit Components in Chapter 15.

1.4 What can you build with Streamlit?

Streamlit offers a versatile platform for creating a wide array of interactive applications. In this section, we'll explore the diverse range of projects that can be built using Streamlit, showcasing its adaptability and utility.

1.4.1 Data applications

Streamlit was originally designed for data scientists, and its usefulness in creating data applications remains one of its greatest selling points.

The kinds of data-related applications you can create with Streamlit include, but are not limited to:

- Dashboards that display metrics that decision makers at a company care about
- Data exploration apps that enable you to dive into and get a feel for datasets
- Visualizations that users can interact with to understand data better
- Machine learning model deployments that enable users to upload inputs to get back predictions

1.4.2 Internal tools for your workplace

Data apps may be the most well-known use case for Streamlit, but in my opinion, it is equally well-suited for developing internal tools for employees at your company.

This might include applications such as:

- Project management dashboards
- Time tracking apps
- Shift scheduling tools
- Inventory management systems
- File converter utilities, and more.

Several factors make these tools ideal candidates for Streamlit apps:

- They generally need only cater to a limited number of concurrent users
- They are scrappy in nature and may be required in a short time frame
- Most companies don't have room in their budgets to hire full-time engineers to build them

Streamlit is easy enough that even semi-technical people with a passing knowledge of Python can wield it quite effectively to build what they need.

1.4.3 Apps that use Generative AI like LLMs

Writing generative AI apps often tends to involve adding a thin layer of business logic on top of API calls to a generative AI service such as OpenAI's GPT or Anthropic's Claude.

Streamlit works quite well for these tools, letting you roll out AI functionality with a UI in a short time with out-of-the-box support for common AI form factors such as chatbots.

It also helps that Python's libraries for interacting with generative AI (LangChain, for one) are second to none, and Streamlit is perfectly placed to take advantage of them.

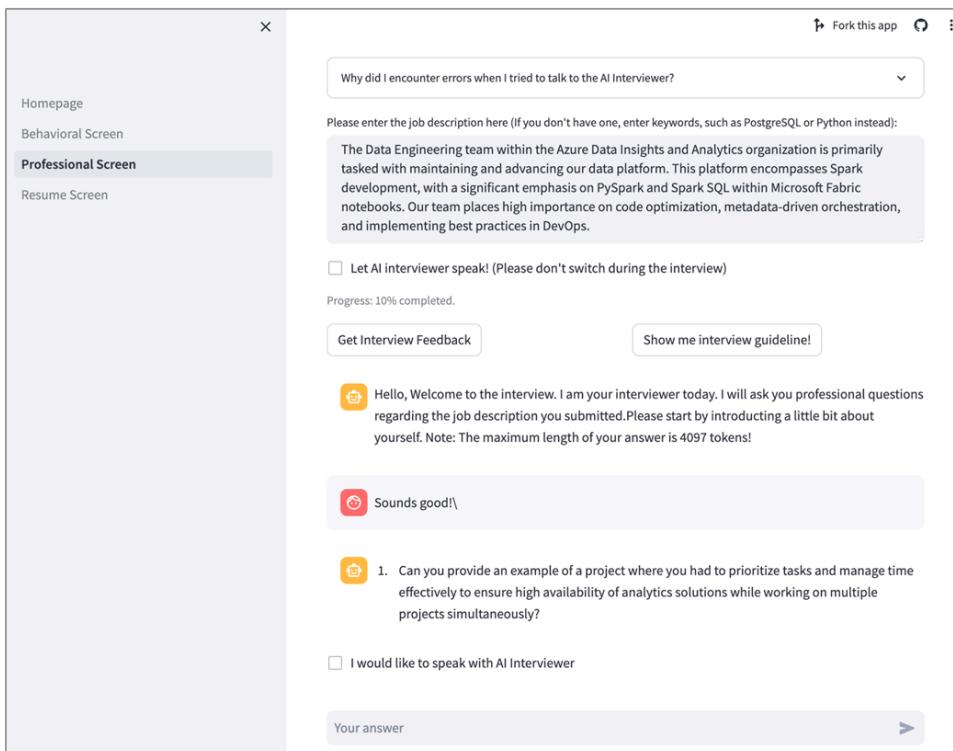


Figure 1.8 An AI interviewer chatbot from Streamlit's app gallery (<https://aiinterviewer.streamlit.app/>)

1.4.4 Prototypes for large apps

Streamlit can be useful even to a dedicated software engineering team tasked with building a large, ambitious application. Such projects tend to be expensive in terms of developer time and effort to develop, and the work is spread out over months or years. Even with an iterative approach to development, it frequently takes a long time to see any results, at which point many incorrect and expensive-to-fix assumptions may have been made.

Early design mocks help solve this issue, but prototypes are even better. With Streamlit, you can churn out lightweight working prototypes that mimic the functionality of the larger app quickly. This shows stakeholders what to expect, and helps validate basic assumptions about the functionality early on in the process.

If you're a software engineer, going through this exercise not only saves the company time and money in the long run; it also drums up excitement among your partners and builds support for your work because people appreciate something they can experience first-hand more than design documents and mocks.

1.4.5 Anything else you can dream up

Earlier in the chapter, I characterized Streamlit apps as "Python scripts where you can click buttons". What this boils down to is that your Streamlit app can do pretty much anything Python can.

You don't have to use Streamlit in ways that people are familiar with. You can find new, creative things to code up into an app. Here are some unconventional app ideas for inspiration:

- A personal AI habit-building buddy that lets you record your activity and offers advice and encouragement
- A maze generator that generates fun puzzles (see figure 1.9)
- A laundry tracker that keeps track of when your clothes need to be washed

The point is, you should feel free to experiment! Often, learning a technology sparks ideas about new potential applications of it that only you could have thought about!

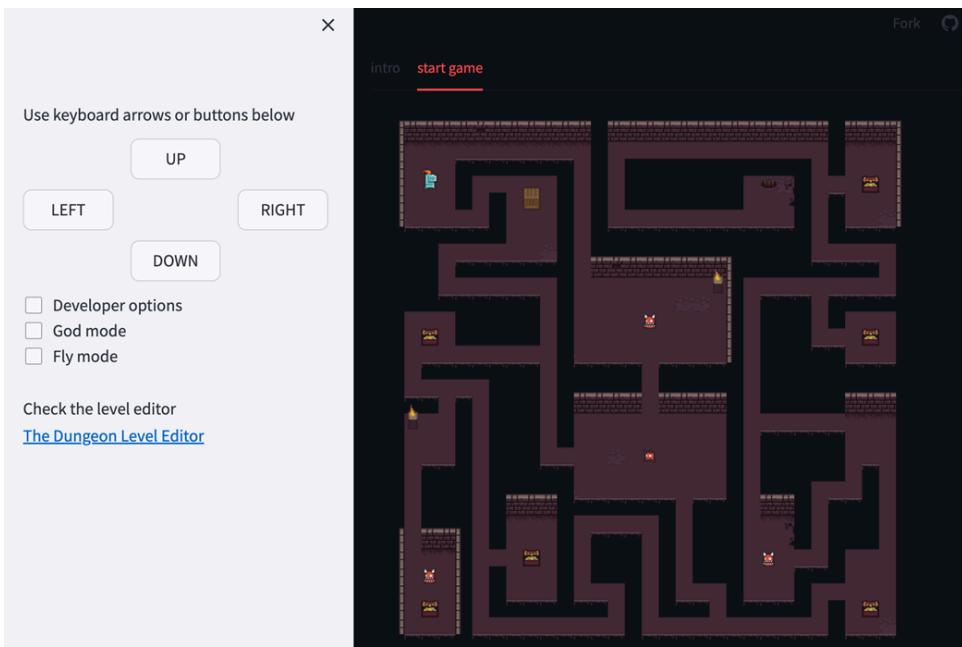


Figure 1.9 Dungeon, a game created with Streamlit (<https://dungeon.streamlit.app/>)

1.5 What not to use Streamlit for

Like any other technology, Streamlit comes with tradeoffs. Let's address these by taking a look at a few things you can't (or probably shouldn't) use Streamlit to develop.

1.5.1 Complex, large-scale applications

The size of a Streamlit app's ideal user base is likely measured in the hundreds or thousands of users, not in the millions.

When it comes to concurrent users (users accessing your app at the same time), depending on the kind of app you're creating and its resource requirements to serve a single user, once you cross a threshold, your app may have trouble scaling.

As we'll see in a later chapter, Streamlit works by running your entire Python script from top to bottom each time something needs to change on the app screen. This can have implications for performance, especially if your script executes heavy computations. Streamlit's caching feature can usually mitigate this issue, but you may occasionally encounter situations where you can't use caching.

There are other reasons that Streamlit may be unsuitable for large production-grade applications, though you can often overcome these limitations by using third-party components. For instance, Streamlit doesn't have a built-in authentication feature, but there are Streamlit components created by other people that add this (e.g., `streamlit-authenticator`).

As the complexity of your app increases, you may find yourself straying from the beaten path more frequently. While the Streamlit forums are invaluable for identifying workarounds for features that Streamlit is missing, once your app reaches a certain level of sophistication, migrating to a different, more flexible framework (such as Flask, Django or some of the other options we explored earlier) with enhanced capabilities may be warranted.

1.5.2 Apps that require a high degree of UI customization

Streamlit is designed to simplify adding most common UI elements into your app, but now and then you may run into situations where you want fine-grained control over how a part of your interface works. Streamlit's limited customizability may pose challenges here.

Though Streamlit does offer some relief in the form of theming for color customization or the ability to write your own components (or include ones others have shared) to augment its native functionality, the larger point stands.

If you need to have precise control over how your app looks or if there are specific visual effects you're trying to achieve, Streamlit may not be for you. Consider alternatives like React, a framework known for its flexibility and extensive customization capabilities (explored in the following section), or even manual page design with traditional HTML and CSS.

1.5.3 Native desktop or mobile apps

Streamlit is a web framework, meaning that it produces apps that run within a web browser. If you're trying to develop desktop or mobile apps outside the browser, opt for native application frameworks such as PyQt or React Native.

It's worth mentioning that you can still access Streamlit apps on mobile devices; the point here is that Streamlit does not produce standalone Android or iOS apps that run independently of the browser.

1.6 How is Streamlit different from other technologies?

If you've thought about or researched ways to create interactive applications in the past, chances are that you've come across or possibly used technologies similar or related to Streamlit before. This section compares Streamlit to some of these in an effort to clarify your understanding.

1.6.1 Jupyter notebooks

A Jupyter notebook is an interactive environment for data exploration and working with code and visualizations. The concept is fairly simple: you write Python code (or even just text or markdown) in a "cell," execute it, and see the output (which can be text, some kind of visualization or even something interactive) directly below. The output of each cell is retained so you can see what came before it.

Useful for things like explaining your thought process, playing around with data, and sharing your work with colleagues, Jupyter notebooks have become a mainstay in the data science community.

Jupyter is similar to Streamlit in a few respects: they are both Python-based, both support tabular and graphical data, both integrate with popular libraries like pandas, and they are both popular in the data science world.

However, there are also important differences:

- Jupyter is used to create interactive documents, rather than actual apps; it is best used for sharing your code and explanations with collaborators rather than having that code be executed by actual end users. Streamlit, as we've seen, is used to create apps for end users.
- Jupyter's support for engineering practices such as version control is sketchy; this is fine because it's meant for exploratory analysis, not shipping production dashboards. Streamlit, on the other hand, fits well into a regular engineering workflow.
- The audience for Jupyter notebooks consists of technical folks, not laypeople. Streamlit doesn't require the users of your app to be able to read or understand code.

All told, Jupyter notebooks are a fantastic initial tool for experimenting with code, data and charts. Streamlit is an amazing tool for building and sharing the final buttoned-up interactive experience you want users to have, often using the code, data and charts you finessed in your Jupyter notebook.

1.6.2 HTML, CSS and Javascript

As we discussed earlier, HTML, CSS, and Javascript are the big three complementary languages of the web.

HTML, or HyperText Markup Language, is used to provide the structure and content of a web page, and define elements like headings, lists, divisions, and links through a tree of "tags" which can contain other tags.

CSS, or Cascading StyleSheets, lets web developers manage the appearance, layout, and formatting of many different web pages in a central place. CSS has options to set colors, spacing, fonts, borders, and more.

Javascript is a programming language primarily used to define the dynamic behavior of a web page. It can be used to create animations, validate forms, connect to other webpages, and pretty much anything else you'd want a web site to do.

Together, these powerful languages enable you to build any web experience you want. While they aren't difficult per se to learn, using them effectively to create complex apps is a laborious process, so much so that "frontend" development is its own discipline in software engineering.

Streamlit abstracts away the complexity of using these languages in their raw form and lets you use a simpler, more concise syntax to write web pages, working in the background to generate the equivalent code that a web browser can understand.

1.6.3 React

React is a popular Javascript framework that helps you create fast, responsive web pages. Like Streamlit, it is an alternative to using raw HTML, Javascript, and CSS to create web applications.

React follows a reusable component-based design where you build up parts of a web page using smaller parts and then combine *those* parts to build larger parts, working all the way up to a full application. Each such part you create is called a component, and can be reused in your app or even shared with other people so they can use it in theirs.

React has a declarative programming approach where you describe the UI you want, and React takes care of updating the DOM (the tree of HTML elements that lies under your web page) to match it.

While React is incredibly powerful and used by frontend developers to create complex UIs, its architecture and approach can be challenging to wrap your head around. It is also not Python-based.

Though Streamlit actually uses React under the hood, from the programmer's perspective, Streamlit trades some of the power and flexibility of React for simpler semantics, pre-built elements with some customizability, and the ability to use Python to write web pages.

1.6.4 Flask and Django

Flask and Django are both Python-based web frameworks.

Flask is lightweight and minimalistic. It's designed to be straightforward to use, and is quite flexible, allowing developers to choose the various libraries and tools they prefer rather than being prescriptive. Flask provides the essentials you need to create a web app, such as the ability to handle routing and HTTP requests, while leaving most of the design choices beyond that to the developer. It's popular with web developers who want to employ a modular approach or exercise a high degree of control.

Django, on the other hand, is feature-rich and sophisticated. It's more opinionated than Flask, following specific design patterns such as the MVT ("Model-View-Template") architecture. It also comes with many built-in modules for common tasks, such as an admin panel to manage your data models. Django is a popular choice to run powerful enterprise-grade applications.

While Flask and Django are similar to Streamlit in that they can all be used to build web applications, a key distinction is that Flask and Django are primarily *backend* frameworks to be used in conjunction with your *frontend*, which still consists of HTML/CSS/Javascript that you have to write or embed within your Python code. Not so with Streamlit, which lets you write frontend code in pure Python.

1.6.5 Tkinter and PyQt

Tkinter and PyQt (pronounced "pie-cute") are Python libraries used to create graphical user interfaces (GUIs).

Tkinter comes bundled with Python and provides a set of widgets you can use to create a desktop application. It is popular with beginners and suitable for creating simple apps.

PyQt is similar, but it's more powerful and mature (and more complex) than Tkinter. It's actually a wrapper around the C++ Qt application framework. PyQt lets you create quite sophisticated GUIs and has a wide range of features.

As you might have guessed, the key difference between Tkinter/PyQt and Streamlit is that the latter is used to create web applications, not desktop applications. Over the last couple of decades, users have increasingly become comfortable with web-based software that never needs to be installed or updated, which makes Streamlit arguably more useful than the other two for most use cases.

1.7 Learning through projects

Now that we know what Streamlit can and can't do, let's talk about what's in store for you as you go through the rest of this book. I strongly believe that the best way to learn a piece of technology is to gain experience in using it.

This book is designed to give you this experience by building real projects that require you to think through actual problems and how to solve them. In fact, through these projects, we will learn not only about Streamlit itself, but also:

- large language models (LLMs) like OpenAI's GPT, and how to create LLM-powered apps
- best practices around User Experience (UX) and User Interface (UI) design
- best practices around code organization and structure

Here's a sneak preview of some of the things we'll build together:

1.7.1 An interactive metrics dashboard

In this project, we'll study the requirements of a CEO who needs to make important decisions about their company, and design an interactive dashboard that they'll look at every day to understand what's going on in the business.

Along the way, we'll explore Streamlit's charting and visualization capabilities, as well as how to work with Pandas dataframes.

1.7.2 A rent-vs-buy housing calculator

In our second real-world application, we'll evaluate the benefits and costs of owning a home as opposed to renting, and design a calculator that helps people make their decision.

Here, we'll experiment with Streamlit's layout options and different ways for users to provide inputs. We'll also dive deep into how Streamlit maintains state.

1.7.3 A CRUD application

Our next stop is a *Create-Read-Update-Delete* (CRUD) application where you'll need to go beyond a siloed app and establish a connection to a database for persistent storage.

We'll see how this unlocks powerful possibilities in the kinds of things you can build. We'll also explore data tables and theming.

1.7.4 An AI trivia app

With three real apps that explore Streamlit's key features under your belt, it's now time to turn your attention to the exciting world of generative AI. To get a taste of what you can do in this space, we'll design an app that can answer trivia questions using OpenAI's GPT.

1.7.5 A chatbot

In this project, you'll expand your LLM capabilities with the LangChain libraries and build your very custom chatbot. We'll learn how Streamlit's chat elements make this easy to do.

1.7.6 A custom knowledge base

Want a second brain? For our third LLM project, that's essentially what we're building. We'll learn how to integrate various sources of information into a query-able knowledge base and expose it in a Streamlit app.

1.7.7 Your own Streamlit component

This one's pretty advanced, so strap in! We'll first identify some functionality we want that Streamlit's pre-built elements can't serve out of the box. We'll then use the React framework to create that functionality and integrate it into an app.

1.8 Summary

- Streamlit is a framework that lets you build web apps in pure Python with no HTML, CSS or Javascript.
- Streamlit has been exploding in popularity due to its simplicity, development velocity, support for LLMs, powerful visualizations, and integration with data science libraries, among other features.
- You can create many types of applications with Streamlit: data apps, internal tools for your workplace, LLM apps, prototypes for larger apps, and more.
- You shouldn't use Streamlit for large-scale apps meant for millions of users, or apps that require a high level of UI customization.

2 Getting started with Streamlit

This chapter covers

- Setting up your development environment
- The Streamlit development workflow
- Building and running your first Streamlit app

Welcome to Chapter 2. This is where the rubber hits the road! By the end of this chapter, you'll be interacting with your first very own Streamlit app!

This book is not just about teaching you Streamlit though; it's also about making sure you're productive with Streamlit and well-placed to develop apps in the real world. So before we jump into writing code, we'll take some time to set up your development environment. Specifically, we'll talk about three things you'll want to consider while doing so: version control with Git, code editing tools, and virtual environments.

We'll then examine the workflow you'll follow while coding with Streamlit so you know what to expect as you build apps throughout this book. With that out of the way, I'll walk you step-by-step through actually creating your first app, a password checker.

Excited yet? Let's dive in!

2.1 Getting Streamlit up and running

First things first! Before you can build any apps, you need Streamlit installed and ready to go. This involves two steps:

- Install the right version of Python (3.8 and above), and pip (a tool that ships with Python and can install Python packages easily)

- Use pip to install Streamlit (spoiler alert: type `pip install streamlit`)

For a detailed installation guide, see Appendix A of this book.

2.2 Setting up your development environment

The tools you use and the way you set up the environment you code your apps in are largely a matter of preference. However, over time, these matters can have an outsized impact on your productivity as a developer, so it's worth your time to consider them.

In this section I'll briefly discuss a few important aspects of your development environment, namely version control, your editing tool, and virtual environments.

If you're an experienced Python developer and already have a setup you're comfortable with, go ahead and skip to the section on installing Streamlit.

2.2.1 Version control with Git

If you've never written code in a professional setting, there's a chance you may not have used a version control system like Git before, or don't fully understand what it's for.

Version control is a structured way to track, manage, document, and experiment with making changes to your programs. Think of it as a time machine for your code.

Git is far and away the most popular version control system that exists today, so in this book, we'll use the terms *Git* and *version control* interchangeably.

In the course of working on your apps, there will be occasions when you change your mind and decide to design something differently. In those situations, Git allows you to "go back in time" to how your code *used* to be and apply the changes from that point.

You can also experiment with different versions of your code while testing out multiple design options and switch between them easily with Git. Perhaps my favorite aspect of Git is that it lets you document changes to your code, giving you a place to explain your thought process on why you decided to make things a certain way. Trust me—you'll thank yourself six months later while you're reading through your own code and trying to decipher it.

All of these situations are extremely common when you're working on a large project, or collaborating with others. Even if that *doesn't* currently apply to you, I highly recommend learning and incorporating Git into your development workflow because the benefits it offers are too numerous to ignore.

If I've sold you on the benefits of version control, check out the section called "A whirlwind intro to Git" in Chapter 5 for a quick look at the basics. It's designed to be a standalone section, so if you like, you can go to it right now and come back when you're done. If you're (understandably) impatient to get started with Streamlit, read on.

2.2.2 Code editors

Streamlit apps, like all Python scripts, are just text files, so all you *really* need to write code is a simple text editor (like Notepad in Windows).

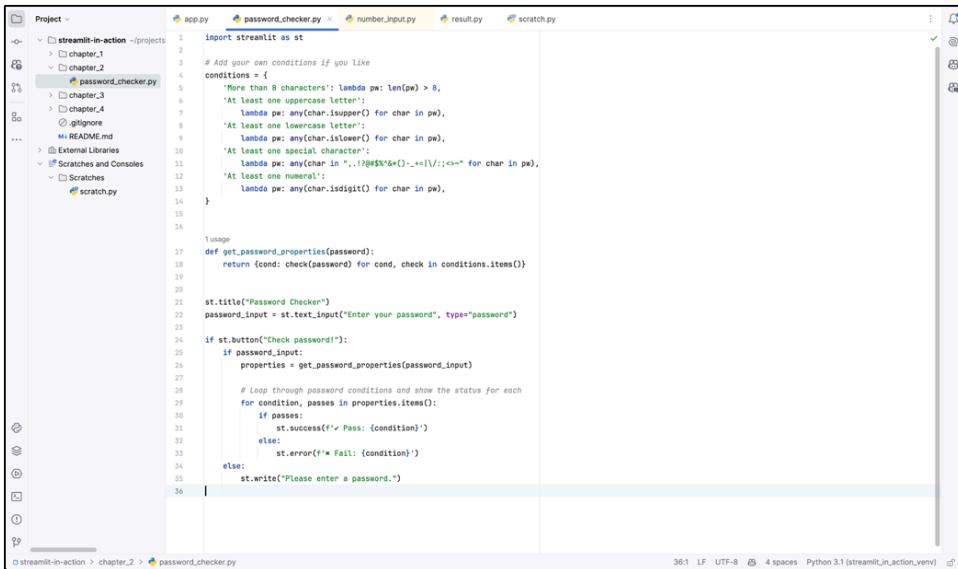
However, using an advanced code editor or an *Integrated Development Environment* (IDE) makes you so much more productive—giving you features like syntax highlighting, easy debugging tools, and code navigation—that it's hard not to recommend using one.

There are many, many tools that will fit the bill, but I'd like to mention two of the most popular ones here.

PYCHARM

PyCharm is a versatile IDE tailor-made for Python, developed by the company JetBrains. It offers comprehensive support for Python out of the box, such as code completion, error detection, quick-fix suggestions and more.

PyCharm is an excellent choice if you're seeking advanced functionality straight from installation without having to tinker around too much. The Pro edition costs money, but PyCharm has a Community Edition you can get for free at <https://www.jetbrains.com/pycharm/download> (make sure you scroll down to where it says "Community Edition").



The screenshot shows the PyCharm Community Edition interface. The left sidebar displays a project structure for a folder named 'streamlit-in-action'. Inside, there are several subfolders ('chapter_1', 'chapter_2', 'chapter_3', 'chapter_4') and files ('app.py', 'password_checker.py', 'number_input.py', 'result.py', 'scratch.py'). The 'password_checker.py' file is open in the main editor area, showing Python code for a password checker. The code imports 'streamlit' and defines a function 'get_password_properties' that checks a password against a set of conditions. It then creates a Streamlit app with a button to check a password and loops through the properties to show success or failure messages. The status bar at the bottom indicates the file is 36.1 LF, UTF-8 encoding, 4 spaces indentation, and is using Python 3.1 (streamlit_in_action_venv).

```

import streamlit as st
# Add your own conditions if you like
conditions = {
    'More than 8 characters': lambda pw: len(pw) > 8,
    'At least one uppercase letter': lambda pw: any(char.isupper() for char in pw),
    'At least one lowercase letter': lambda pw: any(char.islower() for char in pw),
    'At least one special character': lambda pw: any(char in ".!@#$%^&()_-+=|/;:<=>" for char in pw),
    'At least one numeral': lambda pw: any(char.isdigit() for char in pw),
}

def get_password_properties(password):
    return {cond: check(password) for cond, check in conditions.items()}

st.title("Password Checker")
password_input = st.text_input("Enter your password", type="password")

if st.button("Check password!"):
    if password_input:
        properties = get_password_properties(password_input)

        # Loop through password conditions and show the status for each
        for condition, passes in properties.items():
            if passes:
                st.success(f"\u2708 Pass: {condition}")
            else:
                st.error(f"\u2708 Fail: {condition}")
    else:
        st.write("Please enter a password.")

```

Figure 2.1 The project editor window in PyCharm Community Edition

VISUAL STUDIO CODE

Visual Studio Code (VS Code for short) is an immensely popular code editor maintained by Microsoft. It supports a multitude of languages.

VS Code offers some essential functionality such as syntax highlighting right away, but its true strength lies in its ecosystem of plugins that can extend its capabilities. Indeed, with the right set of plugins, you can make VS Code do essentially everything PyCharm Professional (the paid version) can do.

Since VS Code is completely free, it's an attractive proposition for learners who don't mind spending some time to set everything up.

Note: Since both of these (and many others, such as Sublime Text and Notepad++) are viable code editing tools, I won't assume that you're using any particular one. As long as you have a terminal you can type commands into and a program that can edit text files, you're good to go as far as this book is concerned.

2.2.3 Virtual environments

Most useful real-world Python projects you work on will require you to rely on a variety of libraries. These libraries change all the time, releasing new versions that add and remove features or modify existing ones.

As you gain experience and create more complex applications, you'll often find that updating a library for one project causes another to break or leads to unpredictable bugs and conflicts.

Virtual environments are a way out of this dilemma. A virtual environment consists of an isolated instance of Python and a set of libraries and dependencies. Each project you start should live in its own virtual environment. That way, you can modify the dependencies of any project without affecting any others.

Even if you're just starting out and haven't run into dependency management problems, it's a great idea to get familiar with virtual environments. There are several virtual environment-related libraries and tools available with varying levels of sophistication: you may come across `venv`, `pipenv`, `pyenv`, `poetry` and others.

We'll discuss virtual environments in more detail in Chapter 13 while exploring how to distribute your code.

2.3 Running Streamlit for the first time

This is where the fun begins! In this section, we'll run a Streamlit app for the first time, and get a first-hand look at the kind of things you can devise with Streamlit.

If you haven't already done so, go to Appendix A and make sure you have Streamlit installed.

If you have, open up a terminal window and let's get started!

The app we'll be running is the `hello` app that comes pre-built with Streamlit. To see it in action, type `streamlit hello` in your terminal.

This should display some output on the terminal that you can ignore for now, and after a few seconds, opens your web browser to display the actual app.

The `hello` app shows off Streamlit's capabilities by demoing a variety of features: animation, graph plotting, maps, and tables.

Figure 2.2 shows one such demo: an animation built from a mathematical visualization. You can navigate to other demos using the sidebar on the left.

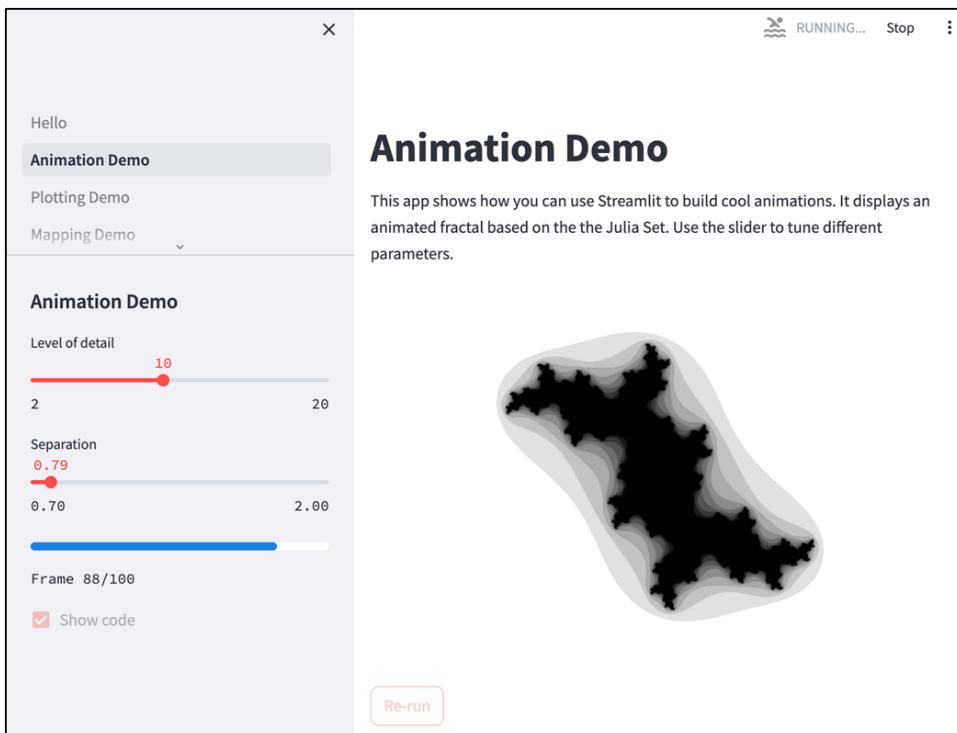


Figure 2.2 Animation Demo from streamlit hello

Figure 2.3 shows a more practical demo related to Pandas dataframes. We'll become intimately familiar with these later in the book.

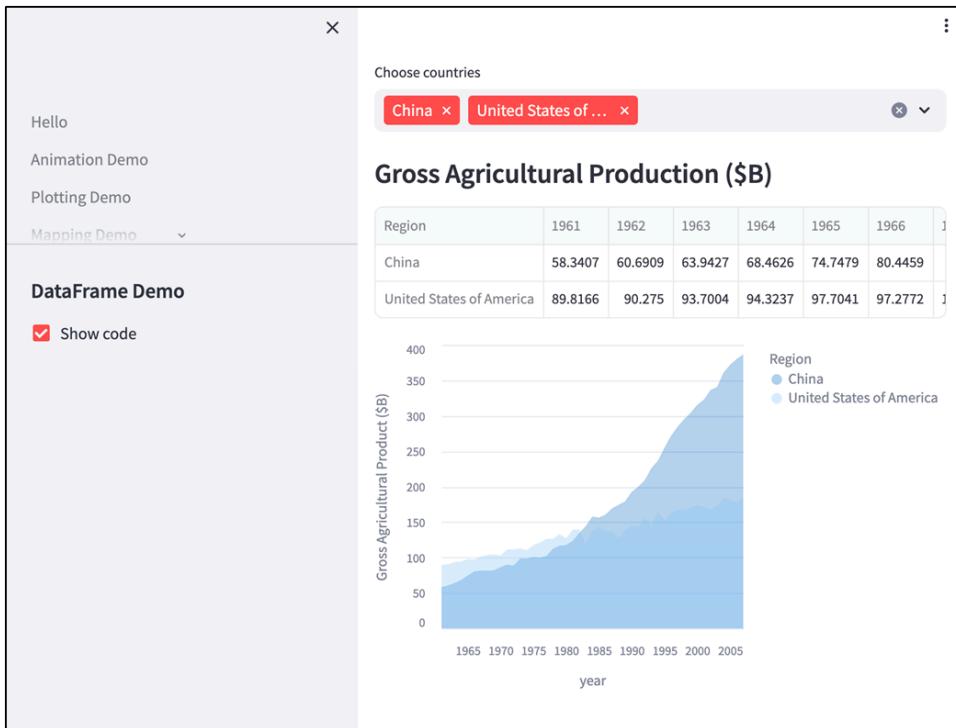


Figure 2.3 DataFrame Demo from streamlit hello

I hope looking at these examples inspires you!

The sidebar also has a “Show code” checkbox which, as you might imagine, displays the source code for each app below the app itself. In each instance, you'll find that the code isn't terribly verbose.

Obviously, you don't have to understand how all of this is done right now, but hopefully you'll gain some appreciation for what Streamlit makes possible.

2.4 The Streamlit development workflow

Writing Streamlit apps - or any kind of programming, really - is an iterative process where you write some code, test if it works, and repeat. If you've never written graphical applications before, you may be curious about what this process looks like.

Figure 2.4 describes the development workflow you'll soon get used to.

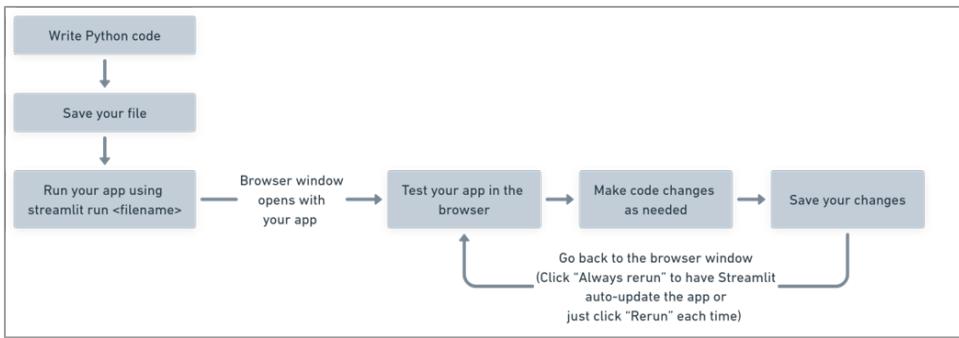


Figure 2.4 The Streamlit development workflow

Let's walk through the steps:

1. **Write your Python code:** Create a first draft of your app in a Python file in your code editor. This can be as basic or as fleshed-out as you like. Some people like to run the app right from the outset as they begin to shape it and visualize every change as they go, so their first draft may just be an empty app with a title.
I myself find this distracting since it's tempting to spend time fine-tuning the appearance of the app if I see it early on. Instead, I prefer to code up a roughly complete version of the app before I run it for the first time. This is entirely a matter of preference though.
2. **Save your file:** This should be self-explanatory. Save your Python file as you would normally, with a `.py` extension.
3. **Run your app using `streamlit run <filename>`:** Run this in your terminal and replace `<filename>` with the path to your file.
This opens a browser window with your app, as you saw when you ran the `hello` app.
4. **Test your app in the browser:** Interact with your app. Play around with it by entering values, clicking buttons etc., and seeing if the output matches what you'd expect.
5. **Make code changes as needed:** Based on your testing, go back to your Python code and edit it.
6. **Save your changes:** Again, self-explanatory.
7. Switch back to your browser window: Remember, you don't need to re-run `streamlit run <filename>` to see your changes in the app.
8. **You have two options here:** If you want Streamlit to re-run your app with your changes automatically, you can click "Always rerun" in your app. Otherwise, you can still click "Rerun" each time.
9. Repeat steps 4-7

You'll be going through these steps a *lot* throughout this book, and it'll soon become second nature. But enough theory, let's put this in action!

2.5 Building your first app

In this section, we'll craft our very first app! To do this, I'll first introduce the concept behind the app and outline the flow of logic.

I'll then walk you through the code for the complete app, explaining each part of it step-by-step. Finally, we'll run the app and make changes to it.

A word of advice before we dive in: Your learning experience will be richer if you take an active role in it, tinkering and experimenting as you go. Be curious and don't be afraid to make changes outside of those suggested in the text! You'll often find that the parts of the material that stick in your mind the best are the ones where you took the initiative to explore and understand on your own.

With that said, it's time to get our hands dirty!

2.5.1 A password checker

You've probably visited one of those websites that have a huge list of conditions that your password needs to fulfill, like containing at least one lowercase letter, one special character, an even number of underscores, the names of up to two dwarves from Snow White, and so forth.

The app we'll create is one that lets the user enter a password they're considering, and shows them which conditions pass and which ones fail.

2.5.2 Logic flow

Figure 2.5 lays out the flow of our app's logic.

It begins when the user enters a password in an input text box and initiates the checking process by clicking a button.

Internally, the app maintains a list of conditions (technically a dictionary, in the code) to be checked in its memory. We loop through this list, categorizing each condition as either "PASS" or "FAIL" depending on whether the entered password meets it, and generate a fresh list (or dictionary) of results.

Following this, we iterate through the results, presenting the outcomes on the screen. Each condition is represented by a box, displayed in green if it passed or red if it failed.

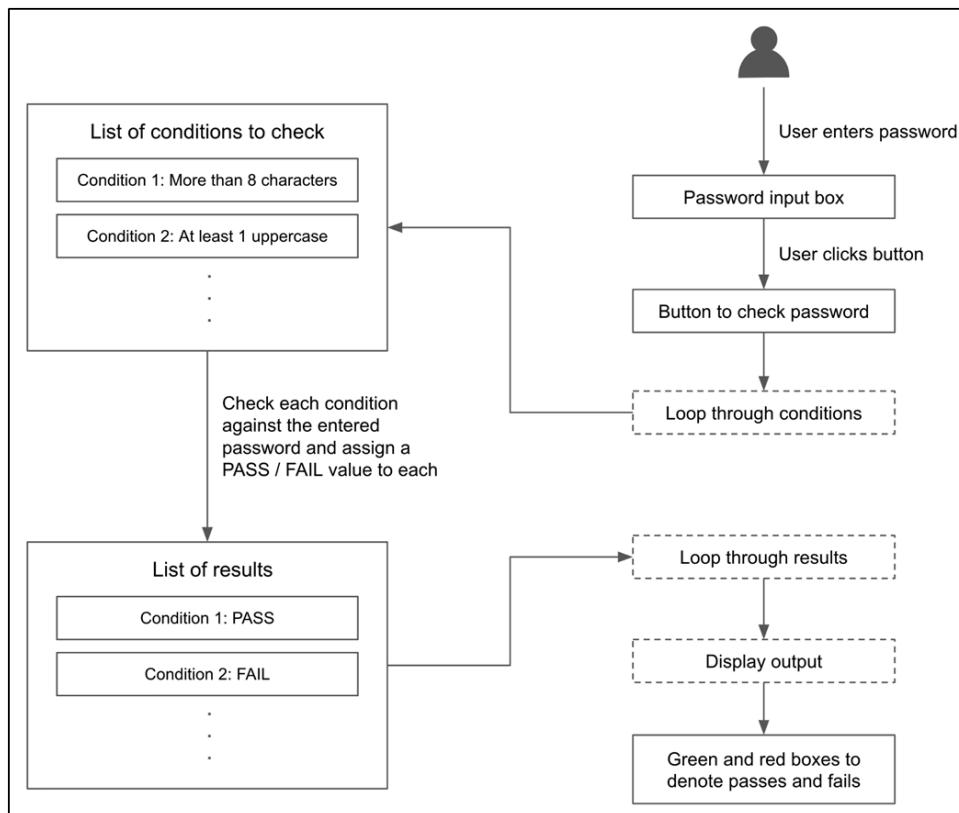


Figure 2.5 The flow of logic in our password checker app

2.5.3 Going through the code

By now, you hopefully have a clear understanding of the logic we want to implement, so let's jump right in and start building!

To get started, go to this book's GitHub page, copy the code in Listing 2.1 into a new file, and save it.

Listing 2.1 The code for our password checker app

```

import streamlit as st

# Add your own conditions if you like
conditions = {
    'More than 8 characters': lambda pw: len(pw) > 8,
    'At least one uppercase letter':
        lambda pw: any(char.isupper() for char in pw),
    'At least one lowercase letter':
        lambda pw: any(char.islower() for char in pw),
}
  
```

```

'At least one special character':
    lambda pw: any(char in ",.!?@#$%^&*()-_+=|\\/;,<>~" for char in pw),
}

def get_password_properties(password):
    return {cond: check(password) for cond, check in conditions.items()}

st.title("Password Checker")
password_input = st.text_input("Enter your password", type="password")

if st.button("Check password!"):
    if password_input:
        properties = get_password_properties(password_input)

        # Loop through password conditions and show the status for each
        for condition, passes in properties.items():
            if passes:
                st.success(f'✓ Pass: {condition}')
            else:
                st.error(f'✗ Fail: {condition}')
    else:
        st.write("Please enter a password.")

```

Let's go through it part by part:

Our first line imports Streamlit itself (you'll need this in every app) and notes that we'll use `st` to refer to it later:

```
import streamlit as st
```

You can technically use whatever you like here, but the convention of `st` is so widely used that you'll frequently hear people referring to Streamlit elements as `st.<whatever>`

We use the following block to define the conditions that we want to check for:

```
# Add your own conditions if you like
conditions = {
    'More than 8 characters': lambda pw: len(pw) > 8,
    'At least one uppercase letter':
        lambda pw: any(char.isupper() for char in pw),
    'At least one lowercase letter':
        lambda pw: any(char.islower() for char in pw),
    'At least one special character':
        lambda pw: any(char in ",.!?@#$%^&*(-_=|\\/:;,<>~" for char in pw),
}
}
```

We do this by creating a Python dictionary, where the keys are the conditions themselves, and the values are the corresponding functions that we'll use to check that condition.

In this case, the functions are *lambdas* or anonymous in-line functions. A lambda is a cool way to define a short one-line function in Python without having to go to the trouble of giving it a name. Each function accepts one parameter—the password, `pw`—and returns a Boolean value of true if the corresponding condition is met, and false otherwise.

The first lambda function simply checks if the length of the password is more than 8 characters. In each of the others, we loop through the characters in `pw` and apply a test (e.g., `char.isupper()`, `char.islower()` etc.) to each.

To evaluate the password for all of the conditions defined above this is the function we'll run:

```
def get_password_properties(password):
    return {cond: check(password) for cond, check in conditions.items()}
```

It returns a new Python dictionary where the keys are the conditions and the values are the results of running the lambda function corresponding to each condition that we defined above.

The syntax we've used here is called a *dictionary comprehension* in Python, and it's short-hand to create a new dictionary from some input.

Next is our first actual Streamlit element. It's a simple one called `title`:

```
st.title("Password Checker")
```

It does what you'd expect it to, which is to display the passed text as a title, in nice large type.

We use the following line to show a password input box to the user:

```
password_input = st.text_input("Enter your password", type="password")
```

Streamlit's implementation of this also offers a toggle to show/hide the entered text. When the user enters some text, it's saved in the `password_input` variable.

`st.button` is one of the simplest Streamlit elements and one you'll be working with frequently. Unsurprisingly, here it displays a button with the text "Check password:"

```
if st.button("Check password"):
```

Notice that this goes in an `if` clause. Whatever code is nested within the `if` (see below) is evaluated when the button is clicked. There are some interesting nuances to this that we'll dive into in Chapter 4.

Once the button is clicked, our app checks if `password_input` has a non-empty value, i.e., if the user has actually entered a password. If they have, it calls `get_password_properties` on it to evaluate the conditions we defined:

```
if password_input:
    properties = get_password_properties(password_input)

    # Loop through password conditions and show the status for each
    for condition, passes in properties.items():
        if passes:
            st.success(f'✓ Pass: {condition}')
        else:
            st.error(f'✗ Fail: {condition}')
```

It then loops through the returned dictionary, where each key is a condition and the corresponding value is a boolean indicating if the condition passed.

If the condition passes, we use another Streamlit element, `st.success` to indicate this, and if it didn't, we use `st.error` to show that it failed.

`st.success` and `st.error` are both just containers for text with some semantic styles applied (i.e., mostly a green box for `st.success` and a red one for `st.error`)

This `else` corresponds to the `if password_input` from earlier:

```
else:
    st.write("Please enter a password.")
```

Here, we're using yet another Streamlit element, the general `st.write`, to ask the user to enter a password.

2.5.4 Running the app

To run the app, in your terminal, navigate to the directory where you saved your file, and type in:

```
streamlit run <filename>
```

For instance, if you saved your code under the file name `password_checker.py`, you would type:

```
streamlit run password_checker.py
```

As you saw when you ran `streamlit hello` earlier, this displays some output in your terminal window and opens your web browser where you can see the app.

Feel free to enter various input passwords and play around with the app! If everything has gone according to plan, you should see something similar to figure 2.6 when you click "Check password."

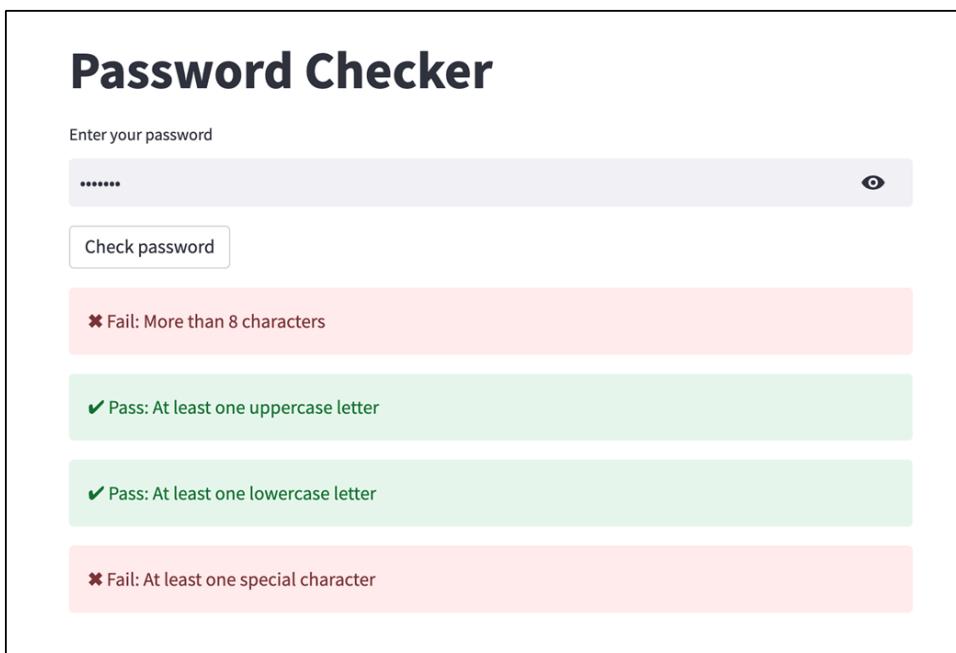


Figure 2.6 Our password checker app

Don't close this page because we're shortly going to see what happens when you make changes to the code.

Before that, let's turn our attention to the terminal output:

```
You can now view your Streamlit app in your browser.
```

```
Local URL: http://localhost:8503
```

```
Network URL: http://192.168.50.68:8503
```

Notice the part where it says Local URL: http://localhost:8503.

This is Streamlit's way of telling you that there's a Streamlit server running on your computer (that's what `localhost` means) on port 8503. The port in the output you see might be something different, like 8501; this is totally fine.

We'll explore how this works in more detail in Chapter 3, but for now, all you need to take away from this is that you can also access your app by going to the "Local URL" specified in the output.

If there are other computers on your network, the Network URL is what those other machines can use to access your app.

Don't close the terminal window or cancel out of it just yet!

NOTE In some cases, you may find that either your browser window does not open automatically when you enter `streamlit run`, or the browser window opens but the page is blank. If this is true for you, here are some things to try:

1. In your browser, manually type in the address listed against "Local URL" in the terminal output, e.g., `http://localhost:8503`
2. Make sure you're running the most up-to-date version of your browser.
3. Navigate to `http://localhost:8503` in a different browser. I've found that Google Chrome tends to have the least issues.

2.5.5 Making changes to the app

Now that we know our app works, let's try making changes to it. For instance, let's say we want to add a check that the entered password contains at least one numeral.

You would do this by adding a new condition to our `conditions` dictionary:

Listing 2.2 Adding a new condition

```
# Add your own conditions if you like
conditions = {
    'More than 8 characters': lambda pw: len(pw) > 8,
    'At least one uppercase letter':
        lambda pw: any(char.isupper() for char in pw),
    'At least one lowercase letter':
        lambda pw: any(char.islower() for char in pw),
    'At least one special character':
        lambda pw: any(char in ",.!?@#$%^&*(-+=|\\/;,<>~" for char in pw),
    'At least one numeral':
        lambda pw: any(char.isdigit() for char in pw),
}
```

Our button doesn't sound excited enough, so let's also add an exclamation point to the text:

```
if st.button("Check password!"):
```

Once you've saved your file, head back to your browser window where you have the app open. On the upper right-hand corner you should see a message informing you that the source file has changed (see figure 1.5) and options to "Rerun" or "Always rerun."

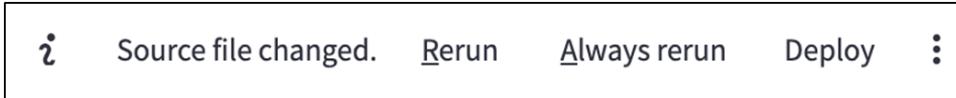


Figure 2.7 Streamlit displays a message on your app when the source changes

This is because Streamlit monitors your app file to see if any changes have been made to it. Click "Rerun" to re-run your app with the latest code.

You'll see that our button now says "Check password!" with the exclamation point, and if you enter a password and click it, you'll see the new numeral test we added. This updated view is shown in figure 2.8.

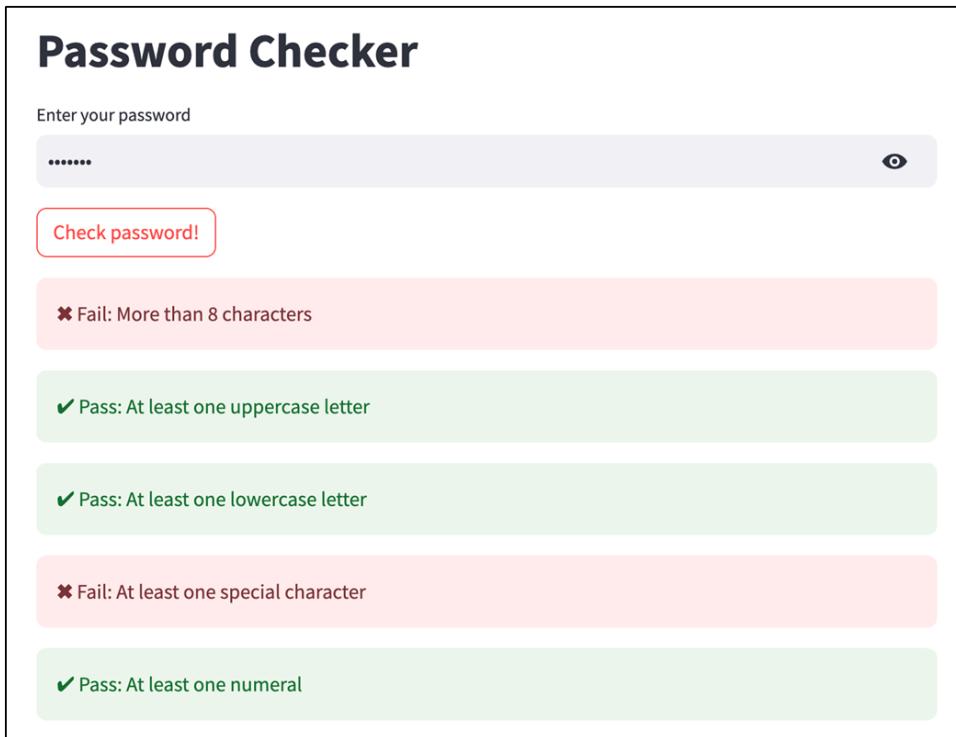


Figure 2.8 The password checker app with our newly added changes

You could also choose “Always rerun,” which means Streamlit will automatically update your app every time you change your code. This makes for a smoother development experience where you edit your code and immediately switch to your browser to see what effect it had.

If you want to disable this behavior (it can sometimes be hard to spot what's changed on a page when you have it on), you can click the hamburger menu on the top right, go to “Settings” and uncheck “Run on save.”

2.5.6 Killing and restarting the server

Throughout all of this, recall that you've kept your terminal window (where you typed `streamlit run ...`) open. This window is where the Streamlit server that "serves" your app is running.

Once you're done playing with the app, you probably want to shut the server down so it's not consuming resources.

To do this, head to the terminal window now and either close it or press “`Ctrl+C`.”

If you now go back to your open browser window, you'll find that you can't interact with your app any more. The button's disabled, it says “CONNECTING” at the top right, and a connection error shows up as seen in figure 2.9.

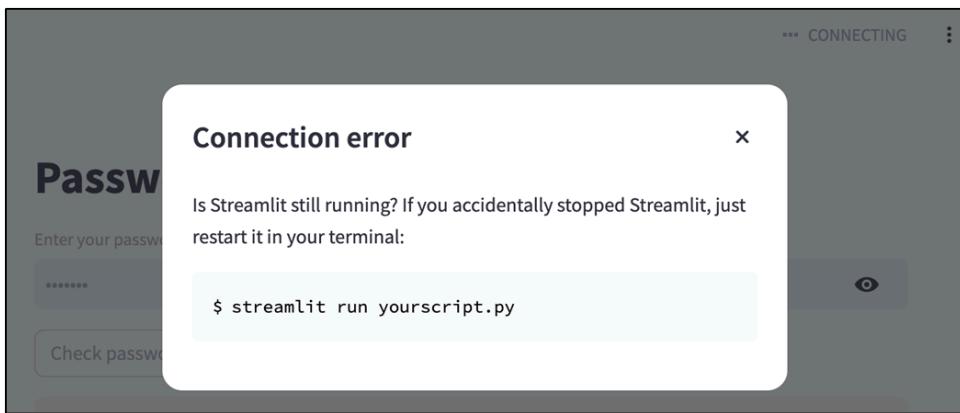


Figure 2.9 When you shut down the Streamlit server, the frontend app can't run

You can bring your app back to life by running `streamlit run <filename>` again, which restarts your server and re-establishes the connection between the frontend and the newly restarted server.

This will open a new browser window with a *different* instance of your app. You'll find that the old window is also now alive because it's able to re-establish the connection to the server.

2.6 Summary

- Streamlit requires Python 3.8 or above to run.
- Git is a form of version control, which helps you track and manage changes to your code.
- Advanced code editors like VS Code and IDEs like PyCharm make you more productive through syntax highlighting, debugging tools, code navigation, autocomplete and more.
- Virtual environments allow you to isolate the libraries and dependencies of each project you work on.
- You run Streamlit apps using `streamlit run <file_name>`, which opens a web browser window with your app.
- You can make Streamlit always re-run your app whenever the source code changes for a seamless development experience.

3 Taking an app from concept to code

This chapter covers

- Defining the scope of an application
- Designing the user interface
- Organizing the code for an app
- Streamlit's workhorse input widgets

In my beginning days as a software engineer, I was often surprised by how much of my time was spent on activities other than writing code. I'd spend multiple days or even weeks simply understanding the problem I was trying to solve and even more time in design, all before typing a single line of code.

At the time, I felt anxious because I didn't feel *productive*. Scribbled meeting notes and design docs didn't *do* anything. With time, I realized that those days and weeks weren't actually wasted; they made the end-product better because of how deeply I'd thought about what I was trying to do.

Likewise, this book is not *just* about teaching you to write Streamlit code. It's about helping you learn to develop apps in the real world. Planning and design are inescapable parts of this process.

Though we don't have weeks to spend on these topics, this chapter will give you a taste of the end-to-end experience of developing an app.

We'll start with the concept for an app and convert it into a set of requirements. We'll then come up with a design that meets those requirements, working backwards from the user experience, and also think about code organization.

Finally, we'll walk through our code and logic, introducing some of Streamlit's most common widgets as we go. That's a lot to get done, so let's get started!

3.1 Concept to code: A six-step process

Writing a piece of software can be an overwhelming task once you get past the initial spark of inspiration. There are so many things to consider! Where do you start? What features are you going to develop and how long will it take? How will users interact with your app? Should you start coding right away and figure it out as you go?

To quote Desmond Tutu: "There is only one way to eat an elephant: a bite at a time."

In creating Streamlit apps, as in consuming large land mammals, the optimal approach is to break it down into smaller chunks. Figure 3.1 displays a simple, logical six-step process you can follow while developing an app, or really just about any piece of software:

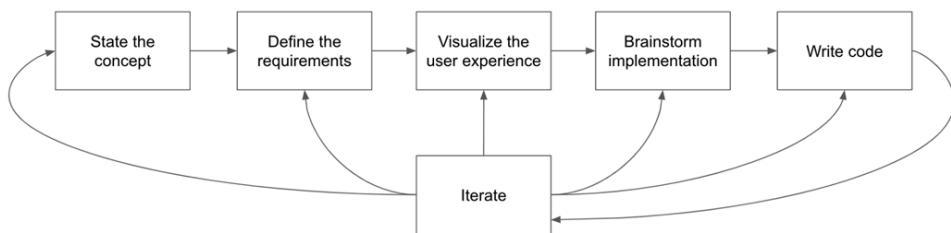


Figure 3.1 The six-stage app development flow

1. **State the concept:** To solve a problem—or to describe it to others—you first need to be able to state it succinctly at a high level.
2. **Define the requirements:** This is where you refine your concept and break it down into cold, hard requirements. Part of this is to define the scope of what your app will and—perhaps more importantly—won't do.
3. **Visualize the user experience:** Draw diagrams and mocks of what your envisioned user's experience will be when they use your app.
4. **Brainstorm implementation:** What components will your solution have, how will they integrate with each other, and what tradeoffs do you face?
5. **Write code:** Actually implement your app.
6. **Iterate:** Inspect your output and refine steps 1 through 5 as necessary.

This may sound like a lot, and you may find yourself thinking: "Is this all really necessary? I'm not building enterprise software in a large team, I'm making a fairly small app for a handful of users."

The beauty of the above steps is that you can adjust them to what makes sense for your project. If you're working on a large project, each of the above steps may take a long time since there's probably a lot of people that need to agree on the overall approach.

But if you're building something small, you can scale each step down to something more reasonable. For instance, you can define the requirements in a quick bullet-point list that takes maybe five minutes to generate, and your visualization of the user experience can be a simple sketch.

In the rest of this chapter, we'll go through each of these steps in the context of an example app.

NOTE For much of this chapter, our attention is going to be on the end-to-end app development process; we'll only cover various Streamlit elements towards the end when we discuss the implementation of the frontend. This is by design, to reflect the real world, where your primary focus in developing a graphical app will be the app itself, not Streamlit. That actually explains the success of Streamlit: it *gets out of your way* so you can develop your app without worrying too much about how you're going to implement the UI.

So even though you won't see a lot of discussion about Streamlit itself in the beginning parts of this chapter, hang in there! We'll get to it organically where it fits best!

3.2 Stating our concept: A unit converter

Whether it's while studying physics, cooking a meal based on a recipe, or traveling internationally, I'm sure you've been in situations where you've had to convert between different units of measurement like cups to ounces, yards to meters, and so on. You may have had to look up the conversion factor online and then use a calculator to do the actual conversion.

The app we'll be working on in this chapter will make this task easier and more streamlined.

As we discussed in the last section, the first step in the development process is to "state the concept," i.e., to express the problem we're trying to solve succinctly, preferably in a single sentence or line.

Here's our concept:

CONCEPT A Streamlit app that allows users to effortlessly convert between different units of measurement like distance, mass, and time.

Stating the concept uncovers the core of what you're trying to do and focuses your thinking, preventing it from wandering off in a dozen directions.

As you're developing the app, you may hit upon various new possibilities and potential features to incorporate. The stated concept is there as a reference point to ensure that whatever changes you're considering still satisfy the core idea laid out in it.

For instance, you might think, "Maybe I should make a unit overview page where I explain the history of the unit and what it was named after." If you compare it to our one-line concept, you'll realize that such a page doesn't really help with our goal of having users convert between units effortlessly, so it's probably wise to drop it.

Of course, this doesn't mean that the concept you've stated is set in stone. If you encounter an idea that significantly enriches the experience you could provide the user but it doesn't fit well with the concept, feel free to re-state the concept.

Finally, stating the concept also has the benefit of giving you a one-line description that you can share with other people to help them understand your work. This can be especially valuable in helping you gain users or find collaborators to partner with.

3.3 Defining the requirements

Your concept works great as a sort of mission statement for your app, but it's somewhat hand-wavy about the details. The next step then is to break the concept down into concrete *requirements* you can build towards.

Think of these as the specific list of things your *stakeholders* (your users, teams that you work with, etc.,) would want the app to do.

Requirements should:

- express a capability the app should have
- be free of "implementation" language, e.g., it should not refer to any technology

For an app you're building in real life, coming up with the list of requirements might involve interviewing people to understand their needs.

For the purposes of our example, here's the list of requirements we want to center our app around:

REQUIREMENTS:

- The user should be able to enter a numeric quantity and the unit the quantity is in (the "from-unit").
- The user should be able to select a unit to convert to (the "to-unit"), which must measure the same type of thing as the from-unit. No converting from pounds to yards, for instance.
- The app should display the converted value as output.
- The app should be able to handle conversions within the imperial or metric systems (e.g., feet to inches or meters to centimeters), and across metric systems (e.g., feet to meters).
- Adding new units or quantities to the app should be straightforward.

Notice how the above points are a lot more concrete than our initial concept. Also note how requirements don't just come from the user; they can also come from other people.

To elaborate, the first four requirements listed above are things that the *user* wants, while the last one is something that makes life easier for the *developer* or *maintainer* as you'd want to minimize the time you spend in responding to common requests like adding another unit of measurement to the app.

In an organizational setting, you can also imagine requirements from other stakeholders, such as the analytics team ("we should be able to monitor and track usage of the app") or the monetization team ("the app should allow the user to subscribe for a monthly fee").

You may have realized that the requirements don't mention anything about the *technology* used to accomplish them. For example, they don't say "the app should display a Streamlit button the user can click to perform the conversion."

This is intentional; requirements are about what stakeholders actually need. How you fulfill them is up to you, the developer.

3.3.1 Defining what the app won't do

While listing the requirements is an important step that provides clear goals to accomplish, it can sometimes feel like you're spelling out the obvious. In practice, there's value in doing it even if that's the case because what's obvious to you may not be obvious to your users or your collaborators. It's quite common for people to form a picture in their head about what you're promising that bears no resemblance to reality.

For that reason, it's vital—and possibly more enlightening—to also define what's *not* in your app's scope.

For the unit conversion app in this chapter, we want to keep things as simple as possible because we don't want to spend too much time on it; we're mainly building it to get familiar with Streamlit.

Any sort of auxiliary functionality that doesn't directly relate to the goal of unit conversion is then immediately out. We don't want to build any usage logging or fancy visualizations.

We should also try to keep our conversion logic simple. Converting from pounds to kilograms is easy enough since you can bake the conversion factor into your app, but how about, say, currencies?

Converting from dollars to euros is hard because the exchange rate changes from day to day, or even from hour to hour. To incorporate that, you'd need to read the exchange rates from somewhere, like an online API. We'll read from an API in later projects in this book, but for now, it's probably best to avoid the additional complexity. So currencies are out.

To state it formally:

WHAT'S OUT OF SCOPE

- Usage tracking, logging, visualizations etc., that do not directly relate to unit conversion
- Conversions between units that don't share a simple, constant conversion factor, such as those between currencies

This can be a useful tool for prioritization and phasing. Even if you're placing something out of scope at the moment, you may want to build it later on.

Articulating the requirements and scope lets you express what features you consider to be priority and which ones you might want to push to a future version.

If you have additional stakeholders, having these written down is a good way to elicit feedback and start a conversation about prioritization.

3.4 Visualizing the user experience

So far, we've spent a fair amount of time thinking about the problem we want to solve; we've conceptualized what we'll be building, refined our understanding of what our app needs to do, and even articulated what it *won't* do.

What we haven't done yet is to start designing a solution. That's what we'll do in this next step.

When you start the process of actually developing an app, you may find yourself wondering what to do first. Do you simply start coding your Python app from top to bottom, figuring out your design as you go? Do you try to figure out what the basic components of your app are and how they'll fit together? Or maybe you work on what you think will be the hardest part of the problem to get it out of the way?

Those are all valid approaches and there are arguments to be made in favor of each of them. One approach I've always found valuable, and the one we'll follow in this chapter, is to *work backwards from the user experience*.

Beginning from the user experience is a good way of ensuring the quality of the end product, because it puts the user front and center, directly addressing their needs and preferences. It also helps identify and solve potential usability issues early on in the process.

3.4.1 Creating a mock

So what experience do we want the users of our app to have? To answer that, let's create a mock or sketch of what the UI might look like.

Looking back at the requirements we enumerated in section 3.3, at its core, the app needs to provide a way for users to input a numeric quantity and the units to convert from and to.

Figure 3.2 shows a quick initial sketch.

Value

From unit

To unit

Pounds ▼ Kilograms ▼

Convert

Answer

13 lb = 5.89 kg

Figure 3.2 An initial mock of the unit conversion app

Figure 3.2 shows a first attempt at a UI design. It's quite simple: there's a box where you can type in the value to convert, and you can choose the from- and to- units from a select box. Once you're done you click a "Convert" button and the answer appears below.

You can make a diagram like this with a pen and paper, a marker and whiteboard, or any graphics program you prefer. It doesn't really have to be elaborate or even particularly neat. The important thing is that it should show the end outcome you're trying to create, forcing you to think about your app from the user's perspective.

3.4.2 Getting the user experience right

Let's take a closer look at our mock. There are select boxes to pick a from- unit and a to-unit, but what if the units aren't compatible? What if the user chooses "Pounds" (a unit of mass) as the from-unit and "Feet" (which measures length) as the to-unit and tries to convert between them (see figure 3.3)?

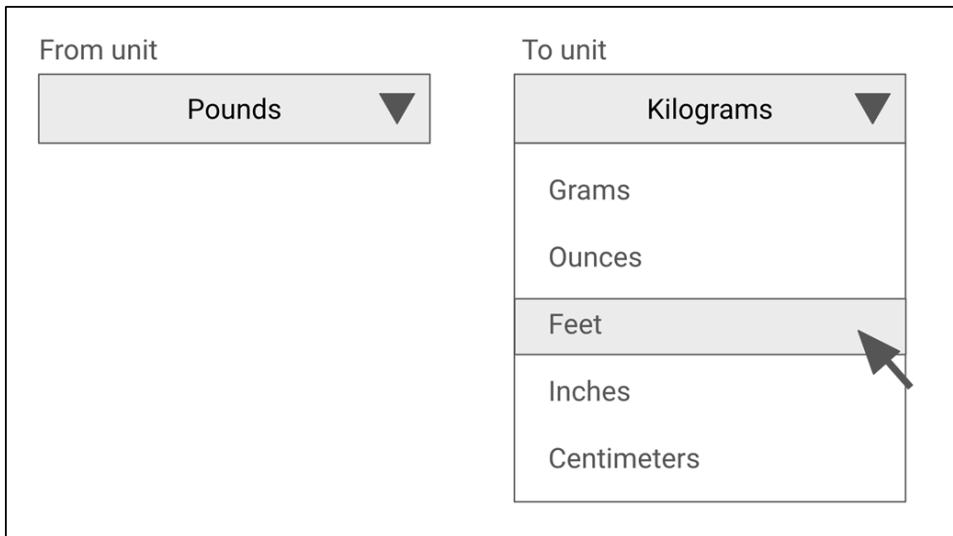


Figure 3.3 Our initial UI makes it possible for users to select incompatible units, like "Pounds" to "Feet"

You could display an error message, of course. That would work, but it wouldn't be a great experience for the user. Users generally like their experience to be as frictionless as possible, and they *hate* getting errors. A better UI would render it impossible to make the mistake in the first place.

For example, perhaps when the user selects "Pounds," we could narrow the options in the to-unit dropdown to only units that are compatible with pounds (see figure 3.4).

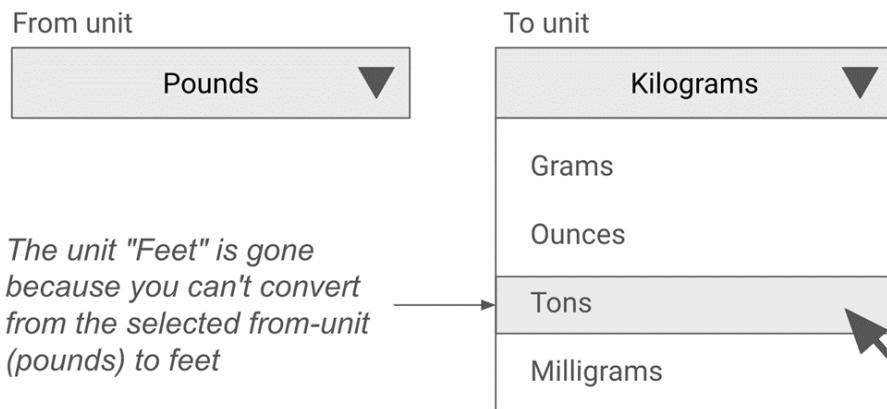


Figure 3.4 We could update the to-unit dropdown to only show units compatible with the selected from-unit

That solves the incompatible-units problem, but may introduce another issue. What if the user wants to select the *to-unit* first? We could potentially apply the reverse logic and update the from-unit list to only contain units compatible with the selected to-unit. Essentially, whatever unit is selected first, the other dropdown is restricted to units compatible with that one, as seen in figure 3.5.

From unit	To unit
Pounds	Kilograms
Grams	
Ounces	
Tons	
Milligrams	

This time, since "Kilograms" is selected as the to-unit, the from-units are restricted to only show units of mass

Figure 3.5 Two-way restriction: When a unit is selected, the other dropdown updates to only show compatible units

But what if the user *now* wants to do a conversion between units of a different quantity altogether, say a length conversion from feet to inches?

When they try to select "Feet" in the from-unit list, they find it's not in the list any more, because the from-unit is artificially restricted to only units of mass by the selection in the to-unit dropdown, which says "Kilograms" (see figure 3.6).

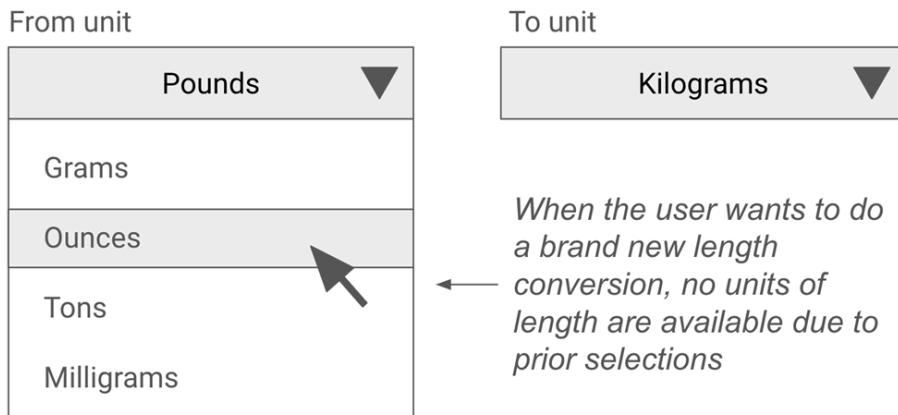


Figure 3.6 Prior selections create a confusing experience because the user can't now select units from other quantities

There are multiple ways to fix this.

We could enforce a rule that the user has to select a from-unit first. This feels like a somewhat artificial restriction though, and not one that's inherent to the business problem.

We could instead introduce a reset button for the user to click whenever they want to start a new conversion, resetting both dropdowns. This works and isn't *terrible* but it seems like a lot of work for the user.

The most intuitive way to handle this would be to simply add another input for the type of quantity the user wants to convert. If the user selects "Mass" in this new dropdown, for example, the from- and to- units would only have units of mass as seen in figure 3.7. Thus, instead of restricting the dropdowns based on each other, we restrict them based on a third external selection.

The figure shows a Streamlit application for unit conversion. On the left, there is a sidebar with radio buttons for selecting a quantity type: Time, Distance, Mass, and Pressure. The 'Mass' button is selected. To the right of the sidebar, there is a 'Value' input field containing the number '13'. Below the input field are two dropdown menus: 'From unit' (set to 'Pounds') and 'To unit' (set to 'Kilograms'). Between these dropdowns is a 'Convert' button. At the bottom of the interface, the converted value is displayed as '13 lb = 5.89 kg' in bold text. A mouse cursor is shown clicking on the 'Milligrams' option in the 'To unit' dropdown menu.

Figure 3.7 Adding a quantity-type selector makes the app more intuitive

This solves the issue much more elegantly while being quite intuitive to the user. It's obvious what the user needs to do here: pick a quantity, enter a value, pick the from- and to- units and click "Convert."

Note that while figure 3.7 shows radio buttons for the quantity type, we could have gone with a dropdown too. The radio button switches things up a bit and involves one less click than a dropdown, but the experience with a dropdown isn't significantly worse. In any case, when you actually get to the implementation using Streamlit, you may identify entirely new possibilities for how to present inputs.

I hope this gives you an understanding of how to start designing a UI and some of the tradeoffs involved. There will, of course, always be more factors we *could* concern ourselves with—for example, does our user care about imperial vs metric system units and should our UI differentiate between them in some way?—but this is already a solid foundation we can feel comfortable with.

We've done a fair amount of refinement to our UI, but you still shouldn't feel that it's necessarily final. As we dive deeper into implementation, you may find more optimizations you'd like to make. In fact, we'll encounter some of these in the following sections and iterate upon our design.

NOTE Since you'll be implementing your UI design using Streamlit, a natural question that might arise while visualizing the user experience is whether you should keep in mind the various elements available in Streamlit so you don't design something you can't actually build.

In my opinion, that's putting the cart before the horse. In the ideal world, you want to figure out the best experience for your users, and then *implement* it using Streamlit. You don't want to let Streamlit limit your thinking about what the ideal user experience looks like.

Of course, this carries some risk; you may, on rare occasions, find that Streamlit doesn't have the exact elements that you were hoping for in your design, and need to adjust your implementation. But that will actually enrich your learning journey by making Streamlit's deficiencies clearer. It will also ensure your user experience won't be constrained by your preconceptions of Streamlit's feature set, which is expanding all the time.

3.5 Brainstorming the implementation

We now know roughly what the end-user experience in our app will be like. Let's turn our attention to making that experience happen.

In this step of the process, we'll enumerate the various parts of our solution, discuss how they fit together, and map out the flow of logic.

There are two big parts to pretty much any application: a frontend and a backend. The frontend handles how the user interacts with your app, i.e., how we collect inputs and display outputs. The backend is the app's "brain" that takes inputs from the frontend, processes them, and hands the output to the frontend to display to the user.

Based on the last section, we already have a fair idea of what the frontend of our app looks like; we just need to translate the UI to the corresponding elements available in Streamlit, which we'll do in our code walkthrough.

We'll therefore turn our attention to the backend, starting with the actual unit conversion.

3.5.1 Performing the actual unit conversion

Let's say we want to convert 5 pounds (lbs) to ounces (oz). From an online search (I grew up with the metric system), 1 pound equals 16 oz.

Thus, to convert 5 lb to oz, we would multiply 16 by 5 to get 80 oz.

We'll refer to the number 16 here as the *lb-to-oz conversion factor*.

Every pair of units has a similar conversion factor. For instance, 1 yard equals 3 feet, so the yard-to-foot conversion factor is 3. To take an imperial-to-metric example, 1 mile equals 1.609344 km, so the mile-to-km conversion factor is 1.609344.

To state this generally, given any pair of units X and Y, to convert a given value expressed in X to Y,

$$\text{Value in unit Y} = \text{Value in unit X} * \text{X-to-Y conversion factor}$$

To do the reverse conversions (e.g., ounces to pounds), you can use a conversion factor calculated as 1 divided by the original conversion factor.

So,

1 yard = 3 feet; 1 foot = $\frac{1}{3}$ yards; foot-to-yard conversion factor = $\frac{1}{3}$

1 mile = 1.609344 km; 1 km = 1/1.609344 = 0.621372 miles; km-to-mile conversion factor = 0.621372

With this information, it seems like all we need to do is to collect the conversion factor for every pair of possible units within a quantity type, and apply the formula given above.

But there's a problem: we could end up with a lot of conversion factors to keep track of.

Let's say that we want users to be able to convert between 3 units: pounds, kilograms, and ounces

There are $3 \times (3-1) = 3 \times 2 = 6$ possible types of conversions that can be done: pounds to kilograms, kilograms to pounds, pounds to ounces, ounces to pounds, ounces to kilograms, and kilograms to ounces.

This means we need to keep track of 6 conversion factors. From our discussion above, we've seen that if we know the X-to-Y conversion factor, then the Y-to-X conversion factor is simply 1 divided by the X-to-Y conversion factor. Using this, we can whittle down the number of conversion factors we need to keep track of from 6 to $6 / 2 = 3$.

In general, if you have n units, to be able to convert between any pair of them, we would need $n(n-1) / 2$ conversion factors.

Our example above didn't seem too bad. But what if we have a lot more than 3 units of a particular quantity type? What if we have 20?

In this scenario, we would need to keep track of $20 * (20-1) / 2 = 20 * 19 / 2 = 190$ conversion factors. Whew! That's a *lot* of numbers to track in your code.

But isn't 20 units for a single quantity type a little extreme? Not really. Take distance or length. If you count the metric system's prefixes that signify magnitude and include units used in astronomy or navigation, you'll come up with something like the following list:

kilometers, meters, decimeters, centimeters, millimeters, micrometers, nanometers, angstroms, inches, feet, yards, miles, furlongs, astronomical units, parsecs, light years, nautical miles, fathoms, leagues, cubits, picometers, decameters, hectometers... you get the picture.

Clearly, keeping track of conversion factors for every single pair of units is not sustainable.

Instead, what we want to do is to designate one unit from each quantity as the *standard unit* for that quantity and only keep track of conversion factors to that unit from all the others.

For instance, if we make meters the standard unit for distance, we would only keep track of the conversion factor from each unit to meters, i.e. "what is the value 1 in this unit in meters?"

Then, to convert from unit X to unit Y, we could convert from unit X to meters and then from meters to unit Y.

So if we want to convert 5 yards to centimeters:

Yard-to-meter conversion factor = 0.9144

Centimeter-to-meter conversion factor = 0.01

We can then follow a two-step conversion process (see figure 3.8):

1. 5 yards = 5×0.9144 meters = 4.572 meters
2. 4.572 meters = $4.572 \times 1 / 0.01$ centimeters = 457.2 centimeters

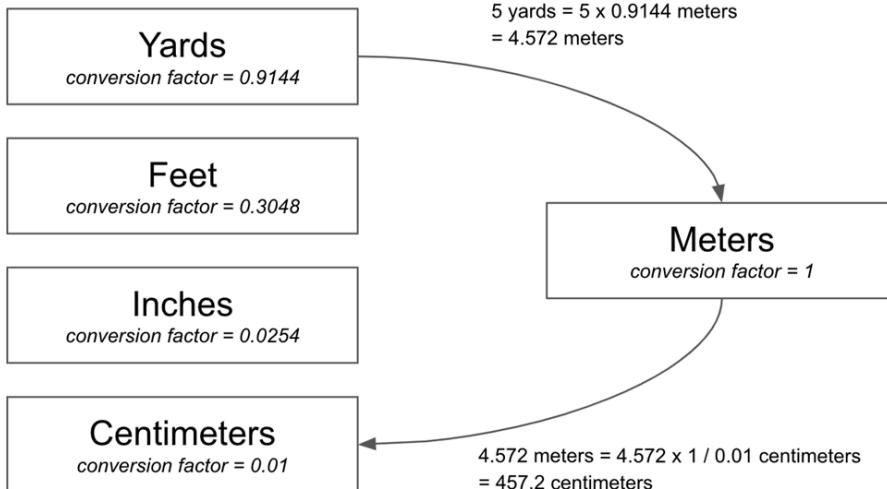


Figure 3.8 Converting from yards to centimeters using meters as a standard intermediate unit

Notice that since we always store the conversion factor of a unit to meters, in step 2, we had to divide 1 by the factor to get the reverse conversion factor, i.e., meter-to-centimeter.

Putting this more generally, given a pair of units X and Y and a standard unit S, to convert a value expressed in X to Y,

$$\text{Value in unit S} = \text{Value in unit X} * \text{X-to-S conversion factor}$$

$$\text{Value in unit Y} = \text{Value in unit S} * 1 / \text{Y-to-S conversion factor}$$

We now end up with a much more manageable number of conversion factors: since we only care about how many standard units a given unit is, we can get away with storing just one factor per unit.

3.5.2 Tracking units and conversion factors

Recall that one of our original requirements is that adding new units to the app must be straightforward.

When we speak of "adding new units" here, we're referring to us updating the app to include more conversion units to convert between, not the user being able to dynamically add units themselves.

This is an example of a *non-user requirement*. Our user presumably does not care about how easy it is for us to add new units; they care that the units are available for them to convert between. So who *does* care? The developer does. *We* do.

As a developer, you're usually never done with a piece of software once you've launched it. If your app is used by enough people, you can count on a consistent stream of feedback, reported bugs, and requested features coming your way and eating up the time that you were planning to spend on other projects. Or worse, someone *else* will be tasked with maintaining your code, and they'll have no idea how it works.

In this situation, if you haven't designed your app's implementation for ease of maintenance, someone will have to spend a significant amount of time digging through your code to figure out the right place to make a change and make sure there are no side-effects. Even if you're not feeling particularly charitable towards other developers, there's always the likelihood that the *someone* will be you. Speaking from personal experience, you'd be surprised at how little you'll remember about the code you wrote as recently as a month ago.

Coming back to our unit converter, one of the most common maintenance tasks you might need to perform is to add new units, and we want to make that as simple as possible. Ideally, there should be exactly one place in the code where you'd need to add the unit and its conversion factor, and it should be the most obvious place.

A good way to achieve this is to keep a master list of all quantities, units and conversion factors in a single configuration file. The file should have an obvious format, and adding a new unit should be as easy as appending a few lines in the file.

Critically, this configuration file should be the *only* file that refers to any specific quantities. That means that no other places in the app should refer to specific quantities or units, like meters or pounds. That's because if they did, adding a new quantity or unit would require updating that part of the code too, which breaks our requirement of it being straightforward to update the code.

This has an interesting implication: we can't hard-code quantities or units into our UI code at any point. Everything related to specific quantities or units needs to be pulled from the configuration file. The rest of our code needs to be independent of them.

We'll see how to do this in our code walkthrough later in the chapter.

3.5.3 Mapping the flow of logic

We now have a good understanding of all of the individual pieces of our app: the frontend, the conversion logic, and the configuration.

Before we code them up, it's useful to have a mental model of how they fit together. Figure 3.9 shows the overall design of our app.

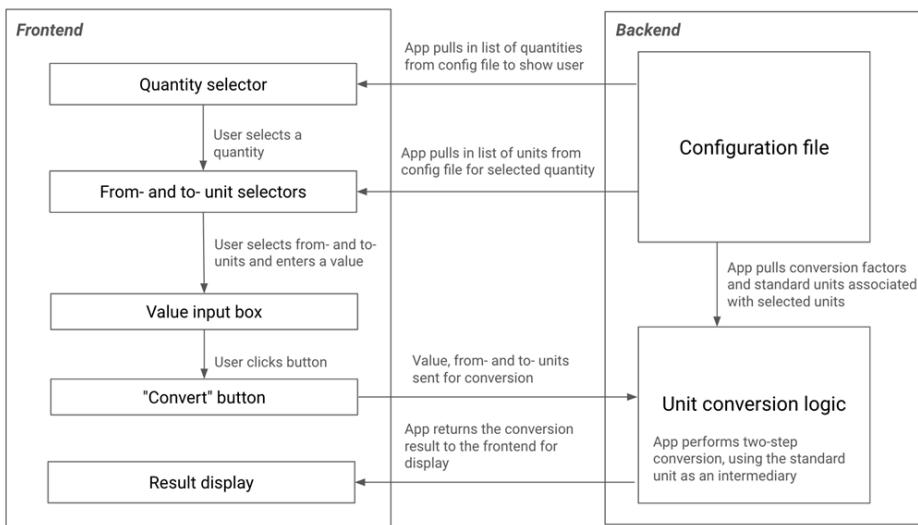


Figure 3.9 Overall design and logic flow for our unit conversion app

The configuration file that we discussed in the previous section powers the quantity selector radio buttons from the UI we visualized earlier (figure 3.10).

- Time
- Distance
- Mass
- Pressure

Figure 3.10 Quantity selector radio buttons from our mocks

Once the user has selected a quantity, the from- and to- unit dropdowns update based on the selected quantity (again powered by the configuration file).

When the user has selected the from- and to- units and clicked the "Convert" button, all entries are sent to the backend where the conversion factors from the configuration are used to perform the two-step conversion outlined in section 3.5.1.

The converted value is then returned to the frontend where the user can see it.

3.5.4 The backend API

There's a key concept in software development called *separation of concerns*. In essence, this means that each building block in a piece of software should focus on one aspect of the overall system's functionality, and should be independent of the other building blocks. Where a block interacts with another, it should do so in strictly controlled ways as defined by a *contract*.

To understand this better, consider how a drive-in fast food restaurant might operate. There's a person at the front taking orders from customers and a kitchen that prepares the food based on a menu. The order-taker doesn't care what ingredients the kitchen uses as long as it can make the items on the menu, and the kitchen doesn't care what language the order-taker speaks to the customer in, as long as the items ordered are from the menu.

If the kitchen wants to hire new cooks or use different ingredients, it can do so without informing the order-taker, as long as the dishes prepared match the menu. If we want to replace the in-person order taker with someone on the phone, we can do that too, without affecting the kitchen.

The thing that makes this possible is the menu, which is a common shared *contract* through which the order-taker and kitchen interact.

Similarly, it's a good idea while developing an app to separate its components and have them interact exclusively through a contract called an *Application Programming Interface* (API for short). In the context of our app, we should maintain this separation between the frontend and the backend.

This means that our frontend should only interact with the backend to ask it to perform any of a certain list of actions.

What exactly are these actions? Let's revisit our logic flow diagram in figure 3.11. There are four arrows between the frontend and the backend:

- one where the frontend pulls in the quantities to show the user,
- one where it pulls in the units corresponding to the quantity selected by the user,
- one where the frontend provides the values to be converted, and
- one where the backend returns the conversion value

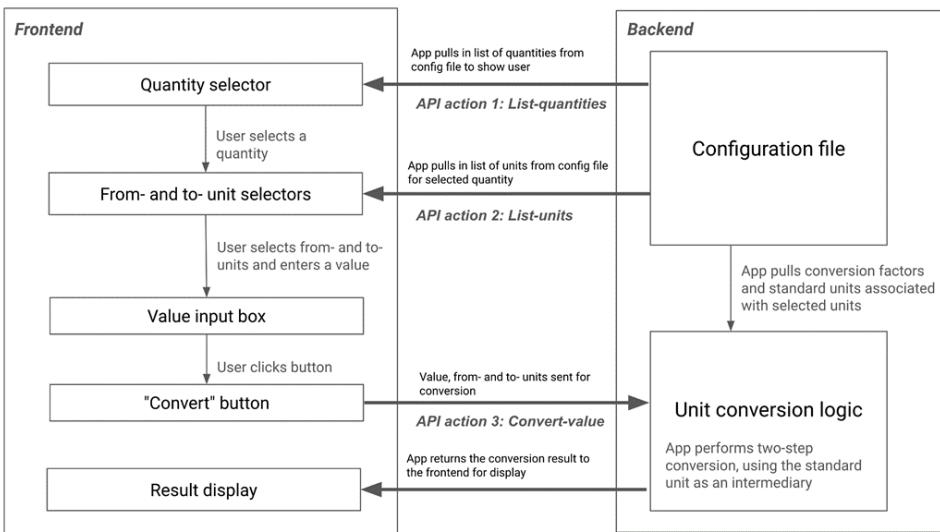


Figure 3.11 Flow diagram for our app with the API actions highlighted

These can be boiled down into three interactions (the last two arrows are really two parts of the same interaction) that the frontend has with the backend:

- **List-quantities:** The frontend asks the backend to provide the list of quantities it supports.
- **List-units:** The frontend gives the backend a quantity and asks it to list the units it knows how to convert between in that quantity.
- **Convert-value:** The frontend gives the backend the value to convert and the units selected by the user, and asks it to do the conversion.

The above actions form the menu or API of our backend. As in our fast food example, as long as the backend can fulfill these responsibilities, it's free to implement them however it wants. Perhaps at some point, we'll find a more efficient way to perform conversions. Or maybe want to hook up the backend to an external service to perform the conversion. In either case, we can change the backend implementation without touching the frontend code.

Or maybe at some point we'll want to enable a command-line interface to our conversion app in addition to our graphical UI. In that instance, we can simply add it with no changes required to our existing frontend code.

As you can see, separating concerns between the frontend and the backend gives us a lot of flexibility. Our app is simple enough that the benefits may not be obvious, but this is a good habit to develop as it'll help you in the real world when you develop more complex apps and need to switch out components easily.

We'll see how to actually implement this API approach very soon.

3.6 Writing code

At this point, we've spent enough time thinking about the design of our app that we're ready to write some code.

In this section, we'll walk through the code for our app, starting with our configuration file, which plays a central role in our design. We'll define a *contract* or *API* between our frontend and backend, then make our backend fulfill that contract.

Finally, we'll write our frontend, learning about the Streamlit elements that make our UI possible as we do so, and have it issue calls to the backend through our defined contract to close the loop.

3.6.1 Creating the configuration file

Our configuration file is what holds information about the quantities and units our app understands.

From our design, we know that the app needs to look up the units for a given quantity. Furthermore, given a unit, it needs to be able to look up its conversion factor with respect to the standard unit for that quantity.

A Python dictionary sounds like it would be ideal for this task. Each key in this dictionary would be a quantity, and the associated values would be a list of units, or better yet, *another* dictionary where the keys are units and values are the corresponding conversion factors.

One possible way to write this might be something like the following:

```
unit_config = {
    "Mass": {
        "Kilograms": 1, # Standard unit
        "Grams": 0.001,
        "Pounds": 0.453592,
        "# ...
    },
    "Length": {
        "Meters": 1, # Standard unit
        "Centimeters": 0.01,
        "# ...
    },
}
```

This certainly works, but it would be nice if we could also display the abbreviation for a unit (e.g., 'oz' for ounces or 'kg' for kilograms), and maybe we should store the standard unit for a quantity as well.

We're getting to the point where units and quantities are well-defined "things" with their own attributes (a unit has an abbreviation and a conversion factor, a quantity has a bunch of units and one unit that's designated as the standard one). We *could* represent this complexity by expanding our dictionary to have more "layers," but it would be better practice to define *classes* for units and quantities.

So spin up your code editor and create a file called `unit.py` (see listing 3.1) to define the Unit class. We're going to use a *dataclass* here, which is a special type of class in Python, enabled through the `dataclasses` module, which is part of Python's standard library.

Listing 3.1 unit.py

```
from dataclasses import dataclass

@dataclass
class Unit:
    abbrev: str
    value_in_std_units: float
```

A dataclass is an easy way to create a class with some standard basic functionality. For instance, with dataclasses, you don't need to specify a `__init__` method as you would do with a normal class, and once you have an object of the dataclass, you can access its attributes with the dot-notation, like this:

```
gram = Unit(abbrev="g", value_in_std_units=0.001)
print(gram.abbrev) # Prints 'g'
```

To achieve the same functionality with a regular class, you would have had to write:

```
class Unit:
    def __init__(self, abbrev, value_in_std_units):
        self.abbrev: str = abbrev
        self.value_in_std_units: float = value_in_std_units
```

With dataclasses, you can use the much more concise syntax from Listing 3.1 by including the `@dataclass` decorator above your class definition.

Turning to the contents of the class, you can see that they're pretty simple; there's an `abbrev` string field that holds the abbreviation for the unit, and a `value_in_std_units`, a floating point number which represents the conversion factor we discussed in Section 3.5.1.

The `: str` and `: float` from Listing 3.1 are called *type annotations*. They're used to specify the data type of a field. Type annotations are not strictly required, but it's a good practice to include them, since (among other things) they make your code easier to understand and allow your code editor or IDE to catch and highlight errors early on.

You may have noticed that we haven't included a `name` field in the `Unit` class. We'll discuss why when we get to the configuration file.

Let's also define a `Quantity` class in a new file, `quantity.py`.

Listing 3.2 quantity.py

```
from dataclasses import dataclass
from typing import Dict

from unit import Unit

@dataclass
class Quantity:
    std_unit: str
    units: Dict[str, Unit]
```

Listing 3.2 shows the `Quantity` class. It contains two fields: `units` and `std_unit`.

`std_unit` here is the name of the standard unit for that quantity.

Notice the more complex type annotation we're using for the `units` field. `Dict[str, Unit]` means that `units` is a dictionary where each key is a string (the name of the unit) and the corresponding value is an object of the `Unit` class. Annotations for some of the more advanced types need to be imported from the `typing` module.

With that out of the way, we can now create our final configuration file (see Listing 3.3).

Listing 3.3 unit_config.py

```
from typing import Dict

from quantity import Quantity
from unit import Unit

unit_config: Dict[str, Quantity] = {
    "Mass": Quantity(
        std_unit="Kilograms",
        units={
            "Kilograms": Unit(abbrev="kg", value_in_std_units=1),
            "Grams": Unit(abbrev="g", value_in_std_units=0.001),
            "Pounds": Unit(abbrev="lb", value_in_std_units=0.453592),
```

```

    "Ounces": Unit(abbrev="oz", value_in_std_units=0.0283495),
    # Add more units here
}
),
"Length": Quantity(
    std_unit="Meters",
    units={
        "Meters": Unit(abbrev="m", value_in_std_units=1),
        "Centimeters": Unit(abbrev="cm", value_in_std_units=0.01),
        "Inches": Unit(abbrev="in", value_in_std_units=0.0254),
        "Feet": Unit(abbrev="ft", value_in_std_units=0.3048),
    }
),
"Time": Quantity(
    std_unit="Seconds",
    units={
        "Seconds": Unit(abbrev="s", value_in_std_units=1),
        "Minutes": Unit(abbrev="min", value_in_std_units=60),
        "Hours": Unit(abbrev="hr", value_in_std_units=3600),
        "Days": Unit(abbrev="d", value_in_std_units=86400),
    }
),
# Add more quantities here
}

```

Our configuration is still in the form of a dictionary, but each dictionary value is now an object of the `Quantity` class. Our file includes three quantities: mass, length, and time.

Let's inspect one of these:

```

"Mass": Quantity(
    std_unit="Kilograms",
    units={
        "Kilograms": Unit(abbrev="kg", value_in_std_units=1),
        "Grams": Unit(abbrev="g", value_in_std_units=0.001),
        "Pounds": Unit(abbrev="lb", value_in_std_units=0.453592),
        "Ounces": Unit(abbrev="oz", value_in_std_units=0.0283495),
        # Add more units here
    }
)

```

As you can see, this is quite readable. We're configuring a quantity named "Mass" as a `Quantity` object with a standard unit of "Kilograms." The `units` dictionary has four entries, one for each unit, where the value is a `Unit` object with an abbreviation and conversion factor.

Since "Kilograms" is the standard unit, its `value_in_std_units` is 1.

You may now also realize why neither of the `Unit` and `Quantity` classes have a `name` field. Since the unit and quantity names are already incorporated into the dictionary keys in `unit_config.py`, also having it in the class is unnecessary and would make the configuration file longer and less readable.

Adding more units is easy; you would just add new entries to the `units` dictionary within the `Quantity`. Similarly, adding new quantities simply requires appending to the `unit_config` dictionary, following the format of the previous entries.

3.6.2 Implementing the API

Earlier in the chapter (section 3.5.4), we enumerated the actions we want the backend to be able to take:

- List-quantities
- List-units
- Convert-value

Let's now actually implement these.

We want each of the above actions to be a function in the backend. For each, we'll come up with the function signature first, and then implement it.

List-quantities is pretty simple; it asks the backend to list all the quantities it knows. There are no arguments required, and the output would probably be a list of strings. Thus:

```
def list_quantities() -> List[str]
```

How do we implement this? Well, the quantities we need are the keys of our configuration object from `unit_config.py` (in listing 3.3). So we can simply write:

```
def list_quantities() -> List[str]:
    return list(unit_config.keys())
```

List-units does take an argument (the quantity to list the units of), and it would again return a list of strings. So we get:

```
def list_units(quantity_name) -> List[str]
```

Implementing this is also fairly straightforward. The list we need is composed of the keys of the `units` dictionary from the `Quantity` object corresponding to the `quantity_name` key in `unit_config`.

```
def list_units(quantity_name) -> List[str]:  
    return list(unit_config[quantity_name].units.keys())
```

NOTE You might realize that we're not handling the scenario where the provided `quantity_name` doesn't exist in `unit_config`. In the real world, we should absolutely do that, but for this chapter, I've excluded error handling to keep the code relatively concise. We'll tackle error handling in future chapters.

Convert-value takes four arguments: a quantity name, the from- and to- units, and a value to convert. We want to include the quantity name here due to the possibility that different quantities may have a unit that's called the same (though our example configuration doesn't have this).

As for the return type, we *could* simply return the converted value, which would be a floating point number, and nothing else.

Recall, however, that our configuration also has abbreviations for each unit. It would be nice if, on conversion, we were also able to give the frontend the appropriate abbreviations so it can display something like "15 ft = 5 yd."

On the other hand, we don't want to be prescriptive about how the frontend *actually* displays the results; that's the frontend's business—remember *separation of concerns*? If the frontend wants to just show the converted number with no abbreviation, that's totally fine too.

One approach would be to wrap any metadata the frontend might need in a dedicated `Result` class and let the frontend figure out what to do with it.

Let's define the `Result` class in a new file called `result.py` (listing 3.4).

Listing 3.4 result.py

```
from dataclasses import dataclass
from unit import Unit

@dataclass
class Result:
    from_unit: Unit
    to_unit: Unit
    from_value: float
    to_value: float
```

Note that rather than placing just the abbreviation for each unit in the result, we're including the entire `Unit`. If we decide to modify the `Unit` class later and add more properties to it, this code won't need to change.

With that, we're ready to define the signature for **Convert-value**:

```
def convert_value(
    quantity_name: str,
    from_unit_name: str,
    to_unit_name: str,
    value: float) -> Result
```

We've already discussed how to implement the conversion, but here it is in code form:

```
def convert_value(
    quantity_name: str,
    from_unit_name: str,
    to_unit_name: str,
    value: float) -> Result:
    quantity = unit_config[quantity_name]
    from_unit = quantity.units[from_unit_name]
    to_unit = quantity.units[to_unit_name]

    # Two-step conversion: from-unit to standard unit, then to to-unit
    value_in_to_units = (value *
        from_unit.value_in_std_units /
        to_unit.value_in_std_units)

    return Result(from_unit, to_unit, value, value_in_to_units)
```

Notice the two-step conversion we discussed earlier. `value * from_unit.value_in_std_units` gives the value in standard units, and the `/ to_unit.value_in_std_units` converts it to the to-unit.

[Listing 3.5](#) puts this all together in a single `backend.py` file.

Listing 3.5 backend.py

```
from unit_config import unit_config
from result import Result
from typing import List

def list_quantities() -> List[str]:
    return list(unit_config.keys())

def list_units(quantity_name) -> List[str]:
    return list(unit_config[quantity_name].units.keys())

def convert_value(
    quantity_name: str,
    from_unit_name: str,
    to_unit_name: str,
    value: float) -> Result:
    quantity = unit_config[quantity_name]
    from_unit = quantity.units[from_unit_name]
    to_unit = quantity.units[to_unit_name]

    # Two-step conversion: from-unit to standard unit, then to to-unit
    value_in_to_units = (value *
                         from_unit.value_in_std_units /
                         to_unit.value_in_std_units)

    return Result(from_unit, to_unit, value, value_in_to_units)
```

3.6.3 Implementing the frontend

We're finally in the portion of this chapter where you'll actually be using Streamlit!

First we'll explore each of the features of Streamlit we'll be using, referring to our UI design from earlier (shown in figure 3.12), and then we'll use them to create our frontend.

The screenshot shows a Streamlit application interface. On the left, there is a vertical list of radio buttons for selecting a quantity: Time, Distance, Mass, and Pressure. The 'Mass' button is currently selected. To the right of this list is a 'Value' input field containing the number '13'. Below the value input is a 'From unit' dropdown menu with 'Pounds' selected. To the right of the 'From unit' dropdown is a 'To unit' dropdown menu. This 'To unit' menu lists several units of measurement: Kilograms, Grams, Ounces, Tons, and Milligrams. An arrow points from the text 'Milligrams' in the dropdown menu towards the 'Convert' button. Below the dropdowns is a 'Convert' button. At the bottom of the interface, the converted value is displayed as '13 lb = 5.89 kg'.

Figure 3.12 The UI visualization for our unit converter app for reference

We're going to be building our UI incrementally. To start, create a new Python file called `frontend.py` and add the imports we need (i.e., Streamlit itself, and the backend API functions we defined).

```
import streamlit as st
from backend import convert_value, list_quantities, list_units
```

Save your file and run the following in the terminal:

```
streamlit run frontend.py
```

or `streamlit run <path to frontend.py>` if your working directory is not the same one that contains `frontend.py`.

This will open a browser window with your app (currently just a blank screen). Each time you make a change, switch back to the browser window and click "Rerun" or "Always rerun" to see the results.

ST.RADIO

The first component of the UI we'll focus on is the quantity selector, which is a bunch of radio buttons set in a panel on the left.

Radio buttons are a UI element that lets a user pick a single item out of a given list of items.

For example, if you append the following to the `frontend.py` file you just created:

```
quantity = st.radio("Select a quantity", ["Mass", "Force", "Pressure"])
```

Streamlit will display the question "Select a quantity" and a list of radio buttons with "Mass," "Force," and "Pressure" as options. Once the user has selected one, the variable `quantity` will contain the option that the user picked (i.e., the string "Mass," "Force," or "Pressure").

In our case, our list of options will come from the backend where you'll recall that we have a function called `list_quantities` (listing 3.5). So we would instead write:

```
quantity = st.radio("Select a quantity", list_quantities())
```

This gives us the output shown in figure 3.13. We now get "Mass," "Length," and "Time" as options because `list_quantities` fetches the keys of `unit_config` (from our configuration file) and returns them as a list that forms the second argument to `st.radio`.

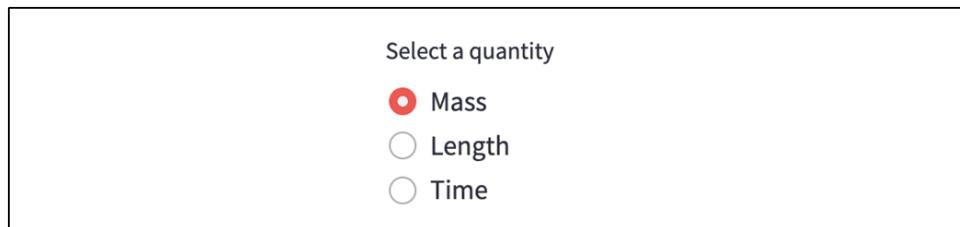


Figure 3.13 An example output for `st.radio`

The `quantity` variable will contain the item currently selected by the user (e.g., "Mass" in figure 3.8).

Streamlit offers plenty of customization for `st.radio`. You can set the options horizontally instead of vertically, add a caption to each option, disable them altogether, and so on.

You can find a complete list of customization possibilities in the Streamlit docs at <https://docs.streamlit.io/develop/api-reference/widgets/st.radio>.

ST.SIDEBAR

We've created our radio buttons, but we now need them in a panel to the left as shown in figure 3.12.

This kind of panel, in Streamlit terminology, is a *sidebar*. Sidebars are extremely useful when you want to create a bunch of links that lead to different pages in the app, provide some meta-information about your app, etc.

To use `st.sidebar`, you need to put something within it. There are two ways to do this:

You can use a context manager (i.e., Python `with` statement) like this:

```
with st.sidebar:
    quantity = st.radio("Select a quantity", list_quantities())
```

Any Streamlit element you place inside the `with` statement will be displayed in the sidebar.

You can also use a dot notation and refer to an element you want to place inside the sidebar as a member of the sidebar, like so:

```
quantity = st.sidebar.radio("Select a quantity", list_quantities())
```

Appending either of the above to `frontend.py` produces the output shown in figure 3.14.



Figure 3.14 `st.sidebar` in action with a set of radio buttons inside

A sidebar is displayed with an 'X' icon that collapses it or a '>' icon that expands a collapsed sidebar.

ST.TITLE

With the sidebar in place, let's turn our attention to the main area of the app. Users need to know what the app is and what it does, so we'll add a title:

```
st.title("Unit Converter")
```

This should be pretty self-explanatory, but just to state it explicitly, `st.title` writes any string passed to it as a title, i.e., in large bold font.

ST.TEXT_INPUT

Next, we need the user to enter the value they want to convert. Let's use a text input for this.

`st.text_input` is Streamlit's way of letting users enter single-line values. We can write:

```
input_num = st.text_input("Value to convert", value="0")
```

to display a text input with the caption "Value to convert" and an initial value of "0." Note that `st.text_input` returns a string, so `input_num` has the string "0."

We want to maintain this as a number, so we'll also cast the value to float:

```
input_num = float(st.text_input("Value to convert", value="0"))
```

With the title and text input, our app should look like figure 3.15.

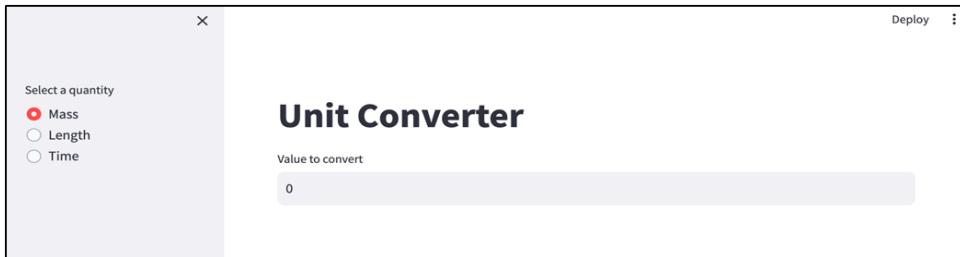


Figure 3.15 Unit Converter app after adding a title and text input

`st.text_input` has lots of customization options, such as placeholder text that's displayed when there's no entered value, an optional tooltip, and the ability to enter passwords (we encountered this in our password checker app in Chapter 2).

Again, docs.streamlit.io has more detail on what's available.

ST.SELECTBOX

We need dropdowns for the user to pick the from- and to- units, so let's create those next.

`st.selectbox` is what we're looking for here. It displays a basic select widget with a label and a set of options.

The arguments are similar to what you would pass to `st.radio`. For example, we could write:

```
country = st.selectbox("Pick a country", ["United States", "Canada", "India"])
```

to display a country dropdown with the variable `country` containing the selected option. When the dropdown first renders, the first option ("United States" in this case) is selected by default.

For our use case, we need to first collect the list of options to display. We can call `list_units` from our backend to fetch the units available for the quantity the user has selected:

```
units = list_units(quantity)
```

We can then use this list to populate the options in our from- and to- unit dropdowns:

```
from_unit = st.selectbox("From", units)
to_unit = st.selectbox("To", units, index=1)
```

The `index` parameter we've included in the second dropdown is used to set the default selected option. A value of '1' means the second option will be selected on the first render.

We do this because we don't want both the from- and to- dropdowns to have the same default selected value, since that would almost never be useful (users wouldn't want to convert from kilograms to kilograms).

At this point, our app looks like figure 3.16.



Figure 3.16 Unit Converter app after adding from- and to- unit dropdowns

This is *fine*, but our design had the from- and to- dropdowns side by side, which seems more natural.

ST.COLUMNS

By default, Streamlit displays the UI elements from top-to-bottom in the same order that it encounters them in your code.

Naturally, you don't always want this; sometimes you want things laid out side by side. We saw how to do this with `st.sidebar`, but an app can only have one sidebar, and it appears to the left of the overall UI; it can't be inline.

`st.columns` is the answer here. To use it, you would first create a list of columns, specifying the number of columns you want:

```
from_unit_col, to_unit_col = st.columns(2)
```

Here, the call to `st.columns(2)` returns a list of two columns. The syntax we're using here is called *list unpacking*, and it assigns the individual items of a list to different variables. As you might imagine, here `from_unit_col` will contain the first column and `to_unit_col` will contain the second.

As in the case of `st.sidebar`, there are two ways to put something in a column: using the `with` context manager or the dot notation. So we could write:

```
with from_unit_col:
    from_unit = st.selectbox("From", units)
with to_unit_col:
    to_unit = st.selectbox("To", units, index=1)
```

or more concisely,

```
from_unit = from_unit_col.selectbox("From", units)
to_unit = to_unit_col.selectbox("To", units, index=1)
```

Generally speaking, the `with` context manager makes more sense when you have multiple elements to display within a container (whether it's a sidebar, a column, or something else), while the dot notation works better when you only have a single item or when you want to display elements out of order. We'll see plenty of examples of these cases throughout this book.

Figure 3.17 shows our app by this point:

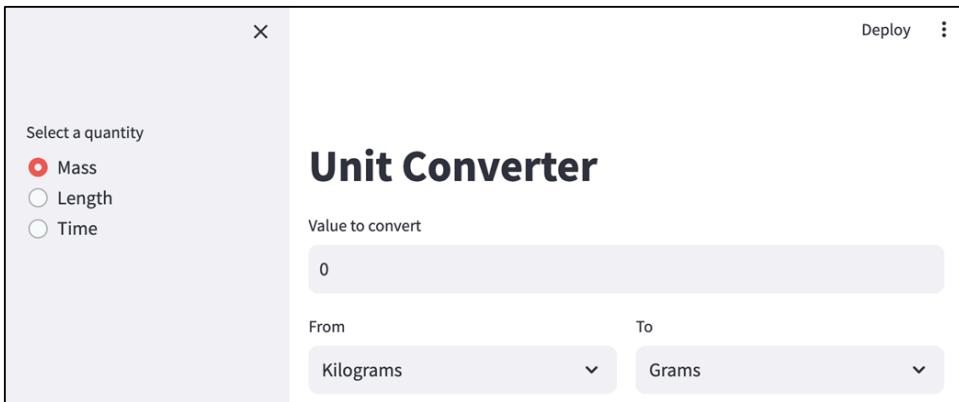


Figure 3.17 Unit Converter app with side-by-side from- and to- unit dropdowns

ST.BUTTON

With all the inputs captured, we're ready to add our "Convert" button.

I hope you remember `st.button` from Chapter 2, Streamlit's ol' faithful make-something-happen element.

To add our button, we would write:

```
if st.button("Convert"):
    # Statements to execute
```

This should be easy enough to understand. It says, "display a button that says 'Convert' and if/when the user clicks it, execute some statements."

Once we've written out what we want the button to do, this will render a barebones Streamlit red-on-white button with the functionality we've defined.

So what do we actually want our button to do? Our backend has a `convert_value` function that performs the unit conversion, so let's start by calling that:

```
if st.button("Convert"):
    result = convert_value(quantity, from_unit, to_unit, input_num)
```

Here, we're passing `convert_value` the user-selected quantity, the from- and to- units, and the value to convert. Recall that `convert_value` returns an object of the class `Result` (defined in `result.py`)

The variable `result` thus has the result of our conversion, complete with the from- and to- values and abbreviations. All that remains is to display it on the screen. Enter...

ST.METRIC

We could have displayed our result as plain old paragraph text, but this is the big outcome of our app we're talking about, the grand finale. We want something that packs more *punch*.

`st.metric` is a widget that's commonly used in dashboards to show important numbers—like the revenue of a company—and how they're trending as compared to a prior period.

A single `st.metric` element is meant to represent a measure the user interested in, and it consists of three parts: a text label, the number itself displayed in large font, and a "delta indicator," which is a value that shows how much the number has increased or decreased from a prior period.

We're going to use `st.metric` to display the from- and to- values along with the abbreviations of the units, so let's first prepare these:

```
from_display = f"{result.from_value} {result.from_unit.abbrev}"
to_display = f"{result.to_value} {result.to_unit.abbrev}"
```

Since `result` is an `Result` object, we form the display text by concatenating its `from_value` or `to_value` field and grabbing the abbreviation from `from_unit` or `to_unit`, which is itself an instance of `Unit`.

To actually use `st.metric`, we write:

```
st.metric("From", from_display, delta=None)
st.metric("To", to_display, delta=None)
```

The `delta` indicator in `st.metric` doesn't make sense for us, so we set it to `None` to hide it.

The above will display the from- and to- results vertically, but we need them side-by-side, so let's use `st.columns` again:

```
from_value_col, to_value_col = st.columns(2)
from_value_col.metric("From", from_display, delta=None)
to_value_col.metric("To", to_display, delta=None)
```

Figure 3.18 shows our completed app.

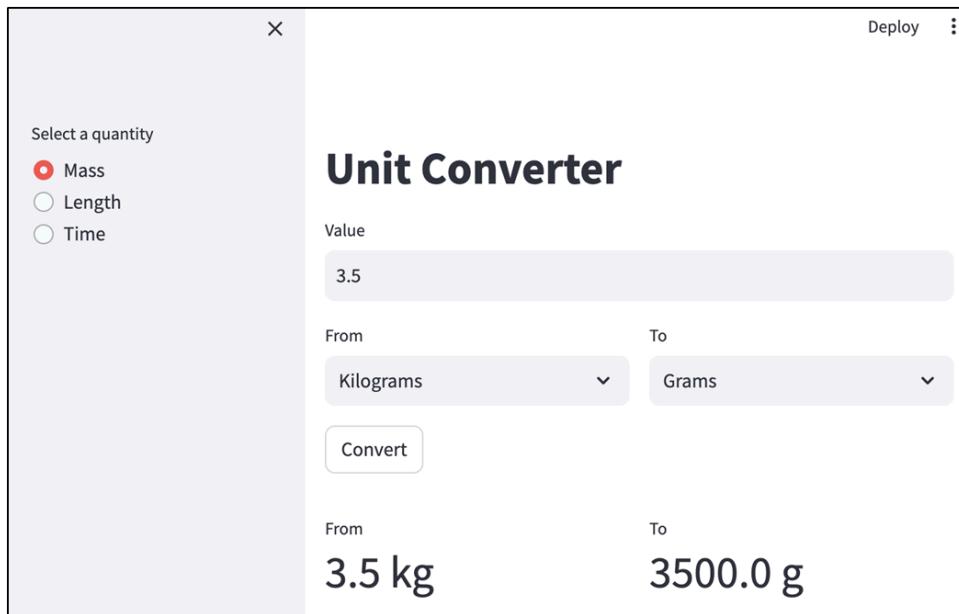


Figure 3.18 Our completed unit converter app

Listing 3.6 shows the `frontend.py` file you should have ended up with if you've been following along.

Listing 3.6 frontend.py

```

import streamlit as st
from backend import convert_value, list_quantities, list_units

quantity = st.sidebar.radio("Select a quantity", list_quantities())

st.title("Unit Converter")
input_num = float(st.text_input("Value to convert", value="0"))

units = list_units(quantity)
from_unit_col, to_unit_col = st.columns(2)
from_unit = from_unit_col.selectbox("From", units)
to_unit = to_unit_col.selectbox("To", units, index=1)

if st.button("Convert"):
    result = convert_value(quantity, from_unit, to_unit, input_num)
    from_display = f"{result.from_value} {result.from_unit.abbrev}"
    to_display = f"{result.to_value} {result.to_unit.abbrev}"

    from_value_col, to_value_col = st.columns(2)
    from_value_col.metric("From", from_display, delta=None)
    to_value_col.metric("To", to_display, delta=None)

```

3.7 Iterating on our app

Whew! We did it! We now have a fully functional app on our hands. In the real world, this would just be the *start* of your journey and you would now launch your app and show it to users.

Users will often have very opinionated feedback about the experience you've built for them and can help you uncover usability issues and blind spots you may not have encountered in your testing.

In this section, we'll simulate this process by using the app ourselves and identifying potential improvements we can make.

3.7.1 Rounding our conversion results

Let's take our completed app for a spin. Figure 3.18 shows the example results of converting kilograms to grams.

That looks mostly fine, but let's try a metric-to-imperial conversion now. Say we want to convert 4000 kilograms to pounds. We fire up our app, choose "Mass," enter our inputs and hit "Convert" to see figure 3.19.

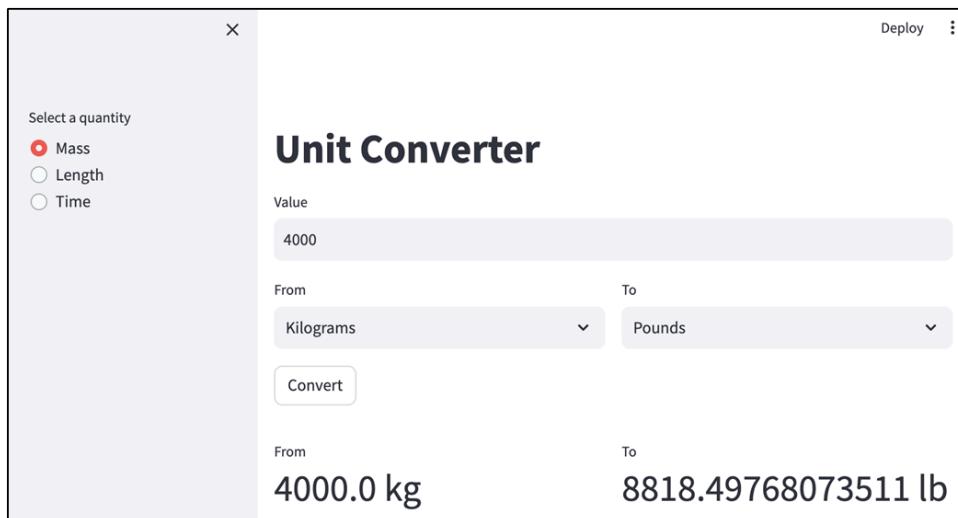


Figure 3.19 A metric-to-imperial conversion that shows why we should be rounding

The answer we got seems right, but that's an awful lot of decimal places. Most people probably don't need that level of precision; in fact it probably detracts from their experience as it takes them a second to figure out why the displayed number is so long.

It would be nice if users had the option of rounding the result to the precision they really need. And come to think of it, for larger numbers we should ideally separate the thousands by commas.

To implement this, let's first create a function called `format_value` that will format a number the way we want it, i.e., with commas separating the thousands, and optionally rounded to a certain number of decimal points.

`format_value` takes three arguments: the value to format, the abbreviation, and an optional number of `decimal_places` to round the value to. If we don't pass the latter (i.e., `is_rounded` is `false`), the function won't do any rounding.

```
def format_value(
    value: float,
    unit_abbrev: str,
    decimal_places: int = None) -> str:
    is_rounded = decimal_places is not None
    rounded = round(value, decimal_places) if is_rounded else value
    formatted = format(rounded, ",")
    return f"{formatted} {unit_abbrev}"
```

We use the `round` function to perform the actual rounding if we need to and Python's built-in `format` function to add the commas, using `,` as the *format spec*.

We could just hardcode the number of decimal places to round to, but ideally we should let the user decide, which means adding a new input widget.

ST.NUMBER_INPUT

`st.number_input` is Streamlit's numeric input widget. It's quite similar to `st.text_input`, but it has some additional functionality, such as the ability to specify a minimum and maximum value, and a step button that lets you increase or decrease the entered value by clicking.

We can use it to collect the user's preferred number of decimal places by adding this line to `frontend.py` right after the from- and to- unit dropdowns:

```
places = st.number_input("Decimal places to round to", value=2, min_value=0)
```

We specify a default value of 2, and a minimum value of 0 since we can't have a negative number of decimal places (trying to enter a lower value shows an error). `places` will hold the entered number.

This should give us the widget shown in figure 3.20.



Figure 3.20 `st.number_input`

Notice the '-' and '+' buttons that let you increase or decrease the value by 1. You can adjust the step-interval to something else by specifying the `step` parameter in `st.number_input`.

You may be wondering why we bothered to use `st.text_input` to collect the value-to-convert and cast to float when `st.number_input` was available. This is because at the time of writing, there doesn't seem to be an easy way to get rid of the '-' and '+' buttons in `st.number_input`. These make sense when we're collecting the number of decimal places, which is an integer in a very tight range, but the value-to-convert is virtually unbounded and there's no predefined step-interval that makes sense.

Okay, now that we've collected the decimal places, we're ready to apply our formatting when we display the results. We can do this by changing the `from_display` and `to_display` variables so that they use the `format_value` function we defined earlier:

```
from_display = format_value(input_num, result.from_unit.abbrev)
to_display = format_value(result.to_value, result.to_unit.abbrev, places)
```

We pass `places` (which we collect from the user) to round the `to_display` variable. We could have done this for `from_display` as well, but it's likely the user entered the precision they want to see in the `from`-value, so we don't want to mess with that.

This gives us figure 3.21.

The screenshot shows a Streamlit application titled "Unit Converter". On the left, there is a sidebar with the heading "Select a quantity" and three radio buttons: "Mass" (selected), "Length", and "Time". The main area has a title "Unit Converter" and a "Value" input field containing "4000". Below it are "From" and "To" dropdown menus set to "Kilograms" and "Pounds" respectively. A "Decimal places to round to" input field shows "3" with a minus and plus sign for adjustment. A "Convert" button is below these fields. At the bottom, the converted values are displayed: "From" is "4,000.0 kg" and "To" is "8,818.498 lb".

Figure 3.21 Unit converter app with a decimal places input

NOTE This won't pad `to`-values with extra trailing zeroes. For instance, if the `to`-value is a whole number, say 600, it will be displayed as "600.0" with just the one trailing zero.

The formatted result looks much nicer, but we've also added an extra numeric input that increases the user's cognitive load a little. Maybe we should only introduce the decimal-places input if the user asks for it.

ST.CHECKBOX

`st.checkbox` is a Streamlit checkbox, i.e., a box that you can, well, check. Please try to contain your shock, we have an app to ship.

Like `st.button`, `st.checkbox` is a *conditional* element; you can use an `if` statement to branch your logic based on whether it's checked or not.

Our use case is to let the user decide if they want to round the conversion result, which we can do by modifying how we obtain the value of the `places` variable:

```

places = None
if st.checkbox("Round result?", value=False):
    places = st.number_input(
        "Decimal places to round to", value=2, min_value=0)

```

The checkbox is unchecked by default since we're passing `False` to the `value` argument.

Notice that we've included the line `places = None` above the `st.checkbox` code. This is because, down below, we're referring to `places` outside of the scope of the `if st.checkbox` block, so we need to pass an initial value to it in case the user leaves the box unchecked.

Our app should now look like figure 3.22.

The screenshot shows a web-based unit converter application. On the left, there's a sidebar titled "Select a quantity" with three radio buttons: "Mass" (selected), "Length", and "Time". The main area is titled "Unit Converter". It has a "Value" input field containing "4000". Below it are "From" and "To" dropdown menus set to "Kilograms" and "Pounds" respectively. A checked checkbox labeled "Round result?" is present. Underneath is a "Decimal places to round to" input field with the value "3". A red-bordered "Convert" button is centered. At the bottom, the converted values are displayed: "4,000.0 kg" on the left and "8,818.498 lb" on the right.

Figure 3.22 Unit converter app with rounding enabled

Or if the user leaves "Round result?" unchecked, we get the full-precision treatment in figure 3.23.

The screenshot shows a Streamlit application titled "Unit Converter". On the left sidebar, there is a radio button group labeled "Select a quantity" with options "Mass" (selected), "Length", and "Time". The main area has a title "Unit Converter". A "Value" input field contains "4000". Below it, a "From" dropdown is set to "Kilograms" and a "To" dropdown is set to "Pounds". There is a checkbox for "Round result?" which is unchecked. A "Convert" button is present. At the bottom, the converted value "4,000.0 kg" is shown next to its equivalent "8,818.49768073511 lb".

Figure 3.23 Unit converter app with rounding disabled

3.7.2 Getting rid of the button

Our app works great now; it displays rounded output, but only if we want it to. We've added a click and a numeric input to enable this, however.

Is there perhaps a way to simplify this experience? Let's turn our attention to the "Convert" button. Do we really even need it?

All it does is trigger the conversion. But we don't really need an explicit trigger. Why not have the app *always* show the result based on the entered inputs? So if the user changes the value, they would immediately see the conversion result rather than having to click the button again.

That definitely seems like a more intuitive experience, so let's make it happen.

As it turns out, this is easy to do. Simply remove the line `if st.button("Convert"):` and move everything in that block outside.

Listing 3.7 shows the final version of `frontend.py`.

Listing 3.7 Final version of `frontend.py`

```
import streamlit as st
from backend import convert_value, list_quantities, list_units

def format_value(
    value: float,
    unit_abbrev: str,
    decimal_places: int = None) -> str:
    is_rounded = decimal_places is not None
    rounded = round(value, decimal_places) if is_rounded else value
    formatted = format(rounded, ",")
    return f"{formatted} {unit_abbrev}"
```

```
quantity = st.sidebar.radio("Select a quantity", list_quantities())

st.title("Unit Converter")
input_num = float(st.text_input("Value", value="0"))

units = list_units(quantity)
from_unit_col, to_unit_col = st.columns(2)
from_unit = from_unit_col.selectbox("From", units)
to_unit = to_unit_col.selectbox("To", units, index=1)

places = None
if st.checkbox("Round result?", value=False):
    places = st.number_input(
        "Decimal places to round to", value=2, min_value=0)

result = convert_value(quantity, from_unit, to_unit, input_num)
from_display = format_value(input_num, result.from_unit.abbrev)
to_display = format_value(
    result.to_value, result.to_unit.abbrev, places)

from_result_col, to_result_col = st.columns(2)
from_value_col, to_value_col = st.columns(2)
from_value_col.metric("From", from_display, delta=None)
to_value_col.metric("To", to_display, delta=None)
```

And figure 3.24 shows a final screenshot of our app.

Select a quantity

- Mass
- Length
- Time

Unit Converter

Value

From To

Kilograms Pounds

Round result?

Decimal places to round to

- +

From To

4,000.0 kg 8,818.4977 lb

Figure 3.24 Unit converter app with the "Convert" button removed

You'll find that removing the button makes the app *flow* a lot better. For instance, if you change the decimal places or the value, you see the results instantly.

I hope this chapter has been a lot of fun! We started with a mere concept, broke it down into concrete requirements, designed a UI, thought deeply about the implementation, wrote code to turn our ideas into a working app, and refined it for a better user experience.

The point of this chapter was to give you a sense of what it's like to develop Streamlit apps in the real world while working with real stakeholders. As you've hopefully learned, there's a lot more to the process than writing code; things like nailing the requirements and designing (and refining) the user experience are equally important.

3.8 Summary

- Building an app from a concept is about a lot more than writing code.
- The six steps involved in creating a Streamlit app in the real world are: state the concept, define the requirements, visualize the user experience, brainstorm implementation, write code, and iterate.
- Requirements can come from your users or from non-user stakeholders.
- It's a good idea to visualize the user experience at the start by sketching mocks so you have something to work towards.
- Brainstorming the implementation involves analyzing tradeoffs and mapping the flow of logic.
- Separating your frontend and backend code and defining an API for the two to interact is a great way to organize your app.
- `st.text_input` and `st.number_input` allow users to input text and numeric values.
- `st.radio` and `st.selectbox` let users select a value from a list.
- `st.sidebar` and `st.columns` are layout elements that let you break the natural top-to-bottom way that Streamlit renders UI elements.
- `st.button` and `st.checkbox` are both conditional elements.

4 Streamlit's execution model

This chapter covers

- Creating apps that require maintaining state between page updates
- Troubleshooting your apps effectively
- The all important `st.session_state` and `st.rerun`
- Streamlit's execution model

In the last two chapters, you've gotten your feet wet with Streamlit by building two fully-functional apps: a password checker and a unit converter. You've learned the basics of Streamlit's syntax and how to create interactive elements. But what happens behind the scenes when you run a Streamlit app? Understanding this is key to building more complex applications.

This chapter delves into the heart of Streamlit's execution model, where we'll explore how to manage an app's state.

This chapter also takes a slightly different approach than the previous ones. While we'll still build a practical application – a daily to-do list app – the primary focus is on equipping you with troubleshooting skills. We'll intentionally introduce some bugs into the app to simulate real-world situations where things might not go according to plan. By following along and fixing these issues, you'll gain a deeper understanding of Streamlit's inner workings and how to debug your own apps effectively.

4.1 A more complex app: Daily to-dos

Ever juggled multiple deadlines at work while mentally planning a vacation for your family, while *also* trying to remember to buy bread on the way home? Regardless of your specific situation, the frenzy of modern life has a way of getting you in its grip, pulling you into a whirlwind of endless activity and demands. Hopefully this chapter's Streamlit app helps you manage the chaos even if it can't actually deliver the bread to your doorstep.

We're going to be making a to-do list app that lets a user track the various things they have to get done in a day.

Since the primary point of this chapter is to get you familiar with Streamlit's execution model, we won't go through the entire six-step development process in detail as we did in the last chapter.

Instead, we'll breeze through the concept, the requirements and a mock design, and then jump straight to implementation.

4.1.1 Stating the concept

As you hopefully remember from the last chapter, the concept is a succinct statement of what our app is. Here it is for our app:

CONCEPT A Streamlit app that lets users add tasks to a daily to-do list and track their status.

That seems pretty crisp and clear, so let's dive into the detailed requirements.

4.1.2 Defining the requirements

To recap from Chapter 3, while the concept provides a general idea of your app, it's your requirements that make it concrete, laying out what the user needs from it.

The requirements for the to-do list app we're building are the following:

REQUIREMENTS:

The user should be able to:

- view their daily to-do list, made up of tasks
- add a task to their to-do list
- remove a task from their list
- mark a task in their list as done
- undo marking a task as done
- see their overall task completion status, i.e., their total number of tasks and the number they've completed

It's just as important, if not even more so, to clarify what our app *won't* do, so let's specify that as well:

WHAT'S OUT OF SCOPE

- Retrieving a to-do list when the user refreshes or reopens the page
- Exporting a to-do list to an external file or format
- Saving the history of added or completed to-dos

The two lists above should give you a sense of what we're trying to build in this chapter: it's a fairly basic daily to-do list that lives entirely within a single browser session.

Essentially, the way we expect the user to interact with our app is to open it in a browser window at the beginning of the day, add their tasks, and mark them as done/not done as their day progresses, *keeping the window open until the end of the day and never refreshing it*. Rinse and repeat the next day.

We're not going to build the ability to *persist* (or save) any tasks outside of the browser session. If you refresh the page, you lose your data.

"Doesn't that limit the usefulness of the app somewhat?" you might ask. Absolutely. It's just that we don't want to introduce the complexity of external storage just yet. We'll explore that later on in the book, especially in Part 2.

However, one might also argue that giving users a blank slate at the start of each day makes them *more* productive. So you see, not being able to save your tasks is a *feature*, not a bug!

It's mostly the complexity thing. Still, you should know that spinning the limitations of your product into positives is practically a survival skill in the industry! I bet your other frontend tech manuals don't also give you free life advice.

4.1.3 Visualizing the user experience

We now know with a fair amount of precision what our app needs to be able to do, so with that in mind and keeping with the principle we introduced in the last chapter of putting the user experience front and center, let's turn our attention to the mock UI design shown in figure 4.1.

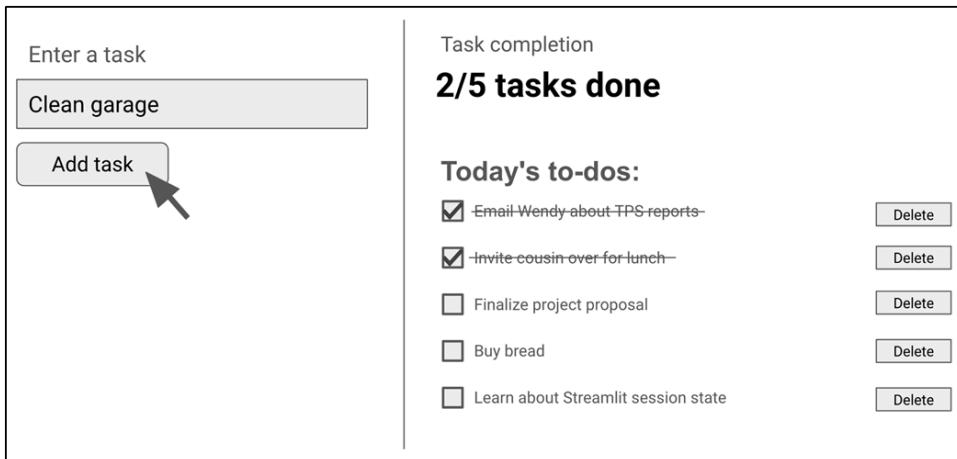


Figure 4.1 The mock UI design for our daily to-do list app

Our design has two sections: a sidebar where you can enter new tasks, and a "main" area where you can view the tasks you've added and update their status.

Once you add a task by entering the task text and clicking the button on the left, it appears on the right. Each task is rendered as a checkbox. You mark a task as "done" by checking the box, which also satisfyingly strikes through the task. You can delete a task entirely by clicking the button to its right.

There's also a tracker at the top which tells you how many tasks you've completed out of the total.

4.1.4 Brainstorming the implementation

You may have realized that our to-do list app is somewhat more complex than the password checker we built in Chapter 2 or even the unit conversion app in Chapter 3. In both of those cases, there was ultimately a single primary action the user could take—to evaluate the entered password in the case of the former, and to perform the conversion in the case of the latter.

Our to-do list has *four* different actions the user can take:

- add a task
- mark it as done
- mark it as not done
- delete it

Let's take a moment to brainstorm how we're going to make this happen.

Central to our implementation is the notion of a *task*, and by extension a *task list*. For the purposes of our app, a task is an object with two properties: a name, and a status which can be "done" or "not done." A task list is simply an ordered list of tasks.

The four user actions mentioned above are simply different ways in which you can modify the task list; adding a task adds an item to the list, marking it as done/not done updates the status of an item in the list, and deleting a task removes it from the list.

At every point, the app should show the latest state of the task list to the user.

We can therefore divide our app into three parts:

- The task list
- Actions, which are wired up to buttons and checkboxes, and modify the task list
- Display logic, which renders the task list on the screen

Whenever an action is performed, the task list is modified, and the display logic automatically updates what's shown on the screen.

Figure 4.2 shows what happens when a new task is added; the task is appended to the task list, and the display logic loops through all the tasks again and renders them on the screen based on a number of rules such as "strikethrough if done."

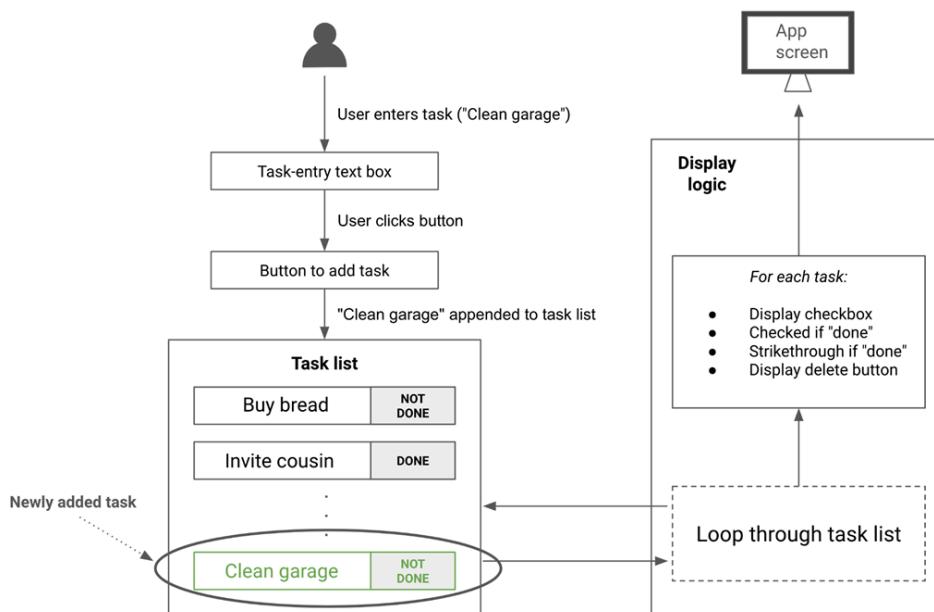


Figure 4.2 Adding a task appends an item to the task list and the display logic renders the updated task list

Something very similar happens when a task is checked off, as shown in figure 4.3. This time, the status of the task is updated in the task list. Everything else goes the same way; the display logic once again loops through every task. The "checked if done" and "strikethrough if done" rule are picked up to give the completed "Buy bread" task the appearance we want.

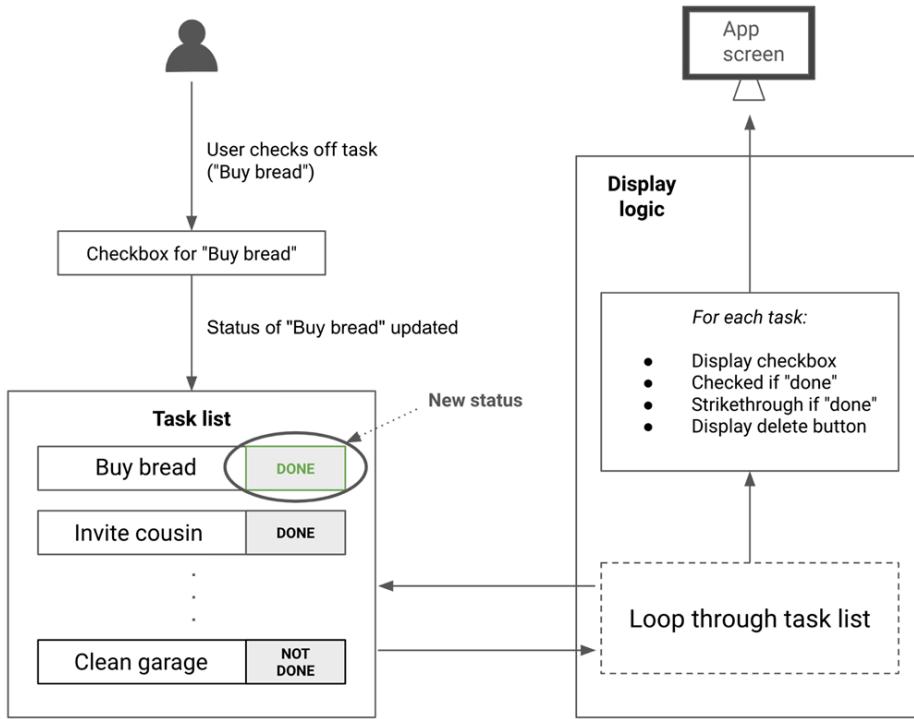


Figure 4.3 Checking off a task updates the status of the item in the task list and the same display logic re-renders the updated task list

Deleting a task or un-checking one works pretty much the same way; the task list stored in memory is updated and re-rendered by the display logic.

At this point, we've identified how we'll represent the key entities in our app, and what effect each user action will have. It's now time to implement our logic.

4.2 Implementing and troubleshooting our app

When we built the unit conversion app in Chapter 3, we took the scenic route, walking through each step of the app development process in detail. However, there's a part of the process we didn't really dwell on: what happens when things go wrong, and how to troubleshoot the problem. It was a fairly smooth ride.

This time around, we'll take a bumpier path that you'll find to be more representative of the real world. We'll run into various issues and errors as we implement our to-do list app. Just as in the real world, these errors will prompt us to learn about Streamit in more depth. And we'll use our deeper understanding to power through and fix the problems.

NOTE Since the focus of this chapter is to give you experience in troubleshooting problems with your apps, we'll eschew some of the best practices we learned in Chapter 3 (such as maintaining a strict separation between backend and frontend, or defining a clear API), in favor of more concise code.

To begin, spin up a new file in your code editor and name it `todo_list.py`.

4.2.1 Displaying the task list

While we've done our planning up front in one shot, when it comes to writing the actual code, we're going to be building our app iteratively as we did in Chapter 3, going part-by-part and viewing the results in Streamlit along the way.

So where do we start? What's the first iteration?

As discussed in the previous section, the notion of a task list is central to our app's implementation. The display logic component of our app always needs to show the latest state of the task list.

Our very first step can be as simple as creating the heading for the task list.

ST.HEADER

Streamlit has several different text elements which simply display text in various sizes and formatting. We've used `st.title` before to render large titles.

`st.header` is pretty similar, but it displays text that's a bit smaller than `st.title`.

Use it by putting the code in Listing 4.1 into `todo_list.py`.

Listing 4.1 todo_list.py with just a header

```
import streamlit as st

st.header("Today's to-dos:", divider="gray")
```

Notice that we've included a divider argument, which simply displays a gray line beneath your header. Neat, huh?

As always, to see your work in action, save your file and run `streamlit run todo_list.py` or `streamlit run <path to frontend.py>` if you're in a different working directory.

When your browser window opens, you should see something like figure 4.4.

Today's to-dos:

Figure 4.4 st.header with a divider

If you happen to have multiple headers in your app, you can even cycle between divider colors by setting `divider` to `True` instead of a specific color.

CREATING A TASK LIST

Next, let's turn to our notion of a task. As mentioned earlier, a task has a name and a done/not-done status.

We can therefore use a dataclass to represent a task with exactly those two fields: a `string name` and a `boolean is_done` to represent the task status. Listing 4.2 shows the `Task` class. Go ahead and save this into a new file called `task.py` in the same directory as `todo_list.py`.

Listing 4.2 task.py

```
from dataclasses import dataclass

@dataclass
class Task:
    name: str
    is_done: bool = False
```

Notice the line `is_done: bool = False`. Here, we're setting `is_done` to `False` by default, in case it's not specified while creating an instance of `Task`. This will come in handy momentarily.

Now that we have a task, our task list is literally a Python list of `Task` objects. You can create this in `todo_list.py` with a couple of dummy tasks to test it out like this:

```
task_list = [Task("Buy milk"), Task("Walk the dog")]
```

Since we've specified a default value of `False` for `is_done`, there's no need to specify it for each instance of `Task`.

Don't forget to import your `Task` class at the top of the file:

```
from task import Task
```

CHECKBOXES FOR TASKS

Starting out, our display logic can be simple: let's just display a checkbox for each task. We know how to create a static checkbox with a string label; recall that we used `st.checkbox` to make a checkbox that rounds the results of our unit conversion in Chapter 3.

But here, we don't know the labels for each checkbox beforehand. Instead we have to infer them from `task_list`. How do we do that?

The answer, of course, is a loop. When a Streamlit element is placed in a loop, a new element is rendered each time the loop runs. We've actually already encountered this in our initial password checker example in Chapter 2, where we used `st.success` and `st.error` in a loop to display the green and red boxes that indicated the pass/fail status of each condition.

We can create checkboxes from our task list like this:

```
for task in task_list:
    st.checkbox(task.name, task.is_done)
```

Recall that the first argument passed to `st.checkbox` is the label (the task's name in this case) and the second is a boolean that indicates whether the checkbox should be rendered as checked or not. We want each checkbox to be checked if the task is done, so it makes sense to pass the task's `is_done` field here directly.

Listing 4.3 shows what `todo_list.py` should look like at this point.

Listing 4.3 task.py with checkboxes for each task

```
import streamlit as st
from task import Task

task_list = [Task("Buy milk"), Task("Walk the dog")]

st.header("Today's to-dos:", divider="gray")
for task in task_list:
    st.checkbox(task.name, task.is_done)
```

Save and run to get the output shown in figure 4.5

Today's to-dos:

Buy milk

Walk the dog

Figure 4.5 Using st.checkbox in a loop to display one for each task

Our checkboxes don't actually do anything yet. We'll get to that in a minute, but first let's add a "delete" button to each task.

ADDING DELETE BUTTONS

We want a button to delete each task in the list, situated to its right. Similar to how we did it for the checkboxes, we're going to be generating these buttons dynamically, so they should go into the loop we wrote earlier.

But if we simply tack on the button inside the loop, Streamlit will put it *under* the checkbox for the task, not to the *right* of the task, because Streamlit renders elements vertically by default as we saw in Chapter 3.

As before, we'll use `st.columns` to work around this. Here, we'll make two columns—one for the checkbox and task text and another for the button. Go ahead and replace your existing `for task in task_list` loop with this:

```
for task in task_list:
    task_col, delete_col = st.columns([0.8, 0.2])
    task_col.checkbox(task.name, task.is_done)
    if delete_col.button("Delete"):
        pass
```

Notice that we're calling `st.columns` a little differently from how we did in the previous chapter: `st.columns([0.8, 0.2])`. Instead of passing the number of columns, we're passing a list of numbers. This list has the *relative widths* of each column. We're saying that the column with the task should take up 80% of the horizontal space and the column with the button should take 20%. If we had simply passed the number of columns, i.e., `st.columns(2)`, Streamlit would have made the two columns equally wide, which doesn't make sense because the task text can be arbitrarily long, while the button can't.

We're not making the button do anything just yet, so we just wrote `pass`, which is a keyword in Python that means "do nothing."

WIDGET KEYS

Let's go ahead and run our app again to see how it looks. Figure 4.6 depicts what you'll likely see.

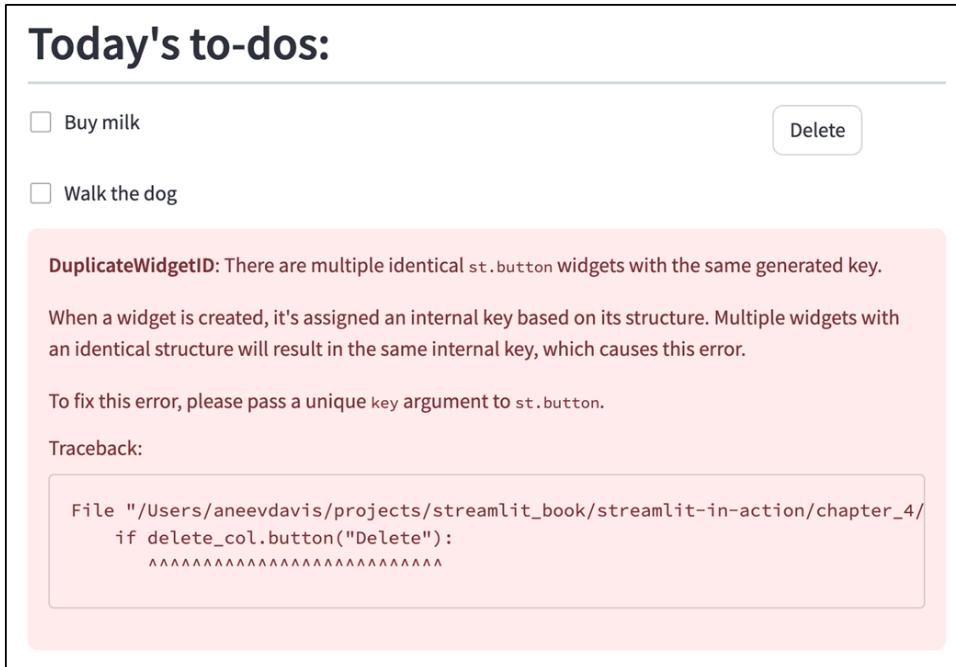


Figure 4.6 Streamlit throws an error when there are multiple identical widgets

There's a button to the right of the first task but not the second. Most importantly, there's an error message big red box underneath; Streamlit is complaining because we tried to create multiple `st.button` widgets with the same `key`.

A `key` is a piece of text that Streamlit uses to identify a *widget*—essentially what we've been calling a Streamlit element, like `st.button`, `st.checkbox` etc. Widget keys need to be unique so that Streamlit can distinguish between any two widgets.

You don't have to specify the `key` for a widget in your code manually most of the time because Streamlit specifies one internally based on its characteristics. For a button, Streamlit's internal key is based on the text in it. So when you have two buttons that say "Delete," their keys are identical, which breaks the uniqueness constraint.

The way around this, as the error suggests, is to manually specify a unique `key` for each button we create.

Since we need a unique delete-button `key` for every task in our list, one way of ensuring a unique `key` might be to include the list index for the task within the `key`. For instance, the `key` for the first task's delete-button could be `delete_0`, the `key` for the second could be `delete_1`, and so on:

```
for idx, task in enumerate(task_list):
    task_col, delete_col = st.columns([0.8, 0.2])
    task_col.checkbox(task.name, task.is_done)
    if delete_col.button("Delete", key=f"delete_{idx}"):
        pass
```

Since we need both the index of the task and the task itself, we've changed the for-loop's header to `for idx, task in enumerate(task_list)`.

NOTE `enumerate`, as you may know, is a handy little function in Python that lets you iterate through a list in an elegant way, obtaining both the index and the element in one shot. The less elegant alternative would have been to write:

```
for idx in range(len(task_list)):
    task = task_list[idx]
    ...
```

As we discussed, we form each button's unique key using its index: `key=f"delete_{idx}"`. If you run your code now, you should see the error disappear, as shown in figure 4.7

Today's to-dos:

Buy milk

Delete

Walk the dog

Delete

Figure 4.7 Passing a unique key to each button allows Streamlit to distinguish between otherwise identical buttons

You may now be wondering, "why didn't we need to pass a key to the checkboxes then?"

Well, because the checkboxes already had unique internal keys since their labels (the task names) were different. We'll actually face the same issue if we try to put two identical tasks in our list. For instance, if we change our task list to `task_list = [Task("Buy milk"), Task("Buy milk")]`, we'll see an error similar to what we saw for the buttons.

It's probably a good idea to let the user enter the same task twice if they want to, so let's fix the problem by passing a unique key to each checkbox as well:

```
task_col.checkbox(task.name, task.is_done, key=f"task_{idx}")
```

This lets us have two tasks with the same name if we like without issues.

4.2.2 Enabling actions

So far, we've set up our app to display our tasks in roughly the way we want them to appear, using dummy tasks to test it. We haven't actually provided a way for users to interact with or modify their tasks.

That's what we'll do in this section. We'll start by defining functions that update our task list, and then hook them up to Streamlit UI elements.

ADDING A TASK

To add a task to our task list, we need a task name. Once we have that, adding it is as simple as creating a `Task` object and appending it to our list.

We can write this out in a simple `add_task` function in `todo_list.py`:

```
def add_task(task_name: str):
    task_list.append(Task(task_name))
```

MARKING A TASK DONE OR NOT DONE

A task's status is denoted by the `is_done` field of the `Task` instance. Therefore, marking it done or not done involves updating this field. Let's create two functions for this:

```
def mark_done(task: Task):
    task.is_done = True

def mark_not_done(task: Task):
    task.is_done = False
```

Note that the argument to these functions is the `Task` instance itself, not the task name string.

DELETING A TASK

Deleting a task is also straightforward. For this function, we need the index of the task in our list, so we can remove it.

```
def delete_task(idx: int):
    del task_list[idx]
```

ENABLING USERS TO ADD TASKS

As we now have an `add_task` function, we no longer have to seed our task list with dummy tasks. Let's replace the line `task_list = [Task("Buy milk"), Task("Walk the dog")]` with an empty list:

```
task_list = []
```

Next, we'll add Streamlit elements to let the user call our `add_task` function. We'll need an `st.text_input` for the user to enter the task name, and an `st.button` to trigger the addition. We'll wrap both of these in `st.sidebar` so they appear in a left-hand-side panel in our app. Again, if any of this sounds unfamiliar, you should review Chapter 3.

```
with st.sidebar:
    task = st.text_input("Enter a task")
    if st.button("Add task", type="primary"):
        add_task(task)
```

Note the `type="primary"` in `st.button`. The `type` parameter lets you add emphasis to a button (in the form of a different color) by denoting that it's linked to a "primary action." In UI design, it's a good idea to have your user's eyes be drawn to the actions that they would commonly perform. Here, adding a task is something we'd expect the user to do all the time, so using a primary button makes sense. If you don't specify this parameter (which we've been doing all along until now), it defaults to "secondary," which—at the time of writing—results in a white button.

Note also that we didn't add a widget key to the button because we only have one "Add task" button and Streamlit doesn't need any extra help to distinguish it from other buttons.

At this point, your `todo_list.py` file should like what's shown in listing 4.4.

Listing 4.4 todo_list.py so far

```

import streamlit as st
from task import Task

task_list = []

def add_task(task_name: str):
    task_list.append(Task(task_name))

def delete_task(idx: int):
    del task_list[idx]

def mark_done(task: Task):
    task.is_done = True

def mark_not_done(task: Task):
    task.is_done = False

with st.sidebar:
    task = st.text_input("Enter a task")
    if st.button("Add task", type="primary"):
        add_task(task)

st.header("Today's to-dos:", divider="gray")
for idx, task in enumerate(task_list):
    task_col, delete_col = st.columns([0.8, 0.2])
    task_col.checkbox(task.name, task.is_done, key=f"task_{idx}")
    if delete_col.button("Delete", key=f"delete_{idx}"):
        pass

```

Save and run your code. To check the result (see figure 4.8), enter a new task called "Clean garage" and click "Add task."

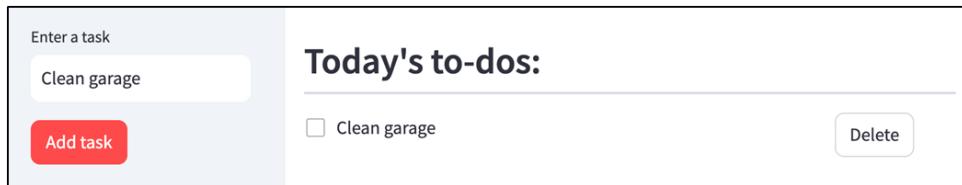


Figure 4.8 To-do list app with one to-do added

So far so good, but when we try adding another task, say "Finalize project proposal," we see the problematic output shown in figure 4.9.



Figure 4.9 When a new task is added, the old one disappears

We see our new task alright but the old one, "Clean garage," is gone. Something seems wrong, but we don't see an error as we did for the widget key issue.

Oddly, clicking "Delete" removes the remaining task (see figure 4.10), even though we didn't actually wire it up to anything; recall that we used `pass` to make the button do nothing—something that's commonly referred to as a *no-op*.

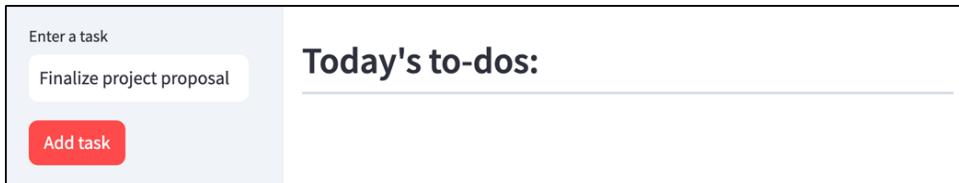


Figure 4.10 Clicking "Delete" removes the task even though we didn't hook it up to anything.

The same thing happens if you add a task again and click the checkbox: the task just disappears. Feel free to try that out as well.

WHAT'S GONE WRONG?

Clearly, our app isn't working as we intended. Streamlit doesn't show an error, so it's up to us to figure out what's happening. Is our display logic only showing the last task added? Or is there something wrong with the task list itself?

Let's find out. One of the most important parts of troubleshooting code is inspecting the value of variables while a program is running. In a normal Python script (i.e., one that you would run from the command-line rather than using Streamlit), you might include `print` statements to display the value of a variable. You could also use your IDE's debugger or the `pdb` module, but let's keep things simple.

`print` statements don't show up on the browser window of your Streamlit app, however. Instead, let's use an appropriate Streamlit element. We're interested in the `task_list` variable, so go ahead and write the following right under the line `st.header("Today's to-dos:", divider="gray")`, before our display logic for-loop.

```
st.info(f"task_list: {task_list}")
```

`st.info` is an element that displays some text in a colored box. It's part of a family of elements we've seen already in Chapter 2: `st.success`, `st.error`, and `st.warning`, which also display text in colored boxes. For `st.info`, the box is blue.

When you save and run (or refresh the page), you'll see a box with the text `task_list: []`, since there are no tasks yet. Add a task like before and you'll see the output in figure 4.11.

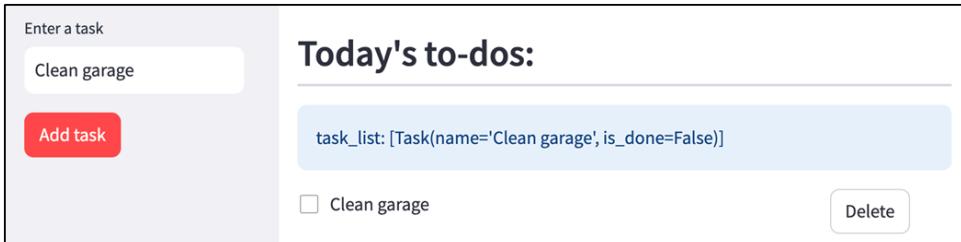


Figure 4.11 `task_list` contains a single Task instance

As you can see, `task_list` now contains a single instance of `Task`, corresponding to "Clean garage." When we add a second task, our `task_list` variable only has the new task. That shows that it's not our display logic that's faulty; the `task_list` variable itself has lost the "Clean garage" task. .

When we try either checking the box next to the task or pressing "Delete" `task_list` is now empty again, which is why no tasks are displayed.

Okay, so here's what we know: adding a task seems to add a task to `task_list` correctly, but whenever you do *anything* else afterwards, whether it's adding another task or clicking a checkbox or the "Delete" button, it removes the previously added task from `task_list`.

Before we can fix this, we'll need to understand why this is happening. For this, let's review how a Streamlit app actually works.

4.3 How Streamlit executes an app

In the last two chapters, we've learned how to use Streamlit and even developed a couple of non-trivial apps with it. However, we've mostly focused on syntax and a surface-level understanding of how apps work.

To be successful in writing more complex Streamlit apps, we'll need to dive deeper than that. To go further, we need to talk about something quite fundamental to Streamlit: its execution model.

4.3.1 Frontend and server

A Streamlit app actually has two parts: a backend *Streamlit server* and a *frontend*.

A server, for our purposes, is a software program that runs on your computer, waiting for requests to be sent to it. In technical terms, we say that a server is *listening* at a *port*.

A port is a virtual designation that identifies a particular kind of communication channel, kind of like an extension number in a large office. Just as an extension allows you to reach a specific person within a company, a port allows network communication to reach a specific program running on your computer.

When you enter `streamlit run <filename.py>` in your terminal, you may have noticed output that looks something like the following:

```
Local URL: http://localhost:8502
```

What's actually happening here is that a Streamlit server starts up and starts listening for requests on the port 8502 (the exact port number may differ for you).

When you now open a browser and navigate to the given address (i.e., `http://localhost:8502`) or just wait until the server automatically does this for you, the browser sends a request to the Streamlit server on port 8502.

In response, the Streamlit server executes your Python script from top to bottom and sends a message back to the browser, telling it what to display, i.e., the frontend.

The frontend is thus the front-facing part of your app that users can see and interact with, and it runs on your web browser. It consists of HTML, CSS, and Javascript code that your browser understands.

4.3.2 App re-runs

Now, here's the important part: the Streamlit server runs your Python script in its entirety every time the page needs to change. This includes each time the user interacts with a widget in your app.

For example, figure 4.12 details what happens when a user clicks a button in your app.

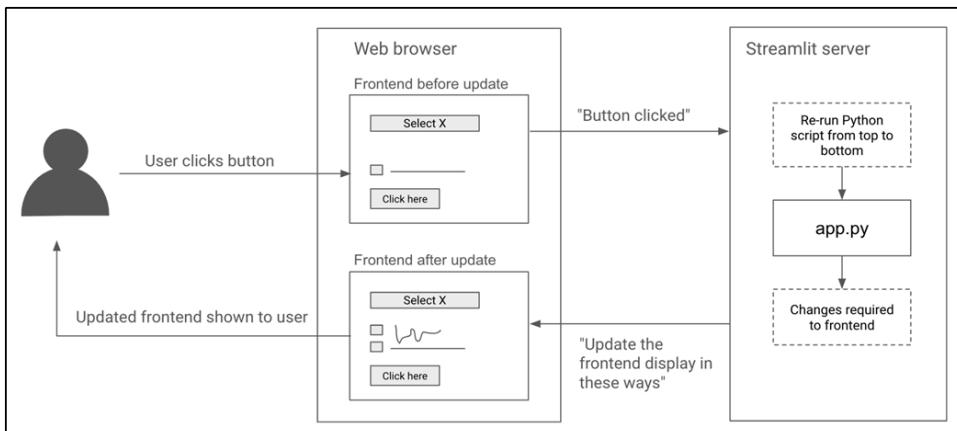


Figure 4.12 Every time the user interacts with the app, the Python script is re-run

Once the frontend detects the button-click, it sends a message to the server, informing it about the click. The server reacts to this information by re-running the Python code, setting the button to evaluate to `True`.

When that's done, the server sends the frontend a message with the changes that need to be made to the display. The frontend then makes those changes and the user sees the updated display.

Note that this isn't unique to button clicks; it applies for *any* interaction or *any* time Streamlit determines the display needs to change. This means that every single time the user clicks a button or selects a different item from a dropdown or moves a slider, the cycle repeats and the server re-runs your whole Python script.

4.3.3 Applying this to our app

Let's see if we can figure out what's happening in our to-do list app with this knowledge of app re-runs.

Listing 4.5 shows the code as it exists at the moment.

Listing 4.5 todo_list.py

```

import streamlit as st
from task import Task

task_list = []

def add_task(task_name: str):
    task_list.append(Task(task_name))

def delete_task(idx: int):
    del task_list[idx]

def mark_done(task: Task):
    task.is_done = True

def mark_not_done(task: Task):
    task.is_done = False

with st.sidebar:
    task = st.text_input("Enter a task")
    if st.button("Add task", type="primary"):
        add_task(task)

st.header("Today's to-dos:", divider="gray")
st.info(f"task_list: {task_list}")
for idx, task in enumerate(task_list):
    task_col, delete_col = st.columns([0.8, 0.2])
    task_col.checkbox(task.name, task.is_done, key=f"task_{idx}")
    if delete_col.button("Delete", key=f"delete_{idx}"):
        pass

```

We're now going to walk through how this code executes at various points in the app's usage.

FIRST RUN

The very first time our app runs, i.e., when the user first loads it, `task_list` is set to an empty list.

Now consider this line within the `st.sidebar` context manager:

```
if st.button("Add task", type="primary")
```

This is an `if` statement, so the line under it, i.e., `add_task(task)`, will only be executed if the `st.button` expression evaluates to `True`.

So far the button hasn't been clicked, so it evaluates to `False` and `add_task` is not called. `task_list` is therefore still an empty list.

The code then proceeds to the `st.info` box and display logic, but since there are no tasks, `st.info` shows an empty list, and the loop never executes, so there are no checkboxes.

USER ADDS A TASK

Let's now say the user has entered a task, "Clean garage," and clicked the "Add task" button. As mentioned earlier, this triggers a re-run of the entire Python code.

NOTE Technically, a re-run may already have occurred at this point even *before* the user clicks the button. When the user has finished entering "Clean garage," if they shift focus outside of the textbox by clicking outside it, that would qualify as an interaction (since the value in the textbox has changed) and trigger a re-run of the code. This doesn't lead to any interesting changes though, so let's ignore it for now.

Starting from the top of the script again, `task_list` is set to the empty list. Due to the line `task = st.text_input("Enter a task")`, the variable `task` now holds the string "Clean garage" as that's what's in the textbox.

Since the button has just been clicked, `st.button` evaluates to `True`, so the `if` statement is triggered and `add_task` is called.

`add_task` creates a `Task` instance for "Clean garage" and appends it to `task_list` so it's no longer empty. This is what `st.info` shows.

The display logic loop thus runs once, and proceeds to render a checkbox and delete-button. This concludes the re-run, producing the results shown in figure 4.13.

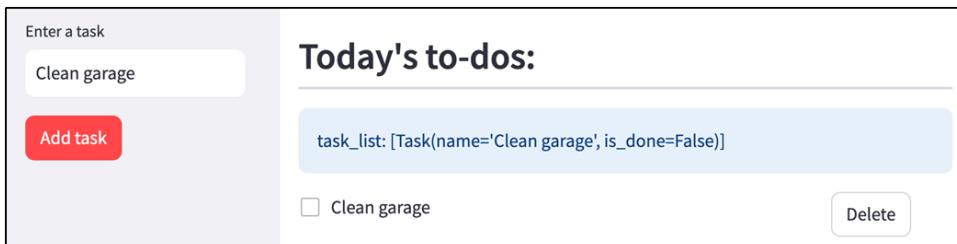


Figure 4.13 When the user clicks "Add task," `st.button` evaluates to `True` and `task_list` has a task

USER CLICKS THE TASK CHECKBOX

So far so good. Everything seems like it's working. But when the user clicks the checkbox for "Clean garage," it triggers another re-run.

Once again, we start from the top, where we have the line `task_list = []`, which again sets it to the empty list, discarding the "Clean garage" task that was in there before!

But let's assume that the `textbox` hasn't been cleared yet, and still says "Clean garage." This means that once the line `task = st.text_input("Enter a task")` has been executed, the variable `task` still contains the string "Clean garage."

What happens when we get to the `st.button` line? The button has been clicked before, so does that mean it would evaluate to `True`? If it does, then `add_task` would be triggered again, appending "Clean garage" to `task_list`, restoring its earlier state, and everything would be fine.

But that's not how `st.button` works. In reality, `st.button` evaluates to `True` only in the re-run that happens *immediately* after a click. In all later re-runs, it reverts to its original `False` value. In this case, clicking the checkbox triggered an all new re-run, so `st.button` now evaluates to `False`.

This means that `add_task` is never called and `task_list` is never updated. It remains an empty list, so in turn, the for-loop is never executed and there are no displayed tasks.

USER ADDS ANOTHER TASK INSTEAD OF CLICKING THE TASK CHECKBOX

Just to close out this discussion, let's consider the scenario where, instead of clicking the task checkbox, the user tries to add another task (by entering "Finalize project proposal" in the task-entry textbox and clicking "Add task").

The execution proceeds similarly in this case. `st_task` is set to the empty list at the top, so we lose the previous "Clean garage" task.

Since we entered a new task, the textbox now holds "Finalize project proposal," so that's what the `task` variable holds.

This time, our `st.button` does evaluate to `True` by virtue of our latest button-click, and `add_task` is called with "Finalize project proposal" as the value of the passed argument. This adds the new `Task` to our otherwise empty list.

At the end of this, `task_list` contains just one element: "Finalize project proposal," which is what's displayed by `st.info` and our display logic loop, as seen in figure 4.14.

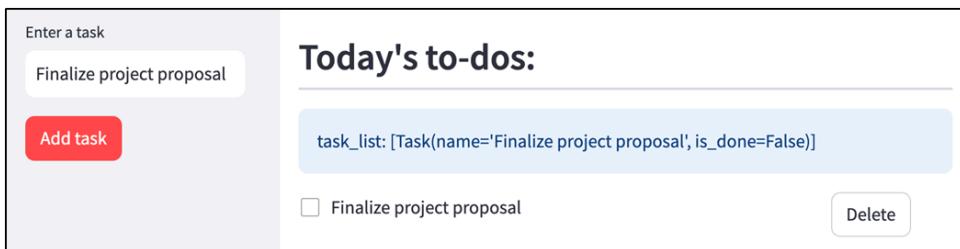


Figure 4.14 When the user adds a different task, `st.button` evaluates to True again, and "Finalize project proposal" is added to `task_list` which was empty at the start.

We can finally explain the weird results we were seeing. The problem boils down to the fact that since our script re-runs each time, `task_list` keeps getting reset.

4.4 Persisting variables across re-runs

In the last section, we were able to explain the unexpected output we were getting by reviewing Streamlit's execution model and stepping through our app's execution at various stages with our newfound understanding. In this section, we'll try and identify an approach to actually solve the issue.

To recap, our dilemma is that the app we wrote is behaving like a goldfish: it has no memory of anything that happened in any previous run. And since Streamlit re-runs the entirety of our code every chance it gets, our app's memory gets wiped out repeatedly, resetting the `task_list` variable that we're using to hold the user's tasks.

4.4.1 `st.session_state`

As it turns out, Streamlit has a solution for this, in the form of `st.session_state`. In a nutshell, `st.session_state` is a container for variables whose values will persist across re-runs.

Session here refers to an app session, which you can loosely think about as the time between when you open an app and either refresh the page or close it.

When you need to remember a value, you can just save it into `st.session_state`, and retrieve that value in the next run, as shown in figure 4.15.

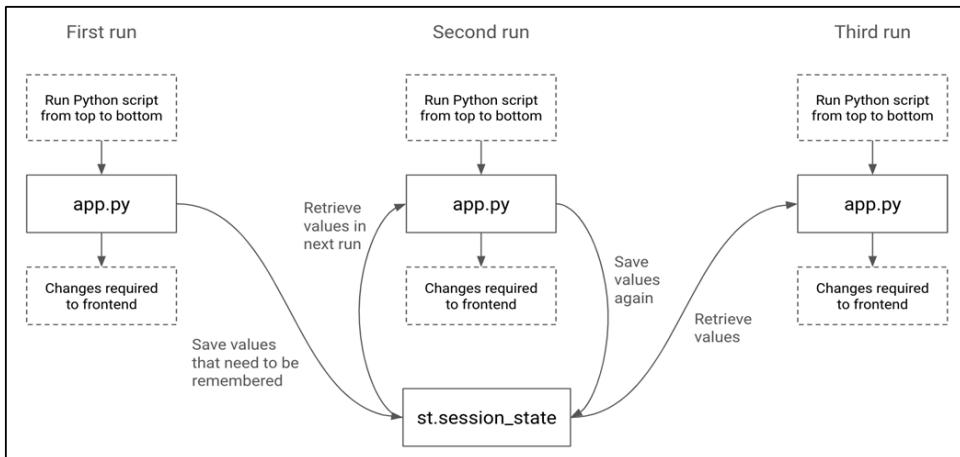


Figure 4.15 `st.session_state` can be used to save and retrieve values between re-runs

`st.session_state` is thus a rock of stability in an ocean of change. Or if you want a more technical metaphor, a store for the variables you want to persist across re-runs.

So how do we actually make use of it? Well, `st.session_state` acts almost exactly like a Python dictionary though it technically isn't one. Just as in the case of a dictionary, you can add key-value pairs to it, check if a particular key exists, look up its value, or remove it altogether. Even the syntax used is identical to that of a dictionary for the most part.

For instance, if you wanted to store a variable `x` with a value of 5 in `st.session_state`, you would write `st.session_state["x"] = 5`, and then retrieve the value using `st.session_state["x"]`.

To check if `x` exists in the session state, you would write `if "x" in st.session_state`. You can even iterate through the items in `st.session_state` using `for key, value in st.session_state.items()`, and delete a key using `del st.session_state[key]`.

Unlike a dictionary though, you can also use dot-notation to refer to the value of a key `x` like this: `st.session_state.x`.

Listing 4.6 shows a toy Streamlit app using `st.session_state` whose job is to simply keep track of and increment a number:

Listing 4.6 A simple number-increment app

```
import streamlit as st

if "number" not in st.session_state:
    st.session_state.number = 0

if st.button("Increment"):
    st.session_state.number += 1

st.info(f"Number: {st.session_state.number}")
```

We start by checking if the key "number" exists in the session state and adding it with a value of zero if it doesn't.

Then we have a button that increments the value of "number" each time it's clicked, and an `st.info` box to retrieve and display the value of number.

Figure 4.16 shows the output after pressing the "Increment" button five times.



Figure 4.16 A toy Streamlit app using `st.session_state` to keep track of and increment a number

If we hadn't used `st.session_state` here and had simply stored `number` in its own variable (or in a regular dictionary), it wouldn't have worked because the value (or the dictionary itself) would have gotten reset each time the app re-ran. `st.session_state` is the only thing that retains its state across re-runs of an app.

Why do we need the initial check to see if "number" already exists in the session state before adding it? Well, without this, we'd run into the same problem as before. Each time the app runs, it would set `st.session_state.number` to zero, overriding whatever value it had been incremented to in the earlier run, and we'd never see the number actually change.

By checking to see if "number" exists, we're ensuring that the line `st.session_state.number = 0` is only executed once—in the very first run when "number" hasn't been added yet.

4.5 Completing our app

We now know how to give our app a "memory." When you start writing Streamlit apps for your own purposes, you'll quickly realize that this knowledge is absolutely crucial—to the point that you couldn't write anything but the simplest apps without it.

Armed with the powerful `st.session_state`, we're ready to take another crack at getting our to-do list app to work!

4.5.1 Adding session state

When we last ran our app, our main problem was the fact that the `task_list` variable, which holds all our tasks, was getting reset with every re-run.

Let's fix this by adding `task_list` to `st.session_state`. Replace the line `task_list = []` in your earlier code with this:

```
if "task_list" not in st.session_state:
    st.session_state.task_list = []
```

This mirrors the toy example we walked through in the last section. The only difference is that we're storing `task_list` in `st.session_state` rather than a single number.

We could now modify the rest of our code to reference `st.session_state.task_list` everywhere it's currently referencing `task_list`, but that seems tedious and rather clunky. Instead, let's just point the variable `task_list` to the version in `st.session_state` like this:

```
task_list = st.session_state.task_list
```

Now the rest of our code should work fine since they're referring to `task_list`.

Listing 4.7 shows what our code should contain now.

Listing 4.7 todo_list.py with `st.session_state`

```
import streamlit as st
from task import Task

if "task_list" not in st.session_state:
    st.session_state.task_list = []
task_list = st.session_state.task_list

def add_task(task_name: str):
    task_list.append(Task(task_name))

def delete_task(idx: int):
```

```

def del_task_list[idx]:
    task_list.pop(idx)

def mark_done(task: Task):
    task.is_done = True

def mark_not_done(task: Task):
    task.is_done = False

with st.sidebar:
    task = st.text_input("Enter a task")
    if st.button("Add task", type="primary"):
        add_task(task)

st.header("Today's to-dos:", divider="gray")
st.info(f"task_list: {task_list}")
for idx, task in enumerate(task_list):
    task_col, delete_col = st.columns([0.8, 0.2])
    task_col.checkbox(task.name, task.is_done, key=f"task_{idx}")
    if delete_col.button("Delete", key=f"delete_{idx}"):
        pass

```

Save, re-run, and try adding multiple tasks. Figure 4.17 shows what you get when you do this.

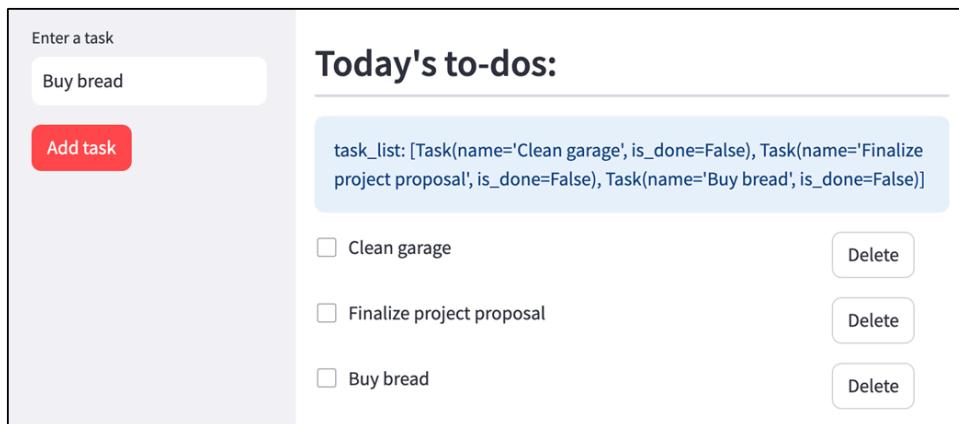


Figure 4.17 With `st.session_state`, Streamlit remembers our old tasks

And voila! `task_list` can finally be updated with multiple tasks, and our display logic shows everything.

4.5.2 Wiring up the "Delete" buttons

With that in place, let's get our "Delete" buttons working. Recall that we had previously set them up to do nothing by writing `pass` under the code for the buttons within the display loop.

```
if delete_col.button("Delete", key=f"delete_{idx}"):
    pass
```

Since then, we've created a `delete_task` function, so let's call that here instead:

```
if delete_col.button("Delete", key=f"delete_{idx}"):
    delete_task(idx)
```

If we now click "Delete" next to "Buy Bread" after saving and re-running (and adding the three tasks back in order if you refreshed the page), we see... no changes! If we click the button a second time, however, the task disappears. Something still isn't right.

I won't detail all of it with screenshots, but if you play around with the app at this point, you'll notice more odd behavior. The first time you click the "Delete" button against the last task in the list, it doesn't do anything. But if you then immediately click a checkbox (*any* checkbox), the task disappears.

Or if you delete a task from the middle of the list, the *next* one disappears, not the one you deleted! But if you *then* do something else, like clicking a checkbox or adding another task, that task comes back and the one you actually deleted is correctly removed and everything is the way it should be.

All in all, there seems to be a *lag* between when you actually click the "Delete" button and when the task is removed. You seem to need to do something else (anything else, like clicking one of the checkboxes, or editing the text in the task entry textbox and clicking outside) *after* clicking the button for the correct results to be displayed.

4.5.3 What's happening behind the scenes

To understand what's going on, we need to do a deep-dive into our app's execution once again. Let's assume we're at the stage in the app where the user has entered three tasks in order: "Clean garage." "Finalize project proposal." and "Buy bread."

At this point, `task_list` has been populated with these three tasks.

STEPPING THROUGH THE APP'S EXECUTION

Let's say the user tries to delete the third task. Figure 4.18 shows diagrammatically what happens in the app. The delete buttons are identified by the Streamlit widget keys assigned to them, i.e., `delete_0`, `delete_1`, etc.

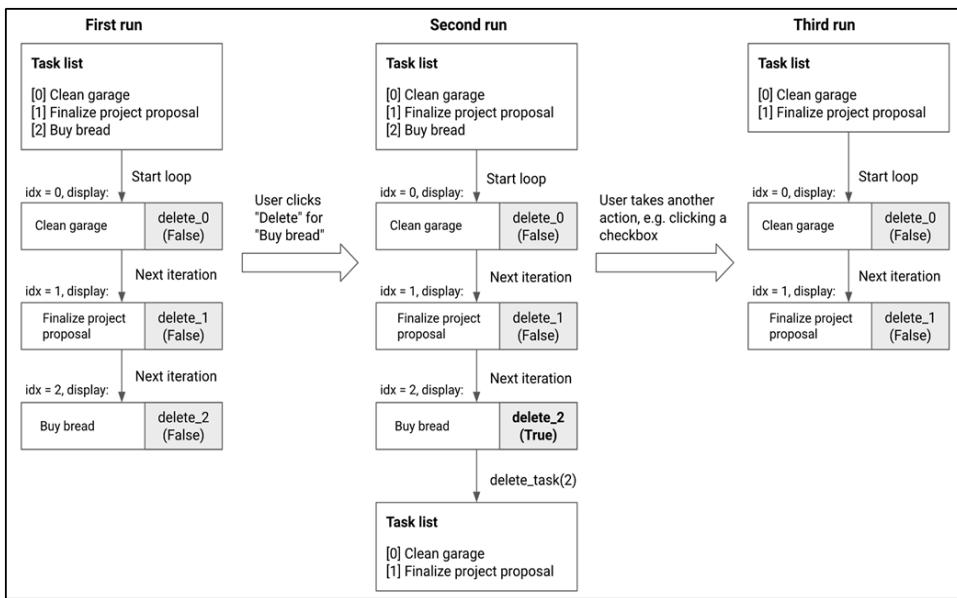


Figure 4.18 Step-by-step app execution: the delete button evaluates to True in the second run, but the task and button are displayed before `delete_task` is called

The first run is the one that happens before the button is clicked. Streamlit simply loops through our task list, displaying each task along with its checkbox and delete button. As we discussed earlier in the chapter, each `st.button` evaluates to `False`, since none of them have been pressed yet.

When the user clicks the delete button for "Buy bread," it triggers a re-run of the app. Everything remains the same until after we have displayed the third task's `st.button`. This time, this button evaluates to `True` since it was just clicked. As the condition is true, the app enters the code nested under `st.button, delete_task(idx)`. Since `idx` is 2 in this iteration of the loop, `delete_task(2)` is called, and "Buy bread" is removed from `task_list`. Execution stops at this point.

See the problem? All three buttons had *already* been displayed *before* `delete_task` was executed, updating `task_list`. And since there are no other user actions, no more re-runs are triggered. So `task_list` is updated all right, but the display logic has already been executed on the old version of `task_list`. That's why we still see the three tasks after clicking "Delete."

But at this point, if the user does something else, like clicking a checkbox or even clicking the "Delete" button again, it triggers another re-run. This time, the display logic runs over the latest version of `task_list`, so we finally see the third task and its checkbox and delete button removed.

NOTE Stepping through the execution like this can also explain the other odd behavior we noticed, such as when you click the delete button for a task in the middle of the list, and the *next* task disappears. This happens because when `delete_task` is called, the list indices all get moved up by one, and the next display loop iteration ends up skipping a task because its index changed.

4.5.4 Triggering re-runs automatically

As we've seen, though our delete button doesn't work correctly immediately, Streamlit does get the results right eventually, provided the user takes an extra action, triggering a re-run.

We can use this knowledge to our advantage. All we need is a way to trigger a re-run of the app through code, rather than through a user action. Streamlit offers this functionality through `st.rerun`, and you can call it at any time without any arguments, like so:

```
st.rerun()
```

When you call `st.rerun`, you're essentially telling Streamlit, "Quit the current run and start again from the top."

In our case, we should trigger the re-run once a task has been deleted:

```
if delete_col.button("Delete", key=f"delete_{idx}"):
    delete_task(idx)
    st.rerun()
```

If you make this change, re-run and recreate the tasks as before, and try deleting "Buy bread" again, you'll see the output in figure 4.19.

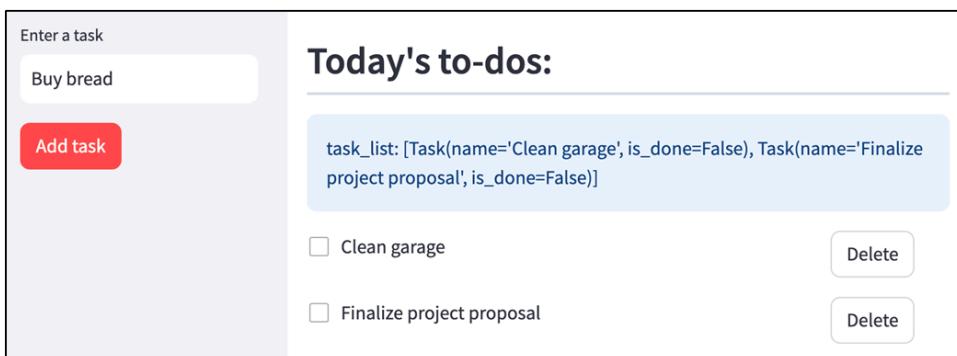


Figure 4.19 Pressing "Delete" works as expected with `st.rerun`

It worked! "Buy bread" is no more, and it's also missing from `task_list` as you can see from the `st.info` box in the screenshot.

4.5.5 Wiring up the checkboxes

Let's move on to the next part of our app: the checkboxes. We've added them in the display and you can check them, but they don't actually do anything. Our next step, then, is to hook them up to the functionality we've defined for changing the status of a task.

Our current checkbox code is a single line:

```
task_col.checkbox(task.name, task.is_done, key=f"task_{idx}")
```

When the user checks a task checkbox, we want to achieve two things:

- Mark the task as done, and
- Strike it through

We also want to reverse the above changes if the user unchecks a box.

To change the task's status, we can use the functions we created earlier for the purpose, `mark_done` and `mark_not_done`.

How do we achieve the strikethrough? For this formatting effect (and several others), Streamlit supports a language called *markdown*.

Markdown is a special text-based notation to add various kinds of formatting. It has ways to display text in bold or italic, to create links, lists, headings, and much more. We'll encounter these in later chapters, but for now let's focus on the strikethrough effect.

To strike a piece of text through in markdown, you surround it with two pairs of tildes, like this:

```
~~Text to be struck through~~
```

This plugs into our checkbox through the `label` parameter, which supports markdown. We'll define a variable that contains the name of a task by itself if the task has not been done, and the name with a markdown strikethrough if it does:

```
label = f"~~{task.name}~~" if task.is_done else task.name
```

We can then feed it into our checkbox:

```
task_col.checkbox(label, task.is_done, key=f"task_{idx}")
```

Finally, let's also wire up our checkboxes to our `mark_*` functions. We want to call `mark_done` if the checkbox is checked, or `mark_not_done` otherwise. Our overall code should now be as shown in listing 4.8.

Listing 4.8 todo_list.py after wiring up the task checkboxes

```

import streamlit as st
from task import Task

if "task_list" not in st.session_state:
    st.session_state.task_list = []
task_list = st.session_state.task_list

def add_task(task_name: str):
    task_list.append(Task(task_name))

def delete_task(idx: int):
    del task_list[idx]

def mark_done(task: Task):
    task.is_done = True

def mark_not_done(task: Task):
    task.is_done = False

with st.sidebar:
    task = st.text_input("Enter a task")
    if st.button("Add task", type="primary"):
        add_task(task)

st.header("Today's to-dos:", divider="gray")
st.info(f"task_list: {task_list}")
for idx, task in enumerate(task_list):
    task_col, delete_col = st.columns([0.8, 0.2])
    label = f"~~{task.name}~~" if task.is_done else task.name      #A
    if task_col.checkbox(label, task.is_done, key=f"task_{idx}"):
        mark_done(task)      #B
    else:
        mark_not_done(task)  #C
    if delete_col.button("Delete", key=f"delete_{idx}"):
        delete_task(idx)
        st.rerun()

#A Add a strikethrough effect to the label if the task is done
#B Call mark_done if the checkbox happens to be checked and therefore evaluates to True
#C Call mark_not_done if the checkbox is not checked

```

Save, re-run, and add your tasks back, then check one of the tasks to get results similar to what's shown in figure 4.20.

The screenshot shows a user interface for managing tasks. On the left, there is a sidebar with a text input field labeled "Enter a task" containing "Buy bread", a red "Add task" button, and a "Delete" button. The main area is titled "Today's to-dos:" and contains a list of tasks. The list includes "Clean garage" (checked) and "Finalize project proposal" (unchecked). Each task has a "Delete" button next to it. Below the list, an info box displays the code: `task_list: [Task(name='Clean garage', is_done=False), Task(name='Finalize project proposal', is_done=False)]`.

Figure 4.20 Checking a task doesn't immediately work as expected

Once again, we didn't get the results we expected. "Clean garage" remains unstruck, and our info box shows that `task_list` hasn't changed. Before you throw your computer out the window and dedicate the rest of your existence to sheep-farming, try checking off another task.

You'll see that now the original task we checked has its strikethrough, and its `is_done` field is `True` as per our `st.info` box.

Sound familiar? Looks like there's a lag of one user action between when we click a checkbox and when the result of that action shows up.

What's going on here is very similar to what we saw in the case of the delete button. Clicking the checkbox does trigger our function and sets `is_done` to `True`, but by that point, the task and its label have already been displayed. Only in the *next* re-run is the *actual display* updated, and that re-run is only triggered when the user takes a further action.

The solution to this is the same as before: we can trigger a manual re-run each time one of our `mark_*` functions runs:

```
if task_col.checkbox(label, task.is_done, key=f"task_{idx}"):
    mark_done(task)
    st.rerun()
else:
    mark_not_done(task)
    st.rerun()
```

Save the output, refresh the page, and let's try again. Figure 4.21 shows the output.

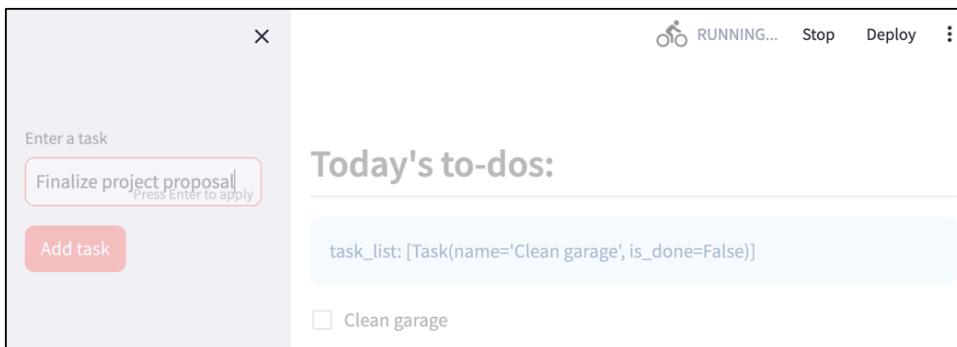


Figure 4.21 Our app hangs and never stops loading

Something's very wrong here. Our app seems to stop responding entirely once we add our first task. The screen's grayed out, and there's the "RUNNING..." indicator at the top.

4.5.6 An infinite re-run loop

You've just encountered your first Streamlit infinite re-run loop. Let's try to understand what went wrong by stepping through the execution one more time. Figure 4.22 shows this in a diagram.

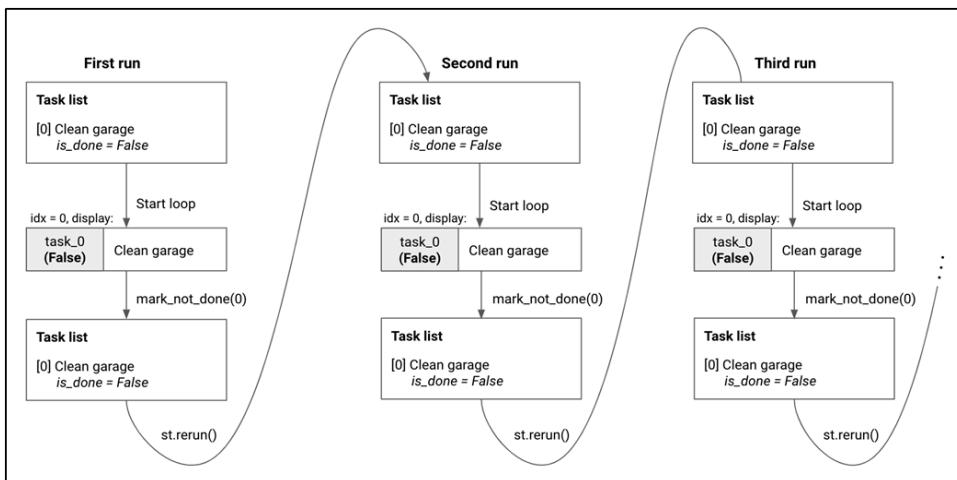


Figure 4.22 Step-by-step app execution: A chain of `st.rerun()` leads to an infinite loop

Once we've added our "Clean garage" task, `task_list` contains a single `Task` instance with its `is_done` field set to `False`.

Since `task_list` is non-empty, we enter the display loop, and the checkbox for "Clean garage" is displayed.

Now our task has branching logic:

```

if task_col.checkbox(label, task.is_done, key=f"task_{idx}"):
    mark_done(task)
    st.rerun()
else:
    mark_not_done(task)
    st.rerun()

```

The checkbox evaluates to `False` since it's not checked, which means the app enters the `else` clause.

`mark_not_done` is called, which sets `is_done` to `False` (even though it's already `False`), and then `st.rerun()` forces Streamlit to stop the current run and start again from the top.

Once again, in the second run, we enter the loop. The checkbox is still not checked, so `mark_not_done` is called again, and `st.rerun()` after that, which begins a third run, and so on and on.

Since this never stops, Streamlit chokes and stops responding.

4.5.7 Preventing the infinite re-run

The trouble here is that `mark_done` is getting called even when there's no need for it. Reviewing the execution steps we saw just now, you'll notice that the "Clean garage" task's `is_done` field was already set to `False`, so there was no actual need to call `mark_not_done` again.

The way our code is set up right now, once we enter our display for-loop, there's no exiting it. If our checkbox evaluates to `True`, `st.rerun()` is called after the `mark_done` function. If it evaluates to `False`, `st.rerun()` is called after the `mark_not_done` function.

We need to make sure that this only happens when it absolutely needs to. `mark_done` (and the associated `st.rerun`) should only be called if the checkbox is checked *and* the task is not already marked as "done." Similarly, `mark_not_done` and *its* `st.rerun` should only be called if the checkbox is not checked and the task is currently marked as "done."

We can make this happen by editing our code like this:

```

checked = task_col.checkbox(label, task.is_done, key=f"task_{idx}")      #A
if checked and not task.is_done:                                         #B
    mark_done(task)
    st.rerun()
elif not checked and task.is_done:                                         #C
    mark_not_done(task)
    st.rerun()

```

#A Save the checkbox's value in a new variable called `checked`, for readability.

#B Only call `mark_done` if the checkbox is checked AND the task is not yet marked as done.

#C Only call `mark_not_done` if the checkbox is not checked AND the task is still marked as done.

This way, when the checkbox is checked, the task's status is set to `is_done`, but in the next re-run, both the if and elif clauses evaluate to `False`, and `st.rerun` never executes.

Go ahead and try it out. Our checkboxes should now be working correctly, as shown in figure 4.23.

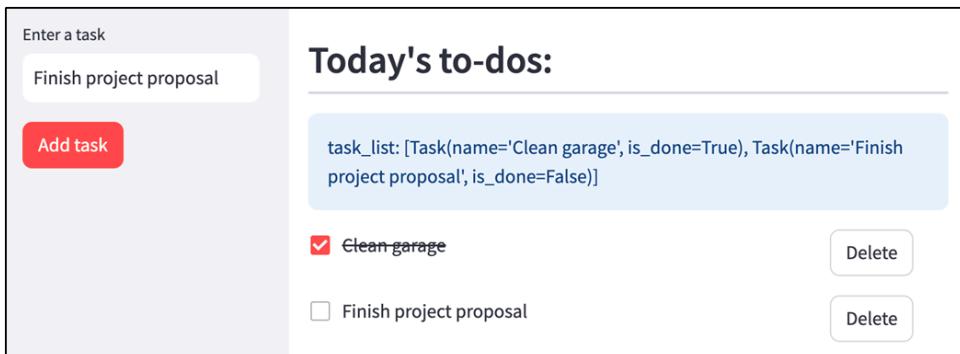


Figure 4.23 Our checkboxes now work as expected

4.5.8 Adding the completion progress indicator

We're *almost* done with our app. The only thing that remains from our earlier mock design is to add a progress indicator to give the user that extra sense of accomplishment.

This is quite straightforward. We want the indicator to be nice and large, so `st.metric`, which we encountered first in Chapter 3, seems ideal.

We'll need to show two things: the total number of tasks, and the number of completed tasks, both of which we can get from `task_list`. Our code for `st.metric` could look like this:

```
total_tasks = len(task_list)
completed_tasks = sum(1 for task in task_list if task.is_done)
metric_display = f"{completed_tasks}/{total_tasks} done"
st.metric("Task completion", metric_display, delta=None)
```

To get `completed_tasks`, we use a list comprehension (you can skip the square brackets when it's wrapped in a function like `sum`) for conciseness.

Oh, and we can probably get rid of our info box (`st.info`) since we're no longer in troubleshooting mode.

Our final code should look like what's shown in Listing 4.9.

Listing 4.9 The final version of todo_list.py

```
import streamlit as st
from task import Task
```

```

if "task_list" not in st.session_state:
    st.session_state.task_list = []
task_list = st.session_state.task_list

def add_task(task_name: str):
    task_list.append(Task(task_name))

def delete_task(idx: int):
    del task_list[idx]

def mark_done(task: Task):
    task.is_done = True

def mark_not_done(task: Task):
    task.is_done = False

with st.sidebar:
    task = st.text_input("Enter a task")
    if st.button("Add task", type="primary"):
        add_task(task)

total_tasks = len(task_list)
completed_tasks = sum(1 for task in task_list if task.is_done)
metric_display = f"{completed_tasks}/{total_tasks} done"
st.metric("Task completion", metric_display, delta=None)

st.header("Today's to-dos:", divider="gray")
for idx, task in enumerate(task_list):
    task_col, delete_col = st.columns([0.8, 0.2])
    label = f"~~{task.name}~~" if task.is_done else task.name
    checked = task_col.checkbox(label, task.is_done, key=f"task_{idx}")
    if checked and not task.is_done:
        mark_done(task)
        st.rerun()
    elif not checked and task.is_done:
        mark_not_done(task)
        st.rerun()
    if delete_col.button("Delete", key=f"delete_{idx}"):
        delete_task(idx)
        st.rerun()

```

Figure 4.24 provides a final glance at our app in all its glory.

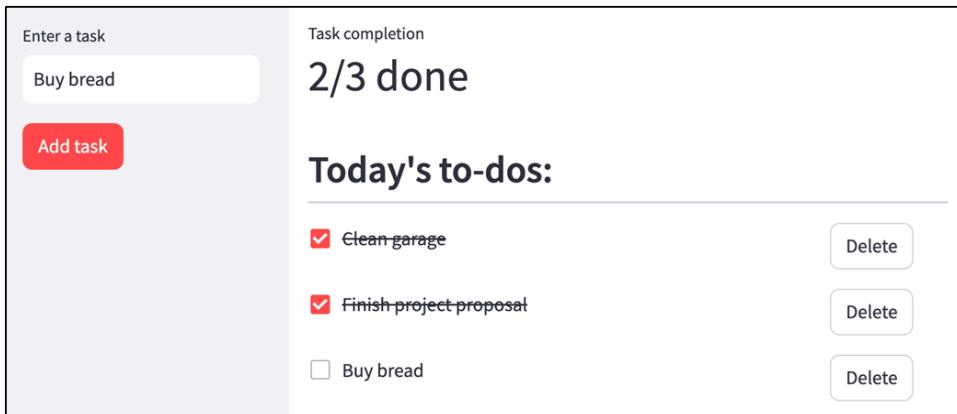


Figure 4.24 The final to-do list app

With that, you have another full app under your belt, and maybe even a tool you can use every day to stay productive! You're now already in a position to start using Streamlit in your own projects. In the next chapter, we'll see how to publish them for other people to use.

4.6 Summary

- In the real world, the development process is not smooth; much of your time will be spent troubleshooting things that don't work as expected.
- `st.header` is used to display headings in large font.
- Streamlit identifies UI widgets using a unique widget key based on its characteristics.
- When two widgets are identical in every respect, you have to specify a widget key manually to enable Streamlit to tell them apart.
- A good way to keep track of the values of variables as the app executes is to display them on the app's screen using `st.info`.
- Whenever the page needs to change, the Streamlit server re-runs your Python code from top to bottom.
- Re-runs reset all the regular variables in your app.
- `st.session_state` is used to store variables that you want Streamlit to remember between re-runs.
- It's a good idea to step through the app's execution when you see unexpected results.
- You can trigger a re-run of your app using `st.rerun`.
- While using `st.rerun`, your app may end up in an infinite re-run loop if you don't provide a path for your script to exit.

5 Sharing your apps with the world

This chapter covers

- The various options available to share your app with users
- Deploying an app to Streamlit Community Cloud for free
- Connecting an app to an external service like an API
- Safeguarding your API keys and other secrets in production
- Managing your app's dependencies

The moment you first successfully run an app you've built from scratch is magical—it's when all the hours spent designing, developing, and refining finally pay off. You've guided it through multiple iterations, squashed bugs, and fine-tuned every feature.

But what's next? Do you keep it hidden away on your local machine? Unless you've built something solely for your own use, the answer is probably no. To make your app truly useful, you need to get your app in the hands of your intended audience.

This chapter is about making the leap from local development to global deployment. We'll briefly discuss the various paths you have available for sharing your apps. We'll then settle on one of them and walk through putting your app in production for the world to experience.

Along the way, we'll cover key considerations involved in making your app public, such as safeguarding confidential information like API keys and managing your code's dependencies. As always, we'll take a practical approach through all of this, giving you direct hands-on experience with everything we discuss.

5.1 Deploying your apps

We've come a long way since we started out with Streamlit. Over the last three chapters, you've created three fully functional—dare I even say, *useful*—applications. However, you've been hiding your light under the proverbial bushel; no one else has experienced your craft. It's time to change that!

5.1.1 What is deployment?

Deploying an app loosely means making it available for other people to use. More specifically, it means hosting your application somewhere that your intended users can readily access.

Recall from Chapter 4 that a Streamlit app consists of a backend server and a frontend that runs on a web browser. While the frontend makes requests to the server and shows you the results, it's the server that really runs the show.

To set up the connection between the frontend and the server and thus load an app, the user has to navigate to the URL and port where the server is running. You've experienced this before; when you start an app with the `streamlit run` command, this is what the command actually does at the end—it opens up a web browser for you and navigates to a URL like "<https://localhost:8501>".

You could have done this manually, too. In fact, as long as your Streamlit server is running, opening the URL in a new browser tab or window creates a new connection to the server and a new instance of your app.

Deploying your app, then, involves starting and keeping a Streamlit server running, ready to accept new connections. Only, rather than you accessing your own app through the `localhost` URL, it'll be other people accessing it through a different URL.

There are several approaches by which you can deploy an app. We'll discuss these briefly in the next section.

5.1.2 Options for deployment

Depending on your requirements, how much money you're willing to spend, and how much effort you're willing to put in, there are multiple options you might consider for deployment. Let's consider some of them briefly:

RUNNING A SERVER OVER YOUR LOCAL NETWORK

The simplest way to deploy your app is something you've already done each time you've run a Streamlit app. Recall that when you do this using the `streamlit run` command, a Streamlit server starts, and you can see output similar to the following in your terminal window:

You can now view your Streamlit app in your browser.

Local URL: <http://localhost:8502>

Network URL: <http://192.168.50.68:8502>

As we've seen many times at this point, the "local URL" here lets *you* access your app from the computer it's running on.

But if your machine is connected to a local network or even your home Wifi, other devices on the network can access it through the *network URL*.

Go ahead and try it out! If you're on Wifi (or a LAN) and have another device—like a smartphone or another computer—that's connected to the same Wifi / LAN, try running one of the apps you've created, note the network URL, and open it in the second device's web browser.

For instance, figure 5.1 shows what I saw when I opened my todo-list app from my phone connected to the same Wifi network:

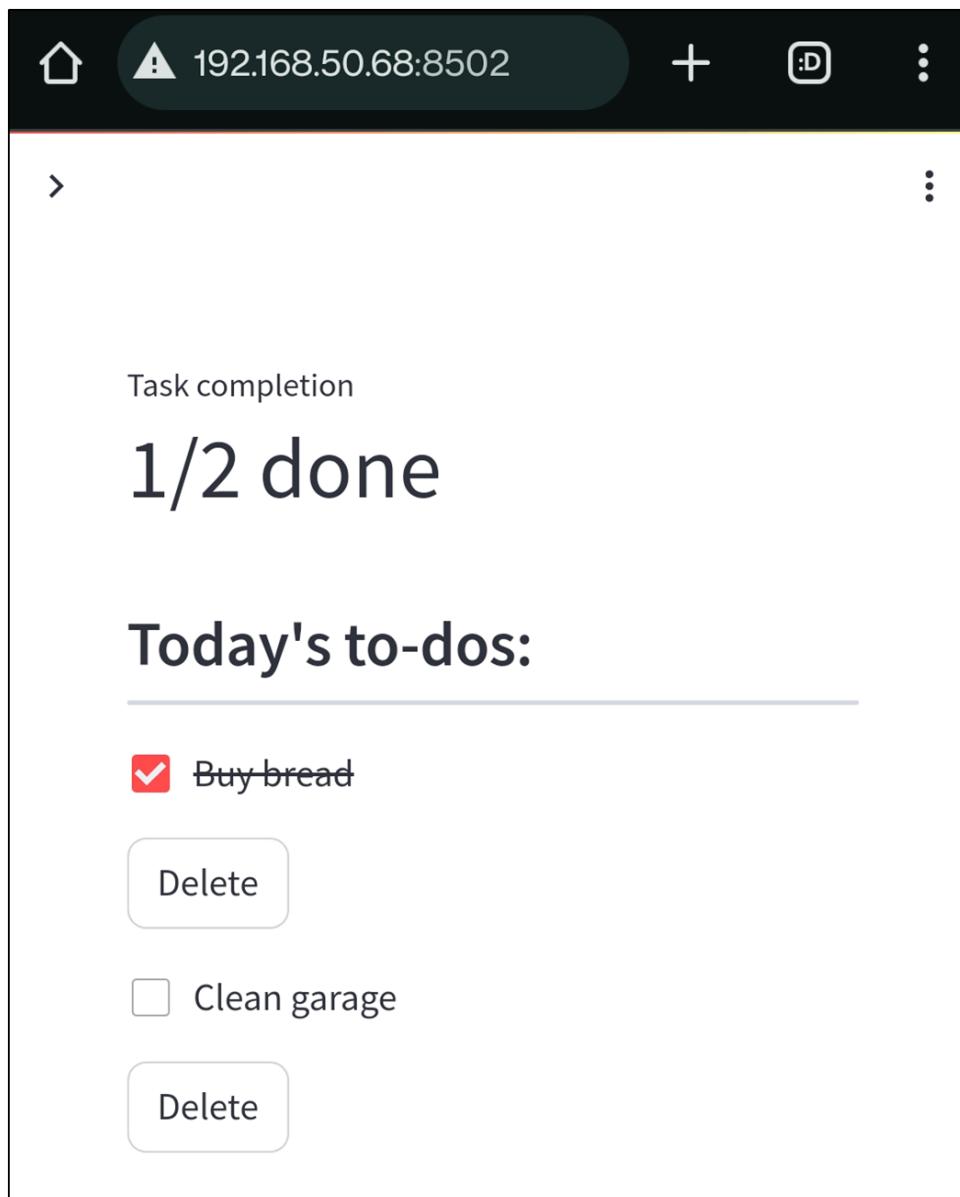


Figure 5.1 Using the network URL of your Streamlit app to access it from a different device connected to the same network

NOTE Getting this to work depends on how your network is set up. For instance, your firewall may block incoming traffic from other devices, thus preventing it from reaching your Streamlit server, or there may be other similar rules in place. Fixing such issues is outside the scope of this book, but you should be able to figure it out with some Googling or help from your network administrator.

One advantage of this method of deployment is that making changes and having them be visible to your users is as easy as editing your code; there are no extra steps!

There are some obvious limitations, however:

- It only works while the computer where you're running the Streamlit server is on and connected to the network.
- It only allows devices connected to your local network to access your app, not the general public.

Still, it can come in handy in a bunch of places. You could create apps for your household, for example, and share the link with your family. You could even use this type of deployment for basic non-business-critical apps at your workplace depending on how lenient your networking and security policies are.

SETTING UP A DEDICATED SERVER

If you're looking to make your Streamlit app available to a broader audience, setting up a dedicated server *may* be a logical step beyond local deployment. This involves using a separate physical or virtual machine that runs independently of your personal computer. By doing this, you can ensure your app is available around the clock and accessible to users outside your local network.

In this setup, you start by selecting a suitable server—this could be a repurposed extra computer you own or a new machine set up specifically for this purpose. After choosing a server—and installing Python and Streamlit on it—you would launch the Streamlit server for your app, and expose the correct port (e.g., port 8501) to external traffic. You'll also need to handle network configuration to allow access, such as setting up port forwarding on your router if the server is behind a firewall.

Operating a dedicated server can be a daunting undertaking that comes with many responsibilities, especially related to security. You would be responsible for configuring firewalls, maintaining and updating your server's operating system and software, etc.

The advantage of going this route is that you'll have complete control over your deployment, but on the flip side, it requires a lot of technical knowhow and probably more importantly, a big portion of your time.

If you simply want to let the general public use the app you've developed, I would recommend one of the remaining options that we'll discuss.

DEPLOYING TO THE CLOUD

For greater scalability, reliability, and ease of access, you can use a cloud-based platform to deploy your app. This approach leverages the infrastructure of public cloud service providers—such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud—allowing you to host your app without the need for physical hardware. It provides numerous benefits, including automatic scaling to handle varying levels of traffic, robust security measures to protect your data, and high availability to ensure your app is accessible at all times.

The key benefit of cloud deployment is that your cloud provider manages much of the infrastructure-related responsibilities associated with maintaining your application. This includes server maintenance and security updates, allowing you to focus on the development and improvement of your app.

Many companies have already migrated, or are in the process of migrating, their internal applications to the cloud. If you're considering making your app available to users within your organization or to a broader audience, cloud deployment can be an efficient and effective solution. Collaborating with your cloud administrator or IT team will help ensure a smooth setup and integration process.

However, it's important to note that using a cloud provider can be expensive as your app gets popular, since costs are usually based on the resources your app uses, which increases with the number of users accessing it.

Chapter 13 will discuss in detail how to deploy your app to public cloud platforms such as AWS and Google Cloud.

STREAMLIT COMMUNITY CLOUD

That leaves us with the option we'll be using for most of this book—Streamlit Community Cloud, a way to publish your apps to anyone who cares to use them, *completely free of charge*.

Streamlit Community Cloud is run by Snowflake, the company that owns Streamlit. It prioritizes ease-of-use, and as the name suggests, is custom-built to run Streamlit apps.

Community Cloud does come with resource limitations, such as how much computational power, memory, and storage your app can use. If you cross those limits—say, if your app blows up in popularity—you may need to look at a different option, such as deployment to a paid cloud provider (see Chapter 13).

However, given that we're in the process of learning Streamlit, Community Cloud is ideal for our purposes. In the rest of this chapter, we'll go through how to deploy an app to it.

5.2 Deploying our to-do list app to Streamlit Community Cloud

As discussed in the previous section, Streamlit Community Cloud serves our deployment needs perfectly, since it's free, custom-built for Streamlit, and incredibly easy to use.

In this section, we'll deploy one of the apps we built previously—our to-do list app from Chapter 4—to Community Cloud, making it so that anyone with an internet connection can use it.

5.2.1 Prerequisites

In addition to Python and Streamlit itself, deploying an app to Streamlit Community Cloud requires the following:

- `git`, the popular version control tool that we discussed briefly in Chapter 2
- A GitHub account
- A Streamlit Community Cloud account
- Connecting your GitHub account to Community Cloud

If you've never used `git` before and want to get acquainted with it, head over to Appendix B, which touches upon how to use it.

CREATING AND SETTING UP A GITHUB ACCOUNT

You may have heard of GitHub, a web-based platform used for version control and collaborative software development. It uses `git`, a distributed version control system, to help developers track changes in their code, collaborate on projects, and manage versions of their software.

NOTE GitHub and `git` are not to be confused with each other. `git` is the name of the version control system. and GitHub is the most popular platform for hosting repositories created using `git`. You can use `git` with other hosting platforms, such as Bitbucket, though Streamlit Community Cloud does require GitHub.

A link to a GitHub account is a pretty standard fixture on most developers' resume these days. Importantly for us, Streamlit Community Cloud expects that your app's code will be stored in a GitHub *repository*, which consists of a collection of files and directories along with their revision history.

Begin by going to github.com and signing up for a new account. The sign-up process is pretty similar to what you would expect on other websites—you'll have to enter your email and verify it, create a username, and select a strong password.

Once you've created your account, you'll need to enable your command-line to authenticate and push code to any repositories you make. There are several ways to do this, but we'll use Personal Access Tokens (PATs), which are alternatives to passwords meant for accessing GitHub through the command-line or an API.

At the time of writing, to get to the PAT creation screen, you can follow these steps:

- Click on your profile picture and then "Settings"
- Find and click on "Developer settings" in the side panel
- Click "Personal access tokens" > "Tokens (classic)"
- Select "Generate new token"

Figure 5.2 shows this path visually (though of course GitHub may change how it's configured).

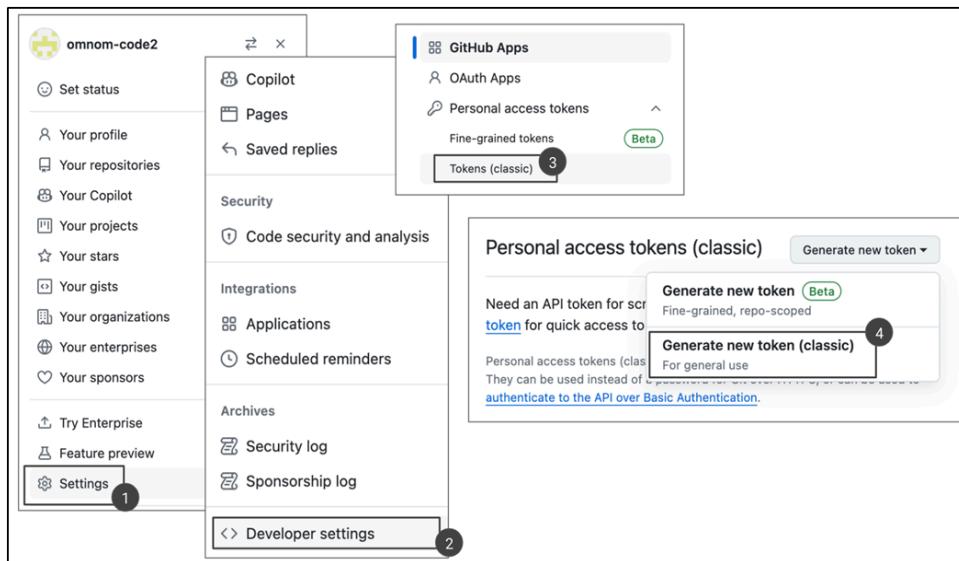


Figure 5.2 How to get to the Personal Access Token generation page on GitHub

In the screen that opens up, make sure to select the "repo" scope, which gives you full control of repositories.

You'll also need to enter a note describing what the token is for (feel free to enter something like "Token to push Streamlit code") and an expiration.

Once a PAT expires, you won't be able to use it any more and will need to create another one, so choose accordingly. A shorter expiry is more secure--as it'll be valid for a smaller period if compromised--but also means you'll have to change it more frequently.

Figure 5.3 shows the selections you could make.

New personal access token (classic)

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

Token to push Streamlit code

What's this token for?

Expiration *

60 days The token will expire on Fri, Oct 4 2024

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input type="checkbox"/> repo:status	Access commit status
<input type="checkbox"/> repo_deployment	Access deployment status
<input type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> security_events	Read and write security events

Figure 5.3 The PAT creation screen on GitHub; make sure to select the "repo" scope

Click "Generate token" to actually create it. GitHub will now show you the token you created. Copy it and store it somewhere safe, because you'll never be able to see it again!

We'll use the PAT later when we're pushing our code.

CREATING A STREAMLIT COMMUNITY CLOUD ACCOUNT

To create a Community Cloud account, go to the Streamlit website, <https://streamlit.io/>, click "Sign up", and follow the instructions. I recommend signing up with your GitHub account.

Once you're done, it should take you to a "dashboard" page.

If you didn't sign up with your GitHub account, you'll see an exclamation point next to "Workspaces" on the top left.

If this is the case, you'll need to separately connect your GitHub account to Community Cloud. As of the time of writing, you can do this by clicking "Workspaces" and then "Connect GitHub account".

If you're already logged in to GitHub, you won't need to do anything else. If you aren't, you'll need to enter your GitHub account credentials.

5.2.2 Deployment steps

Now that you've got your accounts set up, deploying your app is a three-step process:

- Creating a GitHub repo

- Pushing your code to GitHub
- Telling Community Cloud where to look for it

If your app needs to connect to an external service, or if it needs specialized libraries, there are a couple more steps that we'll explore later in the chapter, but the above steps will work for the existing to-do list app that we built in Chapter 4.

If you've already made git a part of your regular workflow as a developer (as we recommended in Chapter 2), you may already have created a repo and pushed your code into it, and can therefore skip ahead to the section "Telling Community Cloud where to find your app."

If not, read the sections below in order.

CREATING A GITHUB REPO

To start, sign in to your GitHub account. The button to create a new repo should be fairly obvious. If you've never created one before in this account, you should see a "Create repository" button as shown on the left of figure 5.4. If you do have some repos already, it should instead display a list of your top repos and you can create a new one by clicking "New", as shown on the right.

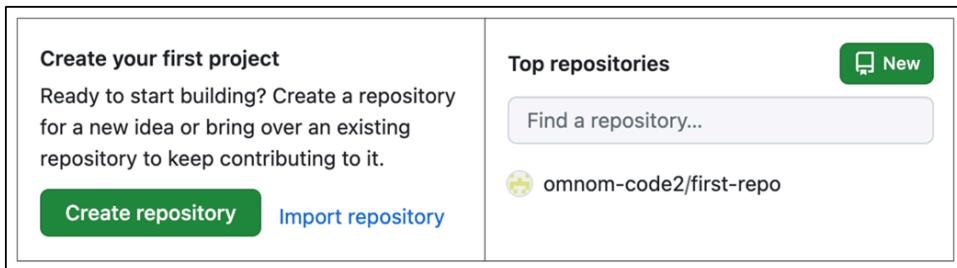


Figure 5.4 Buttons to create a new repo on GitHub

This should lead you to a page that asks you about the details of your new repo (see figure 5.5)

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk ().*

Owner *	 omnom-code2 <input type="button" value="Change"/>	Repository name *	<input type="text" value="streamlit-todo-list-app"/> streamlit-todo-list-app is available.
---------	--	-------------------	---

Great repository names are short and memorable. Need inspiration? How about **miniature-barnacle** ?

Description (optional)

 **Public**
 Anyone on the internet can see this repository. You choose who can commit.

 **Private**
 You choose who can see and commit to this repository.

Initialize this repository with:

Add a README file
 This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

 You are creating a public repository in your personal account.

Create repository

Figure 5.5 The repository creation screen on GitHub

You can ignore most settings here—just make sure you give the repo a memorable name and choose "Public" for the visibility.

When you're done, click "Create repository." You'll be taken to a screen with some instructions and importantly, the URL of your repo, as seen in figure 5.6.

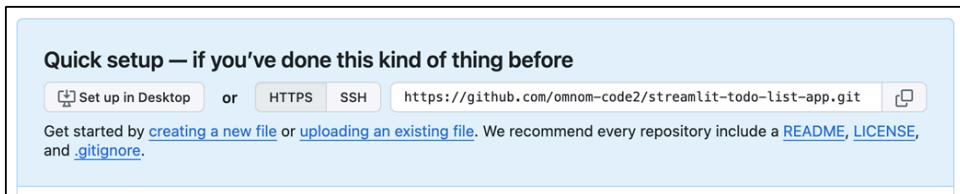


Figure 5.6 Your repo URL

Make a note of this URL as you'll need it in the next section. If you need to, you can find this URL again later by navigating to your repo.

Your repository is now ready for you to put some code in!

PUSHING YOUR CODE TO GITHUB

At this juncture, you have a *remote* GitHub repo, but your Streamlit app code is stored *locally*, on your computer. We'll now make this code available in your remote repo, a process called *pushing* your code.

Open a terminal window and navigate to the directory that contains your to-do list app from Chapter 4, which consists of two files: `todo_list.py` and `task.py`.

Enter the following command to initialize this directory as a *local* Git repository (as opposed to the remote one on GitHub):

```
git init
```

The `git init` command creates an empty Git repo, setting up all the required files in a hidden subdirectory called `.git`.

Next, type in:

```
git add .
```

This adds the contents of the current directory to Git's *staging area*, which is a temporary holding place for changes you make to your code. You use the staging area to prepare the exact snapshot of the code you want to save.

It's time to *commit* your changes, so enter:

```
git commit -m "Commit Streamlit to-do list app"
```

This command captures a snapshot of your code's current state, saving it in the local repository with a descriptive message about the changes.

The last step here is to actually copy your code to GitHub. For this, we'll first need to make your local Git repo aware of your remote repo on GitHub.

Fetch the PAT you created in section 5.1.1 as well as the URL to your remote repo, and combine them to create a PAT-embedded GitHub URL:

```
https://<Your PAT>@<Repo URL without the https://>
```

For example, if your PAT is `ghp_fLbbSwjMlw3gUs7fgRux8Ha3PIlG9w3ZY3zY` (not a real one) and your repo is `https://github.com/omnom-code2/streamlit-todo-list-app.git`, your PAT-embedded URL will be:

```
https://ghp_fLbbSwjMlw3gUs7fgRux8Ha3PIlG9w3ZY3zY@github.com/omnom-code2/streamlit-todo-list-app.git
```

You can now add this URL as a *remote* to your local repo by typing:

```
git remote add origin <PAT-embedded URL>
```

or in our example:

```
git remote add origin https://ghp_fLbbSwjMlw3gUs7fgRux8Ha3PIlG9w3ZY3zY@github.com/omnom-code2/streamlit-todo-list-app.git
```

This tells Git to add a remote repository with the alias "origin" to your local Git configuration, associating it with the specified PAT-embedded URL. This allows you to interact with the remote repository using that alias in future Git commands, and automatically uses the PAT for authentication.

Finally, run the following command to perform the code push:

```
git push -u origin master
```

This does two things:

- Pushes the local branch you're currently in (called "master" by default) to the remote repo you designated as "origin," thus making your code available in the "master" branch of the remote repo.
- Sets the default *upstream branch* (`-u` is shorthand for `--set-upstream`) of your local repo to the "master" branch on the remote repo, so that in the future you can push your code with just `git push` without the `-u` `origin main`.

NOTE We're assuming here that the default branch that Git creates in your repo is called "master." Some versions of Git use the name "main" instead. If you get an error when you use "master," try replacing it with "main." Your command would then become: "git push -u origin main". Alternatively, you can determine the name of the branch you're on by typing "git branch" (your current branch will be highlighted), and use that.

If you navigate to the repo you created on GitHub, you should now be able to see your code, as displayed in figure 5.7

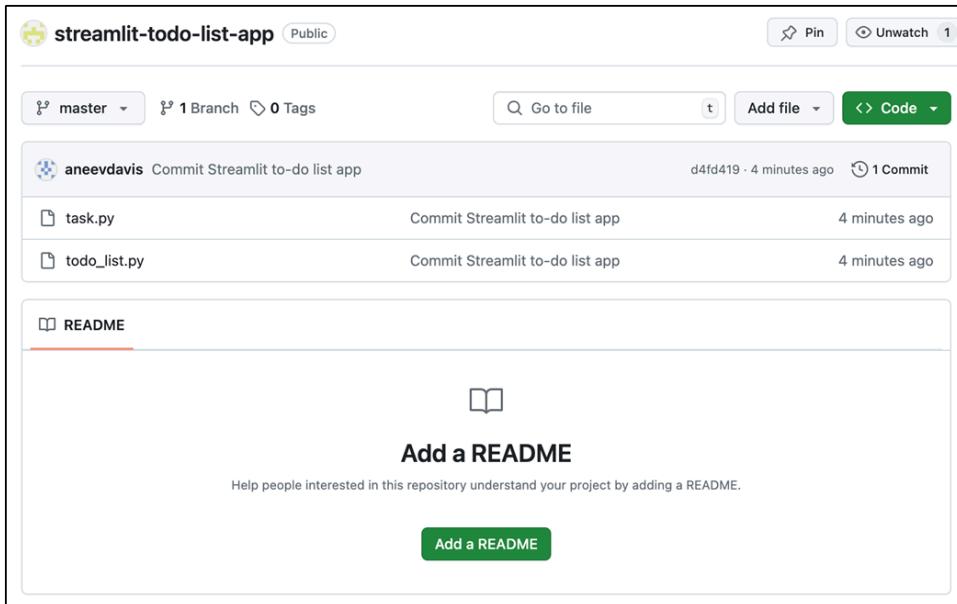


Figure 5.7 Your repo in GitHub, after pushing your code

Git can be quite a complex tool to work with. What I just described in this section is really only the bare minimum of Git you need to know to deploy your code to Streamlit Community Cloud, but ideally, you should be using it as part of your regular development workflow, committing your code each time you complete a meaningful piece of work. This requires you to be familiar with the various Git commands and options. If you'd like to develop a good mental model for working with Git, check out the tutorial in Appendix B.

TELLING COMMUNITY CLOUD WHERE TO FIND YOUR CODE

We've set up our Streamlit code on GitHub. All we need now is to tell Streamlit Community Cloud where to look for it.

Log in to your Community Cloud account at `streamlit.io` and click the "Create an app" button on the top right. If it asks you whether you already have an app, select the option that says you do.

This should take you to the "Deploy an app" page shown in figure 5.8.

Repository [?](#)

Paste GitHub URL

Branch

Main file path

App URL (optional)

stmlit-todo-list-app-1 .streamlit.app

Domain is available

Advanced settings

Deploy!

Figure 5.8 The app deployment screen on Streamlit Community Cloud

Fill out the details of where you pushed your code, including:

- the GitHub repository you created
- the branch you pushed your code to (e.g. `master` from `git push -u origin master`)
- the path to your app (this would be `todo_list.py` since that's the file you use in the `streamlit run` command, and it's located in the root directory of your repo)

In the App URL field, you can choose the address that people can use to access your app. Streamlit will suggest a default URL, but you can override it to something more meaningful. In figure 5.8, I chose `stmlit-todo-list-app-1.streamlit.app`.

There are some advanced settings available that we'll take a look at later in the chapter, but ignore them for now.

That's it! Go ahead and click the "Deploy!" button.

After a minute or so, your app should be ready! Anyone with an internet connection can now visit the address you chose (<https://streamlit-todo-list-app-1.streamlit.app/> in this case) to run your to-do list app and get their life in order. You difference-maker, you!

5.3 Deploying an app that uses an external service

As we saw in the last section, deploying a simple app to Streamlit Community Cloud follows a logical path and is quite straightforward. Our to-do list app is "simple" in the sense that it is fairly self-contained.

For one thing, other than Streamlit and Python itself, it has no other libraries or software that it depends on. For another, it does not interact with any external service or API.

This is not going to be true for most practical apps you create. In the real world, you'll build software on top of software built by other people. Therefore, it's quite likely that your business logic will need third-party libraries that don't come pre-installed with Python.

You will also find that, often, your app needs to reach out to services on the internet to do something useful. In fact, it's almost inevitable. At some point, your app will need access to an English dictionary to check if a word entered by a user is valid, or currency exchange rates to show a price in a different currency, or news headlines from around the globe. In these cases, you'll usually need to sign up for and connect to some kind of Application Programming Interface (API) that provides the specific service you need.

In this section, we'll add some of this complexity to our existing to-do list app and see how to deploy the changes correctly.

5.3.1 A quote-of-the-day to inspire users

Imagine a user of your to-do list app starting their day. The sky is clear, the birds are chirping, and they have a clean slate in front of them. The *diem* is theirs to *carpe*. So, humming the spring portion of Vivaldi's *Four Seasons*, they begin adding their tasks.

Fast forward five minutes, and they've added the eighteenth thing they just remembered has to be done *today*. Shoulders sagging, your user is now vaguely humming the Darth Vader tune from Star Wars in their head, slowly realizing how much of an uphill climb the day is going to be.

Well, we can't have that! What if our app could provide a word of comfort to the user in their time of need, a quote to motivate them? Might we not find a choice piece of wisdom to share with them—something like "*The secret of getting ahead is getting started*"—to inspire them to rise to the occasion?

Of course, some of your more cynical users will just throw their coffee mugs at the screen, but hey, you can't please everyone.

Regardless, let's think about what adding a quote-of-the-day to our app will entail. We could hardcode a bunch of quotes in our code, but that seems wasteful and not very scalable. Instead, we'll use a public API to obtain the quotes.

APIS AND HOW TO CALL THEM

An API is really just a fancy term for a set of instructions that lets different pieces of software talk to each other. You might remember that in Chapter 3, we defined an "API" for the backend of our unit conversion app which was a *contract* that defined how our frontend was allowed to interact with it.

In general parlance, "API" means pretty much the same thing, except that rather than the contract between two parts of the same application, we're referring to the contract that defines how any piece of software is allowed to interact with a particular external service.

An API can be structured however its developer wants and the only perfect way to understand how to use it is to read the documentation. That said, there are some common *conventional* API patterns you'll encounter quite frequently.

Generally speaking, you can *call* this type of API by sending an HTTP (or web) request to a URL that takes the following form:

```
https://<base address>/<endpoint>
```

The base address is a common address that's part of every request sent to the API, whereas the endpoint is specific to the type of request you're making. For instance, a weather-related API might have `api.weathersite.com` as the base address, and `forecast` and `history` as the endpoints for weather predictions and past weather data respectively.

You would normally customize your API request by passing key-value parameters either in the URL (in the case of a GET request) or as a request payload (in a POST request). You may also need to pass additional information, such as an API key, as HTTP headers. See the sidebar to learn more about how HTTP works.

In our weather example above, we might want to pass a date as a URL parameter to the API to get the forecast for that date, so that the URL becomes:

```
https://api.weathersite.com/forecast?date=2024-08-01
```

The API would then return a response, often in a format called JSON (JavaScript Object Notation) that you can parse in your code to make sense of it.

In our example, the response might be something like:

```
{  
  "high_temp": 72,  
  "low_temp": 60,  
  "forecast_text": "Nice and sunny, a good day for the park"  
}
```

SIDE BAR: HOW HTTP REQUESTS WORK

HTTP, which stands for HyperText Transfer Protocol, is the set of rules that define how messages are sent across the web. Think of it as the grammatical structure that computers use when they communicate over the web.

To communicate using HTTP (e.g. when a web browser wants to talk to a web server to retrieve a web page), an HTTP *client*—like a browser—sends properly formed HTTP *requests* to a web server. The server then returns an HTTP *response*, which may consist of an HTML file, or an image, or just some text, or pretty much anything else.

HTTP requests consist of:

- a request line (for our purposes, let's just think of this as a URL)
- some HTTP headers, which are key-value pairs providing additional information about how to handle the request
- an optional body, which contains data that the client wants to send to the server

There are several different types of HTTP requests, but by far the most common are GET and POST.

Loosely speaking, GET requests are "lightweight" requests that don't have a body. All of the information about the request is contained within the URL. Any data that needs to be sent to the server is encoded as key-value pairs and attached to the end of the URL like this: ?param1=value1¶m2=value2...

When you enter a query in a search engine, that's usually a GET request. In fact, when you type a URL in your browser's address bar and press return, you're sending a GET request to that URL.

POST requests *do* have a body that contains additional information. This body is often called a *payload*, and may contain things like information entered into a web form, or an uploaded file.

THE API NINJAS QUOTES API

The API we'll be using is from a website called API Ninjas (<https://api-ninjas.com/>). API Ninjas offers free APIs for a variety of services, such as real-time commodity prices, exchange rates, URL information lookups, face-detection in images, and much more.

They make money off the paid version of their APIs, which you have to use if you issue more than 10,000 calls a month. Since we're just learning, we can make do with much less than that, so we'll just use the free version. Specifically, we're interested in their Quotes API, which—according to their documentation—"provides a nearly endless amount of quotes from famous people throughout history."

To access it, you'll need to make an account with them. Head on over to <https://api-ninjas.com/> and sign up. As usual, you'll need to provide and verify your email and create a password.

Once you're signed in, find your API key (you'll have one key that can access all of their APIs), which is a set of characters that identifies you when you connect to the API. As I write this, there's a "Show API Key" button under "My Account". Make a note of the key.

Check the documentation at <https://api-ninjas.com/api/quotes> to get a sense of how to connect to the Quotes API.

The base URL here is `api.api-ninjas.com`, a common one shared across all of their APIs. The endpoint for the Quotes API is `/v1/quotes`, which "returns one (or more) random quotes".

The documentation also mentions that you can pass a category parameter that gives you a quote from a specific listed category. There's also a limit parameter that lets you specify how many quotes you want, but that's a paid feature that we won't use. By default, the API will return one quote which is all we need.

Let's say we want a quote from the category "friendship". We would send a GET request to the URL

```
https://api.api-ninjas.com/v1/quotes?category=friendship
```

As an experiment, try going to this URL in your browser (which, as the sidebar on HTTP requests mentions, sends a GET request). If everything's working as intended, you should get an authentication error message:

```
{"error": "Missing API Key."}
```

That's because we also need to provide the API key we noted earlier. The documentation mentions that this needs to be passed as an HTTP header by the name `X-Api-Key`.

To do this, we'll use a Python library called `requests`.

5.3.2 Using the `requests` library to connect to an API

As we've seen, the API we'll be using to generate inspirational quotes is a web-based one, where calling it requires our code to send and receive messages via HTTP.

We *could* do this with vanilla Python, but we'll instead use `requests`, which is the Python world's go-to library for HTTP communication. `requests` provides a lot of built-in functionality around handling HTTP requests, and has a more user-friendly set of tools than the alternatives.

First, you'll need to install `requests` using the following command:

```
pip install requests
```

Once that's done, verify that it's installed using `pip show requests`, which should display the version of `requests` you have, along with some other information.

Sending an HTTP request using the `requests` module is quite easy: there are simple `get` and `post` methods we can use.

To give this a shot, open up a Python shell (enter `python` or `python3` in the command line) and try the following:

```
>>> import requests
>>> response = requests.get("https://api.api-ninjas.com/v1/quotes?category=friendship")
>>> response
<Response [400]>
>>> response.text
'{"error": "Missing API Key."}'
```

Here we're simply using the `requests.get` method to send the same request we sent through the browser earlier. This method returns an instance of the `Response` class, which encapsulates the HTTP server's response to our request.

We then access the `text` property of `response`, which gets the body of the response as a string. As you can see, the string is exactly what we got before in the browser.

As we learned from the docs, we can pass our API key using the `X-Api-Key` header. The `requests.get` method accepts an argument called `headers` in the form of a regular Python dictionary, so let's get that set up:

```
>>> headers={"X-Api-Key": "+4VJR..."} #A
>>> response = requests.get("https://api.api-ninjas.com/v1/quotes?category=friendship",
headers=headers)
>>> response.text
'[{ "quote": "People come in and out of our lives, and the true test of friendship is
whether you can pick back up right where you left off the last time you saw each
other.", "author": "Lisa See", "category": "friendship"}]'
```

#A Replace the "+4VJR..." with the API key you noted down earlier

Looks like it's working now! When we passed along the API key we obtained from API Ninjas in the `X-Api-Key` header, we got back an actual quote from the API!

There's a lot more to the `requests` library than this. We'll encounter the module quite a few times in the remainder of this book, but if you'd like to learn more, check out the documentation at <https://requests.readthedocs.io/>.

5.3.3 Incorporating quotes in our app

We now have everything we need to add a quote-of-the-day feature to our to-do list app. Since this is quite a bit different from the core functionality of managing your to-dos, it probably makes sense to create a new file, `quotes.py`, for this.

Specifically, we need a function, say `generate_quote`, that takes an API key and pulls a quote from the Quotes API.

Listing 5.1 shows what `quotes.py` might look like.

Listing 5.1 quotes.py

```
import requests

API_URL = "https://api.api-ninjas.com/v1/quotes"
QUOTE_CATEGORY = "inspirational"

def generate_quote(api_key):
    params = {"category": QUOTE_CATEGORY}
    headers = {"X-Api-Key": api_key}
    response = requests.get(API_URL, params=params, headers=headers)
    quote_obj = response.json()[0]
    if response.status_code == requests.codes.ok:
        return f"{quote_obj['quote']} -- {quote_obj['author']}"
    return f"Just do it! -- Shia LaBeouf"
```

We place `API_URL` and `CATEGORY` (set to "inspirational", which is one of the categories the API Ninjas documentation tells us we can use) at the top of the file as easily configurable constants so that if anyone needs to change them later, it's easy to do so without modifying the function itself.

Within the function, you may notice that instead of passing the category directly in the URL using the `?category=` notation, we're doing something different; we're creating a dictionary like this:

```
params = {"category": QUOTE_CATEGORY}
```

and passing that dictionary in the `params` argument of `requests.get`:

```
response = requests.get(API_URL, params=params, headers=headers)
```

The other way works just as well, but this is more readable. Besides, if we ever need to send a POST request (using `requests.post`) instead, the `params` argument works there too, while you can't use the URL method.

Next, rather than using `response.text` as we did in the Python shell earlier, we're using `response.json()`. That's because, from our earlier experiment in the shell, we know that the response from the API is in JSON format:

```
>>> response.text
'[{"quote": "People come in and out of our lives, and the true test of friendship is
whether you can pick back up right where you left off the last time you saw each
other.", "author": "Lisa See", "category": "friendship"}]'
```

JSON is a format that lets you create arbitrary hierarchies of data using text. It's actually pretty easy for Python developers to read a blob of JSON data since the way you'd store the data in JSON is almost identical to the way you would encode that same data in a Python literal.

So in the above value of `response.text`, you can probably tell that the data is encoded as a list (because of the square brackets) that contains one element: a dictionary with three keys: "quote", "author" and "category".

`response.text`, however, is a string. To be able to access the data easily in Python, we need to parse it into a Python object. That's what the `response.json()` method does: it parses the textual body of the response into a Python object, assuming the text is JSON.

```
quote_obj = response.json()[0]
```

Once we have the result of `response.json()`, we can treat it as a normal Python object. In this case, we know the data is a single-element list, so we use `response.json()[0]` to obtain the only element of the list, which is our three-key dictionary.

The next portion checks if the request went through fine:

```
if response.status_code == requests.codes.ok:
    return f"{quote_obj['quote']} -- {quote_obj['author']}
```

Every HTTP response comes with a status code that classifies the result of the request. You may have seen these before. A status code of 200 means "OK", the code 404 means "Not found" etc.

In the above snippet, the `if` statement verifies that the status code corresponds to a successful result, or in this case, that a quote was returned. `requests.codes` provides a more readable way to refer to specific HTTP status codes. `requests.codes.ok` just means the code 200.

If everything's fine, we construct a string with the quote itself, as well as the author, separated by "--". We use `quote_obj['quote']` and `quote_obj['author']` to extract these from the `quote_obj` dictionary we parsed the JSON result into.

```
return f"Just do it! -- Shia LaBeouf"
```

Finally, if anything went wrong with the API call and the status code was *not* 200, we fail gracefully by returning an evergreen motivational quote from Shia LaBeouf, one of the great philosophers of our time.

Once `quotes.py` is complete, using it to create a quote to display in our main app is pretty simple. In `todo_list.py`, import the `generate_quote` function at the top:

```
from quotes import generate_quote
```

Then, to display the quote, call `generate_quote` with your API key right above where you're showing the task completion metric, and put the result in an `st.info` box:

```
st.info(generate_quote("+4VJR..."))
```

As before, don't forget to replace the argument to `generate_quote` with your actual API key.

If you `streamlit run` your app at this point, you should see your quote-of-the-day show up as in figure 5.9.

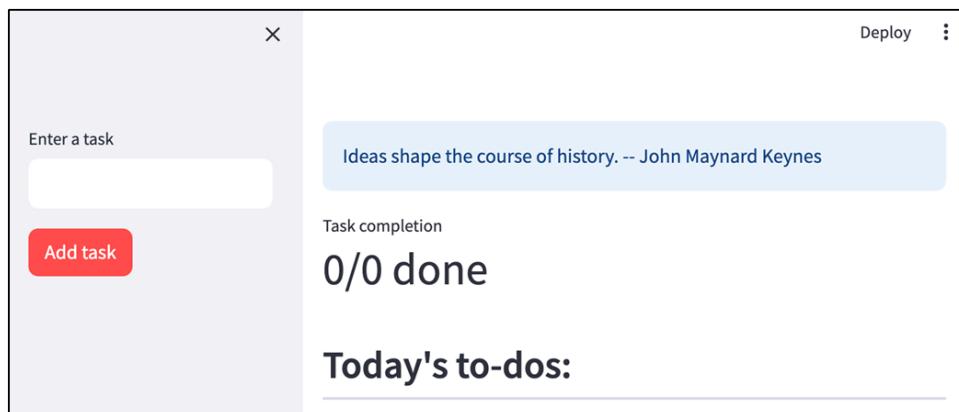


Figure 5.9 Quote-of-the-day in our to-do list app

Great! That seems like our API call is working exactly as we intended! We can rest assured that if our user feels like giving up in the middle of organizing their day, they'll have a daily motivational quote to inspire them to stay strong.

But wait! Recall that Streamlit reruns your *entire* code each time any interaction takes place. That includes making an API call through `generate_quote`. Basically, each time your user adds a task, or marks something done, or basically does anything in the app, an API call is made and a new quote is generated. Forget quote-of-the-day, we're dealing with quote-of-the-click here.

Normally, you might be forgiven for wanting to chalk that up as "a feature, not a bug". But remember that we only have 10,000 free API calls available per month. If we keep pulling a quote every time the user clicks on anything, we'll burn through even that hefty quota pretty quickly.

Instead, let's make use of our old friend `st.session_state` from Chapter 4. Just as we saved our task list so it doesn't get reset with every run, we'll also save our quote so we only call the API once:

```
if "quote" not in st.session_state:  
    st.session_state.quote = generate_quote("+4VJR...") #A
```

#A Again, replace the "+4VJR..." with the API key you noted down earlier

We can now replace our earlier `st.info` line with this:

```
st.info(st.session_state.quote)
```

If you try the app now, you'll see that the quote doesn't change when you add a task or do anything else in the app, only when you refresh the page.

So are we done? Well, not quite...

5.3.4 Accessing your API key safely with st.secrets

Our app works just fine now, but it has a glaring security flaw. We just embedded our API key directly in the code.

If we now commit this code to a public GitHub repository, anyone will be able to see it and therefore find our API key! In this instance, with a free API, the stakes are pretty low, but imagine if you were paying for the API—either because you exhausted your free quota, or because you're using a different API that costs money!

Anyone who accesses your GitHub repo might—unwittingly or with malicious intent—set themselves loose on your API account, costing you hundreds or maybe even thousands of dollars.

NOTE It's worth internalizing this rule right now: never store any secret information, whether it's an API key or a password, or any other kind of credential, in your code—especially if the code is going to be publicly accessible.

What's the alternative then? How do we give our app access to our API key if we can't put it in code? Generally speaking, there are a few ways to do this: you could put your key in an environment variable that's stored securely on the server where your app is running. Another option is to use a secrets management service like AWS Secrets Manager or HashiCorp Vault which allow you to retrieve confidential info securely.

Fortunately, Streamlit provides an easy way to keep your secrets safe: `st.secrets`.

Like `st.session_state`, `st.secrets` is a dictionary-like structure that you can use to store secret information as key-value pairs.

To make a value available to `st.secrets`, you put it in a special file called `secrets.toml`. While you're developing locally, Streamlit will pull values from the `secrets.toml` file. When you push your code, you take care to *not* push the `secrets.toml` file.

Instead, you would configure the information directly into Streamlit Community Cloud, so that your deployed app can access it in production.

THE SECRETS.TOML FILE

In the root folder of your local repository (where you ran `git init` earlier), create an empty directory called `.streamlit`, and under it, an empty text file called `secrets.toml`.

Next enter your API key in the file as shown in listing 5.2.

Listing 5.2 secrets.toml

```
[quotes_api]
api_key = "+4VJR..."
```

As before, use your actual API key here.

The `.toml` format may be new to you. TOML stands for Tom's Obvious Minimal Language. It's a format for configuration files designed to be concise and easily readable by humans, and was originally created by a guy named—believe it or not—Tom.

`.toml` files mostly consist of key-value pairs and support a range of simple and complex data types, including arrays and tables. They can be divided into sections denoted by square brackets.

The file shown in listing 5.2 has a single section, `quotes_api`, with a single key-value pair (with the key being `api_key`).

If we were to read it in Python (outside of Streamlit), you would usually do this using the `toml` module), it would map to the following nested dictionary:

```
{'quotes_api': {'api_key': '+4VJR...'}}
```

In Streamlit, this is more or less what `st.secrets` would also contain.

We can therefore now replace the part of `todo_list.py` that calls `generate_quote` with this:

```
if "quote" not in st.session_state:  
    api_key = st.secrets["quotes_api"]["api_key"]  
    st.session_state.quote = generate_quote(api_key)
```

At runtime, Streamlit will read our `secrets.toml` file to populate `st.secrets` as shown above, and we can use `st.secrets["quotes_api"]["api_key"]` to refer to the API key. Try it out—you'll see that your app is still able to fetch a quote from the API.

NOTE Technically, the `.streamlit` directory should be located in the directory that you run the command `"streamlit run <file_path>"` from. For instance, if you're currently in a directory called `"apps"`, your actual code is located in a folder called `"todo_list_app"`, and the command you use to run your app is `"streamlit run todo_list_app/todo_list.py"`, then the `.streamlit` folder should be located in the `"apps"` folder, not in the `"todo_list_app"` folder. When you deploy your code to Streamlit Community Cloud, `"streamlit run"` will effectively be run from your repo root, which is why I suggested placing `.streamlit` there. While developing locally, if your local repo root is different from the folder you usually execute `"streamlit run"` from, you'll probably need to `"cd"` into the repo root to get this to work.

At the end of this, your `todo_list.py` file should be as shown in listing 5.3

Listing 5.3 The final state of todo_list.py

```
import streamlit as st
from task import Task
from quotes import generate_quote

if "task_list" not in st.session_state:
    st.session_state.task_list = []
task_list = st.session_state.task_list

if "quote" not in st.session_state:
    api_key = st.secrets["quotes_api"]["api_key"]
    st.session_state.quote = generate_quote(api_key)

def add_task(task_name: str):
    task_list.append(Task(task_name))

def delete_task(idx: int):
    del task_list[idx]

def mark_done(task: Task):
    task.is_done = True

def mark_not_done(task: Task):
    task.is_done = False

with st.sidebar:
    task = st.text_input("Enter a task")
    if st.button("Add task", type="primary"):
        add_task(task)

st.info(st.session_state.quote)

total_tasks = len(task_list)
completed_tasks = sum(1 for task in task_list if task.is_done)
metric_display = f"{completed_tasks}/{total_tasks} done"
st.metric("Task completion", metric_display, delta=None)

st.header("Today's to-dos:", divider="gray")
for idx, task in enumerate(task_list):
    task_col, delete_col = st.columns([0.8, 0.2])
    label = f"~~{task.name}~~" if task.is_done else task.name
    checked = task_col.checkbox(label, task.is_done, key=f"task_{idx}")
    if checked and not task.is_done:
```

```

    mark_done(task)
    st.rerun()

elif not checked and task.is_done:
    mark_not_done(task)
    st.rerun()

if delete_col.button("Delete", key=f"delete_{idx}"):
    delete_task(idx)
    st.rerun()

```

5.4 Deploying our changes to Streamlit Community Cloud

Generally speaking, making changes to an already-deployed Streamlit app is simple. All you need to do is to commit your changes and use `git push` to push them to your GitHub repo, and Streamlit Community Cloud will automatically pick them up.

However, given the most recent changes we've made to our app, in our case this process will be slightly more complicated. There are three complicating factors:

- We're now using an additional third-party library, `requests`, and we have to make sure the right version is installed in production
- We need to make sure we don't ever push `secrets.toml` to GitHub, even by accident
- Since we're not going to store our API key in the GitHub repo, we need to configure it in Community Cloud directly

5.4.1 Using a `requirements.txt` file to manage Python dependencies

One of the things that can trip you up when you first start to publish your apps to the world is that you now have two *environments* to manage: your development environment (often simply called *dev*), which is in your laptop or whatever computer you're using to code your app, and your production environment (called *prod*), which is in Streamlit Community Cloud.

To make sure the app that you've coded in dev works as expected in prod, you have to make sure that two environments are configured in exactly the same way, or at least that *any differences between dev and prod are not relevant to your app*.

Obviously, this means that your Python code that runs in prod should be exactly the same as what's running on your computer in dev, but that's not enough; your Python code may rely on software you didn't write yourself, such as the `requests` module in the case of our to-do list app, or indeed, Streamlit itself—this means that we also need to make sure that these *dependencies* of your app are the same in prod and dev, or at least that they don't differ in any ways that matter to your app's functionality.

For instance, if you use a feature of the `requests` library that was introduced in `requests` version 2.1.1 but the version of `requests` you have installed in prod is version 2.0.5, your app will run into errors in prod. To be sure your code won't break, you need to install `requests` version 2.1.1 or higher in prod. Technically, to be really sure your code won't break, you need to install exactly the version 2.1.1, as there's always a small possibility that the feature you used may be deprecated or removed in a future version.

In the Python world, we conventionally use a file called `requirements.txt` to prevent these kinds of discrepancies. The premise of `requirements.txt` is simple: each line in it represents a Python module and a string that specifies a version or a range of versions of that module that is compatible with your code.

When fed into the `pip install` command (as `pip install -r requirements.txt`), `pip` will automatically go through the file and install the right versions of every module in it.

Streamlit Community Cloud knows what to do with a `requirements.txt` file, so if you add one to your GitHub repo, it will automatically install the libraries you need to make your app work.

CREATING A REQUIREMENTS .TXT FILE

Our app only has two outside dependencies—`streamlit` and `requests` (you can determine this by examining the import statements in all of your `.py` files). The contents of our `requirements.txt` file can therefore be quite simple as shown in listing 5.4

Listing 5.4 A simple requirements.txt

```
requests
streamlit
```

While this is a valid `requirements.txt`, it doesn't say anything about what versions we want installed.

One safe way to handle this would be to determine the versions of these libraries we have installed in dev and specify those exact versions in `requirements.txt`.

There are a couple of ways to do this:

- you could type `pip show streamlit` and `pip show requests` to see (among other information) the versions of each library you have.
- you could also enter the command `pip freeze > requirements.txt` to automatically create the `requirements.txt` file from the modules you have installed in dev—this, however, will include every module you have installed, regardless of whether you're importing them in your code. If you go this way, it might make sense to delete all the lines from the file except those that correspond to `streamlit` or `requests`.

Either way, once you've determined the right versions using `pip show` or already created the file using `pip freeze`, your `requirements.txt` file should look something like listing 5.5.

Listing 5.5 A strict requirements.txt

```
requests==2.31.0
streamlit==1.34.0
```

We don't *have* to be this strict with the versions. We could instead allow any version of these libraries that are the same or higher than the ones we have installed. That way, we could allow our app to benefit from any under-the-hood improvements these libraries may make in the future, while still being *reasonably* sure it won't break. In this case, our file may look like listing 5.6.

Listing 5.6 A this-version-or-higher requirements.txt

```
requests>=2.31.0
streamlit>=1.34.0
```

Once you've got your file ready, save it to the root of your local Git repo.

NOTE You may notice that your app works fine in production even if you *don't* create a `requirements.txt` file. That's because Streamlit Community Cloud pre-installs the two libraries we're using here, i.e. `streamlit` and `requests`. Obviously the `streamlit` library is required for any app to function. Since the `streamlit` module actually depends on the `requests` module internally, `requests` is pre-installed too! Still, you'll undoubtedly encounter situations in the future where you need libraries other than those two, and you'll need a `requirements.txt` then. Even in the current case, you'll need this file to restrict the exact versions of these modules that are installed in prod.

5.4.2 Using `.gitignore` to protect `secrets.toml`

You may be wondering about how using a `secrets.toml` file actually protects your API key from prying eyes—the key may not technically be in your Python code any more but it's still stored in a file next to your actual code. Well, the point is to keep `secrets.toml` around locally but to never commit it to our Git repo.

To make certain that we don't accidentally commit the `secrets.toml` file, we'll use a file called `.gitignore`.

`.gitignore` is a simple text file located in the root of your repository that tells Git to ignore certain files and never commit them.

If the file doesn't already exist, create an empty text file called `.gitignore` in the repo root. Then append the path to your `.streamlit` folder. Since the `.streamlit` folder should also be in the repo root, you would simply add the following to `.gitignore`:

```
.streamlit/
```

If you now do a `git add` or `git commit`, your `secrets.toml` file (and indeed, any file in the `.streamlit` directory) won't be committed.

NOTE Git may already be tracking your `.streamlit` folder from before you added it to `.gitignore`. In this case, to remove it from Git's index, you may have to enter this command:

```
git rm -r --cached .streamlit/
```

At this point, your directory structure should look like the following:

```
Root of your Git repo
├── .streamlit (folder)
│   └── secrets.toml
└── .gitignore
└── requirements.txt
└── quotes.py
└── task.py
└── todo_list.py
```

Add your changes to Git, commit them with a message, and then push them to GitHub. You could use the following commands:

```
git add .
git commit -m "Add quote-of-the-day functionality"
git push
```

Note that you can use the simple `git push` command above since you set the upstream of your branch to your remote repo earlier using `git push -u`.

Your changes should now be reflected in your production Streamlit app. Or should it? Navigate to your app's public URL now to see what's shown in figure 5.10.

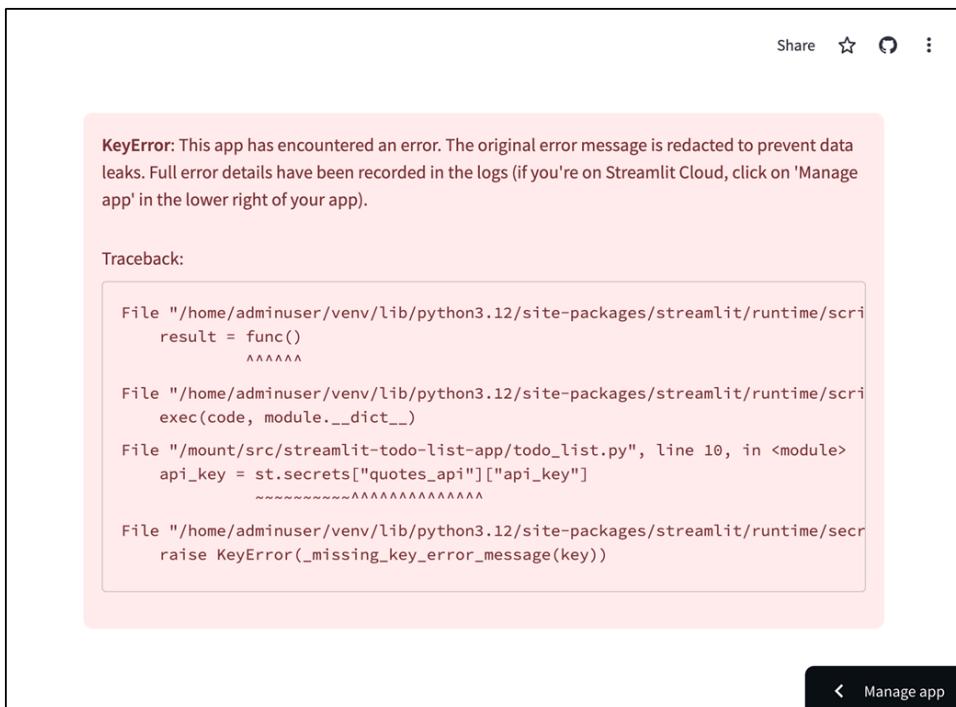


Figure 5.10 Our app throws an error since it's unable to access an API key

Looks like there's a final bit of configuration we have to do before your app works okay.

5.4.3 Configuring secrets in Community Cloud

We've taken pains to keep our API key secret but our app still needs it while it's running in prod. To enable this, we need to configure it in Streamlit Community Cloud where we've deployed our app.

We can do this in the *App settings* screen on Community Cloud. You can access this screen in two ways:

- When you're on your deployed app's public URL, click "Manage app" on the bottom right (you can see it in figure 5.11), click the three vertical dots and then "Settings"
- When logged in to Community Cloud on streamlit.io, click the three vertical dots next to your app in the list of apps, and click "Settings" from there.

Either way, once you have your App settings page open, click "Secrets" on the side panel to get to the screen shown in figure 5.11.

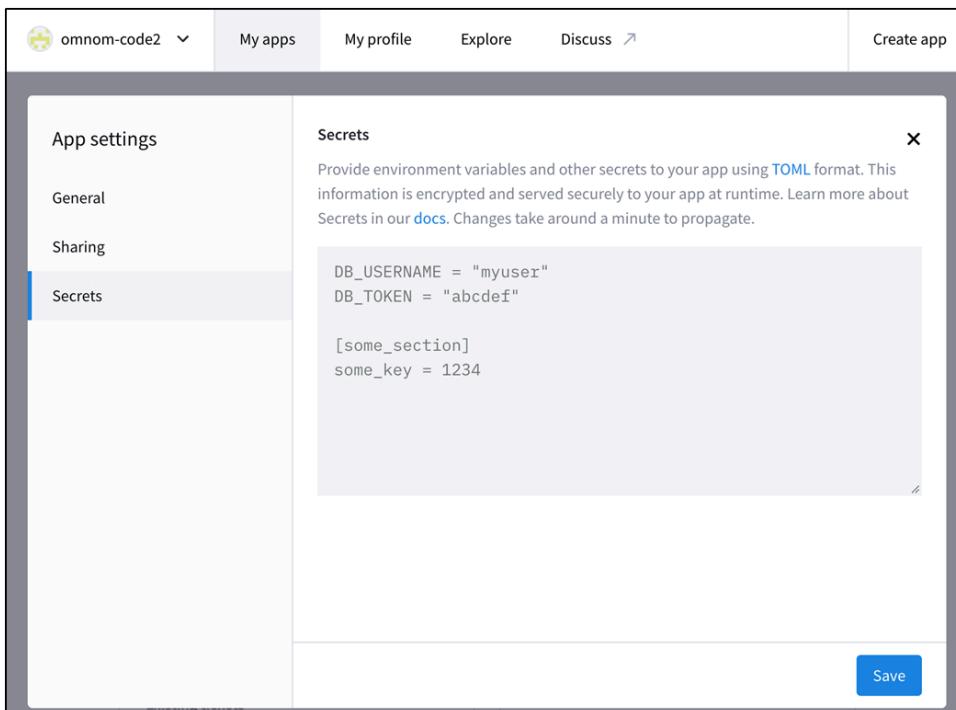


Figure 5.11 The secrets configuration screen on Streamlit Community Cloud

In the textbox under "Secrets", simply copy and paste whatever you have in your `secrets.toml` file on your dev machine, and click "Save".

With that, you're all done! Go to your app's public URL again. It should now be fully functional, as seen in figure 5.12.

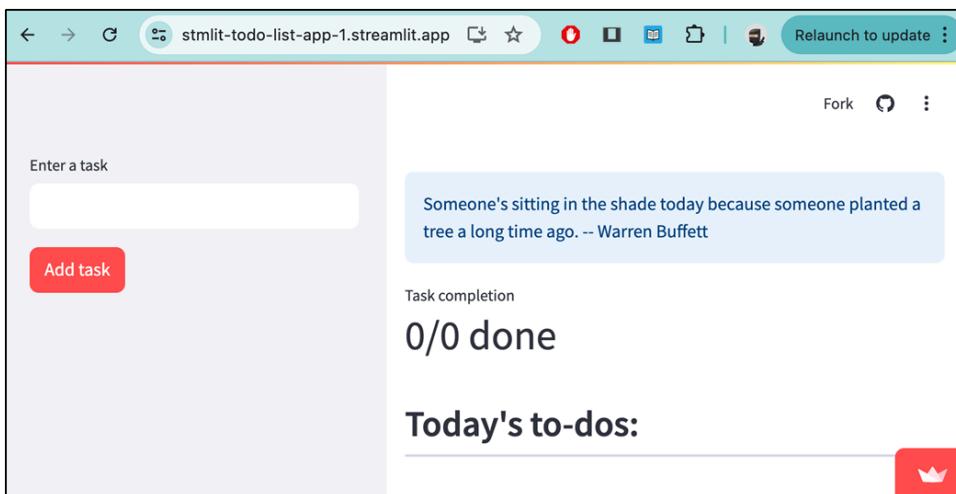


Figure 5.12 Your fully-functional app, deployed to production

By now, you should be equipped to build and share useful Streamlit apps with a public audience. In the next chapter, we'll explore a more advanced app focused on working with data.

5.5 Summary

- Deployment is the process of hosting and setting up your app so your intended users can access it.
- There are multiple ways to deploy an app—you could simply run a server over your local network, set up a dedicated server, or use a cloud provider.
- You can deploy an unlimited number of apps to Streamlit Community Cloud for free, subject to some resource usage limits.
- Deploying to Community Cloud involves creating a GitHub account, pushing your local code to a remote GitHub repo, and telling Community Cloud where to find your code.
- You can connect to external services through the HTTP protocol using a library called `requests`.
- In conjunction with a `secrets.toml` file located within a `.streamlit` folder, `st.secrets` can be used to keep your credentials safe.
- Make sure to never commit your `secrets.toml` file—prevent accidents by adding it to `.gitignore`. Instead, use the App settings page on Streamlit Community Cloud to hold in production the information you store in `secrets.toml` locally.
- You can use a `requirements.txt` file to specify the versions of various Python libraries your app depends on.

- Deploying any changes you make to an already-deployed app on Community Cloud is as simple as pushing the changes to your remote GitHub repo.

6 A dashboard fit for a CEO

This chapter covers

- Building an interactive metrics dashboard
- Wrangling data using the Pandas library
- Caching the results of functions to improve Streamlit app performance
- Creating filters, panels and other widgets in a dashboard
- Developing data visualizations and charts using Plotly

Ever wondered how executives at large companies are able to stay on top of the business they run? Imagine the complexity and the sheer *number* of products and services offered by a company like Amazon, 3M, or Google. How can one person make sense of it all? How do they know if their business is meeting expectations and what areas need their focus?

In well-run companies, the answer—or part of it—is metrics. Executives rely on a carefully curated set of metrics, or numbers that give them a high-level overview of the company’s performance. Metrics help leaders make informed decisions, identify potential issues before they become major problems, and pinpoint areas where the company can improve or innovate.

But metrics alone are not enough—they need to be presented in a clear, digestible way. That’s where dashboards come into play. A good dashboard allows users to explore various cuts of data, transforming raw data into a story, highlighting what’s important and helping leaders stay focused on the bigger picture.

In this chapter, we’ll develop such a dashboard designed for a CEO to monitor key performance indicators (KPIs) for their business. By its end, you’ll not only know how to build a robust, interactive metrics dashboard using Streamlit, but also understand how to present data in a way that empowers decision-makers to focus on what truly matters.

6.1 A metrics dashboard

Change is afoot at Note n' Nib Inc., everyone's favorite fictional online shopping site for stationery! The founder-CEO has retired to enjoy his millions in Ibiza, and his successor, a data-obsessed bigwig from Silicon Valley, has big plans for the company.

His enthusiasm is punctured a little when, in his first staff meeting, he asks the VP of Sales where he can look up the latest sales numbers and the VP fishes into his suitcase and retrieves a *paper* report from two months ago.

"Don't we have a dashboard where we keep track of daily sales data?", the CEO asks, dreading the answer. His fears are confirmed when the VP mutters something about the *old* CEO having possessed a marvelous *intuition* that he frequently relied upon to make decisions—and pen-and-paper being the core of the business after all.

An hour later, the VP of Sales summons you, a rising star in the department (and his go-to for stuff he doesn't want to deal with himself) and tasks you with building "one of those fancy metrics pages with line charts" for his boss.

You know Engineering is busy with a major overhaul of the website, so you don't want to drop this on their plate. Luckily, you've been experimenting with this cool new Python framework called Streamlit and you're raring to use it at work.

6.1.1 Stating the concept

The VP didn't give you a lot to work with, but you've been around long enough to have a pretty good idea of what the boss wants. As always, let's start by spelling this out:

CONCEPT A dashboard that lets executives view various cuts of sales data and track key metrics to make decisions.

6.1.2 Defining the requirements

We have our work cut out for us to translate the concept into concrete requirements. Specifically, the phrases "sales data", "various cuts", and "key metrics" need to be expanded upon.

EXPLORING THE DATA

You've managed to get Engineering to export key data fields from their systems that you think will be relevant to your project into a Comma-Separated Values (CSV) file. You can download this file from the GitHub repo for this book (the file is called `sales_data.csv` and is in the `chapter_6` directory).

Once you've downloaded it, open it up in a spreadsheet program like Microsoft Excel or Numbers on macOS, and inspect the first few lines (shown in Figure 6.1)

date	product_name	segment	category	gender	age_group	state	sales	gross_margin	transactions
2019-01-01	InkStream	Fountain pens	Writing tools	M	18-25	CA	134.91	83.61	8
2019-01-01	InkStream	Fountain pens	Writing tools	M	18-25	TX	149.9	92.9	9

Figure 6.1 The first two rows of the CSV data we'll work with

The data represents the sales of various products sold by Note n' Nib, broken out by various *dimensions* such as segment and category. It also has demographic information about the kind of people who bought these products—specifically their gender and age group, as well as the state they hailed from. The file has data from January 2019 to August 2024, and contains *measures* such as sales (how much revenue did Note n' Nib make), gross margin (how much of the sales was profit, accounting for the cost), and the number of transactions covered by each row.

For instance, here's how we would interpret the first row: On Jan 1 2019, men aged 18-25 from California buying Inkstream fountain pens (classified as "writing tools") made 8 transactions on the Note n' Nib website, bringing in \$134.91 in sales to the company, of which \$83.91 remained after subtracting costs.

The *primary key* (the set of columns that uniquely identify a row) here is the combination of `date`, `product_name`, `segment`, `category`, `gender`, `age_group`, and `state`.

After speaking to the CEO, you also determine that he cares primarily about the following numbers: **total sales**, **gross margin**, **margin percentage**, and **average transaction value**. We'll dive into these and how they're calculated later in the chapter, but these are the "key metrics" from our stated concept.

The CEO wants to be able to see how these numbers differ across different **products**, **categories**, **age groups**, **genders**, and **states**—the "various cuts" from the concept—as well as across time.

You also do some thinking about *how* the data should be represented, and finally come up with an initial set of requirements.

REQUIREMENTS

The user should be able to:

- view in aggregate the total sales, gross margin, margin percentage, and average transaction value of products sold on Note n' Nib
- view how these numbers differ across different years, products, categories, age groups, genders, and states
- filter data by these dimensions to explore cross-sections of the data
- visualize the trends and breakdowns of the metrics over time, broken down by each dimension

You want to deliver an initial version of the dashboard quickly without getting bogged down in additional feature requests, so you clearly define what's out of scope too.

WHAT'S OUT OF SCOPE

At first launch, the dashboard will *not* support:

- Drilling down into the data to view specific rows
- Forecasting future values of any metrics
- Providing explanations about why a metric has changed over time

Some of these can be future expansions to the dashboard, but for now we'll try and limit its functionality to *observing* data as opposed to actively *analyzing* or *predicting* it.

6.1.3 Visualizing the user experience

The next step in the app development flow we explored in Chapter 3 is to visualize the user experience. From the requirements, it's evident that the four key metrics are of vital importance, so we should display them prominently, ideally in such a way that the user's eye is immediately drawn towards them. The requirements also mention filtering the data based on the dimensions and cuts we discussed. It seems reasonable to include a panel where users can do this.

A picture speaks a thousand words, so we want to provide clear visualizations of the data; the trends and breakdowns indicated in the requirements could be realized through *time series charts*, which show the progression of a number over time, and *pie charts*, which tell you how a whole quantity breaks down into its components.

Figure 6.2 shows a mock interface designed based on the above.

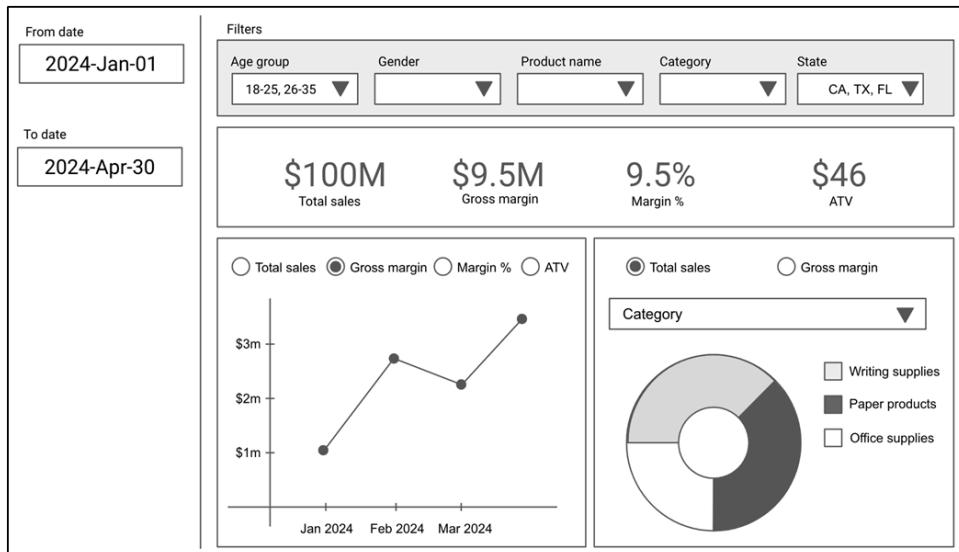


Figure 6.2 A UI mock for the dashboard

As you can see, this design incorporates everything we discussed: the four key metrics are shown in a large unmissable font, and their values correspond to a slice of the data that is controlled by the filters at the top and a date range selector on the left.

There's a line chart that shows how any selected metric has changed over time, and a pie chart that breaks down the total sales or gross margin by a selected dimension, like the product category.

6.1.4 Brainstorming the implementation

Before we get started with actually writing code, let's map out the flow of logic and data in our dashboard at a high level. Figure 6.3 shows one way we might choose to structure this.

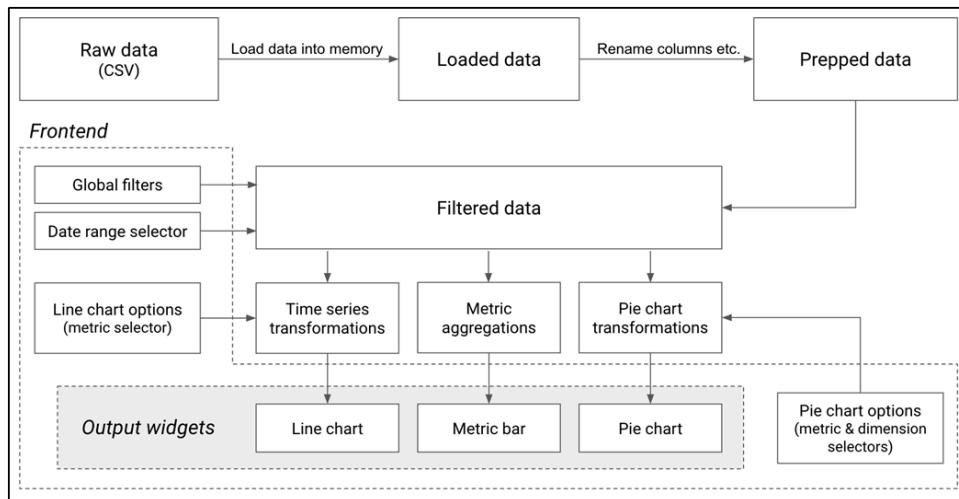


Figure 6.3 Flow of logic and data in the dashboard

The first step is to read the raw data from our CSV file and to load it into memory. There may be some basic data "preparation" or "cleaning" we want to perform next, such as renaming columns for convenience.

In our mock UI in figure 6.2, the filter bar at the top and the date range selectors to the side are meant to be *global*, i.e. they affect all of the widgets in the dashboard. So it stands to reason that we should apply the filters and select just the range we need before we pass the data around to the other parts of the dashboard.

There are three sections in the dashboard where we'll display data in some form: the metric bar with the overall metric values, the change-over-time line chart, and the pie chart. To obtain the content to show in each of these, we need to apply *transformations* to the filtered data, meaning we'll need to group, aggregate and otherwise *transform* it.

The line and pie charts have additional options (you need to choose a metric to display in both charts, and a breakdown dimension—like the product category or gender—in the pie chart), which will serve as inputs to the transformations we'll apply.

This hopefully gave you an overview of the design we'll be implementing in the rest of the chapter. Don't worry if some of these parts aren't quite clear to you yet; we're about to explore each part in a lot more detail.

6.2 Loading the data

In any application that presents or visualizes data, the first step involves obtaining data from some source. Sometimes this source is simply information entered by the user, but often it's an external source such as a database or a file. We'll talk about connecting to databases in later chapters, but for now we'll use the CSV file we reviewed earlier.

In this section, we'll walk through how to load data into your app from an external file, keep it in memory, and display it in Streamlit. Along the way, we'll introduce Pandas, the quintessential Python library for data manipulation. We'll also discuss how to improve your app's performance through caching when the data you need to load is very large.

6.2.1 The Pandas library

I mentioned Pandas in passing in Chapter 1, describing it as a popular library for working with tabular data. In truth, Pandas has become such an integral part of the data ecosystem that it's hard to imagine working with data in Python without it (or something like it at any rate).

INSTALLING PANDAS

You can install Pandas the same way you would install any other Python library: using `pip`. Type the following to set it up now:

```
pip install pandas
```

Once that finishes running, verify that everything's set up right by running `pip show pandas`, which should display some information about the library.

EXPLORING OUR SALES DATA IN PANDAS

Pandas revolves around the concept of the dataframe, which is a two-dimensional, tabular data structure similar to a table in a spreadsheet. It consists of rows and columns, where each column holds data of a specific type (eg. integer, float, string). Dataframes allow for efficient data manipulation and analysis, making them a versatile tool for handling structured data.

To see dataframes in action, let's load our sales data CSV into a Pandas dataframe.

Navigate to the local directory where you downloaded `sales_data.csv`, open a Python shell and type the following commands:

```
>>> import pandas as pd  
>>> df = pd.read_csv('sales_data.csv')
```

The `import pandas as pd` is a regular Python import statement. It's conventional to refer to the Pandas module as `pd` (just as we use `st` for Streamlit).

```
>>> df = pd.read_csv('sales_data.csv')
```

This is where we actually load the CSV file. Pandas makes this incredibly easy using the `read_csv` method. `read_csv` actually has a ton of parameters you can pass it (such as whether the file contains a header, what column names and types to use, and so on), but since all of these have sensible default values, you can also just pass it the path to the file and nothing else.

At the end of this, we have a variable `df` that holds a Pandas dataframe.

Let's verify this using Python's built-in `type` function, which returns the type of the object that a particular variable holds:

```
>>> type(df)  
<class 'pandas.core.frame.DataFrame'>
```

We can also get more information about the dataframe using its `info()` method:

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1035000 entries, 0 to 1034999
Data columns (total 10 columns):
 #   Column        Non-Null Count  Dtype  
--- 
 0   date          1035000 non-null   object 
 1   product_name  1035000 non-null   object 
 2   segment        1035000 non-null   object 
 3   category       1035000 non-null   object 
 4   gender         1035000 non-null   object 
 5   age_group     1035000 non-null   object 
 6   state          1035000 non-null   object 
 7   sales          1035000 non-null   float64
 8   gross_margin   1035000 non-null   float64
 9   transactions   1035000 non-null   float64
dtypes: float64(3), object(7)
memory usage: 79.0+ MB
```

This tells us a variety of things. The dataframe has more than a million rows, numbered from 0 to 1034999. It has ten columns, with the first seven being of the `object` type (strings, essentially) and the last three being `float64`s (or floating-point numbers).

NOTE

These names may be confusing to you since they're not the regular Python types (`str`, `float`, etc.) that you're likely used to. This is because Pandas uses its own data types (or `dtypes`), derived from a related Python library called `numpy`, for efficient computations.

You also have the option of having Pandas use the Apache Arrow format, which can be more performant for large datasets. To do so, while reading the CSV, add a `dtype_backend` parameter like so:

```
pd.read_csv('sales_data.csv', dtype_backend='pyarrow')
```

If you do this, you'll notice that the data types shown by `.info()` are `string[pyarrow]` and `double[pyarrow]` rather than `object` and `float64`.

Let's look at some of the content in the dataframe next. Since we don't have enough space on this page to print all the columns, we first select a subset of them:

```
>>> only_some_cols = df[['date', 'product_name', 'segment', 'state']]
```

You can create a new Pandas dataframe by performing some operation on another dataframe. That's essentially what happened here. When we pass a list of columns to a dataframe using Pandas' user-friendly square-bracket notation (`df[<list of column names>]`), we get a new dataframe containing only the columns we passed, which we can then assign to another variable (`only_some_cols` in this case).

Finally, to see the first few rows, we use the `.head()` method on our smaller dataframe, which shows us the values in the first five rows.

```
>>> only_some_cols.head()
      date product_name      segment state
0  2019-01-01    InkStream  Fountain pens    CA
1  2019-01-01    InkStream  Fountain pens    TX
2  2019-01-01    InkStream  Fountain pens    FL
3  2019-01-01    InkStream  Fountain pens    UT
4  2019-01-01    InkStream  Fountain pens    RI
```

We'll learn more Pandas as we go along, so let's stop here for now and get back to building our dashboard.

6.2.2 Reading and displaying a dataframe

Our dashboard app is going to involve a lot more code than some of the previous apps we've written, so it would be a good idea to spread it across multiple modules or `.py` files.

LOADING DATA FROM THE RIGHT FILE PATH

We'll start with a dedicated Python script file to read in the data from our CSV. Copy `sales_data.csv` to the folder where you intend to keep your code files, and then create `data_loader.py` with the content shown in listing 6.1.

Listing 6.1 data_loader.py

```
import pandas as pd
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent
SALES_DATA_PATH = BASE_DIR / "sales_data.csv"

def load_data():
    return pd.read_csv(SALES_DATA_PATH)
```

The `load_data` function simply uses Pandas to read in the CSV as we saw in the last section, but there seems to be more going on here. How are we populating the `SALES_DATA_PATH` variable? Why couldn't we just set it to `"sales_data.csv"` directly, given that `data_loader.py` is in the same directory as `sales_data.csv`?

The trouble here is that file paths in Python are considered to be relative to the *working directory you're executing a script from*, not the directory that the file containing the line currently being executed is located in.

For instance, if you're currently in `'/Users/alice/'`, your `.py` file and CSV are in the folder `'/Users/alice/streamlit_project/'` and you write `pd.read_csv('sales_data.csv')`, Python will look for the path `'/Users/alice/sales_data.csv'`, which doesn't exist.

You could hardcode the absolute path to the CSV and pass `'/Users/alice/streamlit_project/sales_data.csv'`, but that will obviously create issues when your app is deployed on a different computer where it won't be in that exact path.

No, what we need is a way to refer to the current `.py` file and construct a path relative to that file's path.

This is what the two lines near the top do:

```
BASE_DIR = Path(__file__).resolve().parent
SALES_DATA_PATH = BASE_DIR / "sales_data.csv"
```

`__file__` is a special variable in Python that contains the path of the file currently being executed.

`Path` is a class from the `pathlib` module (which comes built-in with Python) that lets you work with file paths in a user-friendly, object-oriented way. `Path(__file__)` creates a `Path` object corresponding to the current Python script, `data_loader.py`, and `.resolve()` dynamically generates the absolute path to `data_loader.py` (regardless of whether it's in your local machine or a production deployment). The `.parent` then refers to the directory `data_loader.py` is in.

Finally, we use the `'/'` operator which works with `Path` objects (and is *not* the mathematical divided-by operator in this context) to generate the final path to our CSV, stored in `SALES_DATA_PATH`.

It's worth noting that `SALES_DATA_PATH` is not a string, it's still a `Path` object. Fortunately, Pandas' `read_csv` knows how to handle those, so we can pass it directly to that function.

USING ST.WRITE TO DISPLAY THE DATAFRAME

Let's now use the `load_data` function we just created in our Streamlit app. Create an entrypoint (the file we'll use with `streamlit run`) to the app called `dashboard.py`, shown in listing 6.2.

Listing 6.2 dashboard.py

```
import streamlit as st
from data_loader import load_data

data = load_data()
st.write(data.head(5))
```

This part is fairly straightforward. `data` is a Pandas dataframe with data from our CSV (since that's what `load_data` returns).

The last line, `st.write(data.head(5))`, is interesting. We've briefly encountered `st.write` before, in Chapter 2. Streamlit's docs describe `st.write` as "the Swiss Army knife of Streamlit commands" and that's fairly accurate.

You can basically pass pretty much any kind of object to `st.write` and it'll display the passed object in a graceful and sensible way. This means you can pass it a string and it'll write the string to the screen, but you can also pass it a dictionary and it'll print out the contents of the dictionary in a well-formatted way (try it out!). You can even pass it internal Python objects like classes or functions and it'll display information about them.

`st.write` works well for our purposes because we can pass it a Pandas dataframe and it'll show the data on the screen. `data.head(5)` returns a dataframe with only the first 5 rows of the data, and using `st.write` on it helps us verify that the data was loaded correctly, as shown in figure 6.4—which is what you'll get if you execute `streamlit run dashboard.py`.

	date	product_name	segment	category	gender	age_group	state	sales	group
0	2019-01-01	InkStream	Fountain pens	Writing tools	M	18-25	CA	134.91	
1	2019-01-01	InkStream	Fountain pens	Writing tools	M	18-25	TX	149.9	
2	2019-01-01	InkStream	Fountain pens	Writing tools	M	18-25	FL	74.95	
3	2019-01-01	InkStream	Fountain pens	Writing tools	M	18-25	UT	29.98	
4	2019-01-01	InkStream	Fountain pens	Writing tools	M	18-25	RI	0	

Figure 6.4 Streamlit can display Pandas dataframes natively

You may notice that it takes a few seconds for the app to load. This is because our CSV file is rather large (more than 90 megabytes) and reading it takes a while. At first glance, this might not sound like a huge deal, but recall once again that Streamlit reruns your *entire* script every time anything needs to change on the screen.

That means that each time the user changes a selection or clicks out of a textbox, your app will re-read the CSV, slowing down the whole app. Besides being incredibly wasteful, that would degrade your dashboard's user experience, so let's address that next.

6.2.3 Caching data

One of the effects of Streamlit's execution model is that, without intervention, expensive operations such as reading a file or performing a complex computation are executed over and over again to get the same results every time. This can be problematic for data apps like the one we're building now since they frequently rely upon such operations.

In prior chapters, we've seen one way to deal with the problem: we could save the data into `st.session_state`. That way the data would only have to be read once per user session, and the app wouldn't slow down during user interactions.

This is an *okay* solution, but doesn't solve a couple of issues:

- The data would still need to be loaded every time the web page is refreshed. If the user opens the dashboard in multiple tabs—and given the number of browser tabs the average person has open at any point, they probably will—it would take a while to load each time.
- The dashboard would have to read the data from scratch for *each* user even though it's the exact same data.

Streamlit offers a better way to deal with this situation, in the form of `st.cache_data`.

ST.CACHE_DATA

`st.cache_data` is Streamlit's way of *caching* or storing the results of a slow function call so that the next time the function is called with the same parameters, it can simply look up the stored result of the *last* call rather than actually executing the function again.

For our use case, we can simply cache the result of the `load_data` function we wrote earlier. Streamlit would then store the Pandas dataframe it returns, and subsequent app reruns or even page refreshes wouldn't cause the function to be executed again.

`st.cache_data` uses a different Python construct from the Streamlit elements we've seen so far: it's a *decorator*, which you can think of as something that takes a function or a class and adds some new feature to it without you having to rewrite the function or class.

To apply `st.cache_data` on the `load_data` function (in `data_loader.py`), simply write `@st.cache_data` above it, like this:

```
@st.cache_data
def load_data():
    return pd.read_csv(SALES_DATA_PATH)
```

Here, `st.cache_data` is *decorating* the `load_data` function, transforming it with the caching feature so that when it's called again, it'll return the previous cached result rather than executing its logic.

Since we're now referring to a Streamlit element within `data_loader.py`, we also need to include the Streamlit import at the top:

```
import streamlit as st
```

If you run the app now, you'll briefly see a spinning icon with the text "Running `load_data()`." (see figure 6.5) before your dataframe is displayed, but if you reload the page your data should now load instantly.

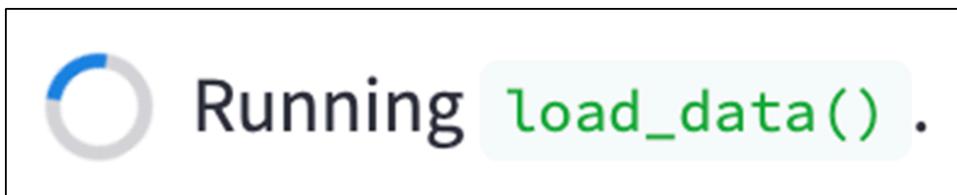


Figure 6.5 By default, `st.cache_data` shows a spinner with a function name when actually running the function

Looks like the caching is working! Here's a question though: what happens when the data changes? The CEO wants a dashboard with *up-to-date* sales data, so we can assume that the source data will change periodically—at least daily if not more frequently than that. For our example, let's assume that Engineering will overwrite the existing CSV with a version with newer data every day.

If the data is cached the way we've set it up, Streamlit won't pull in the updated CSV when it changes. It will just see that the return value of `load_data` is cached from the first run (perhaps several days ago) and use that.

We need a way to set an *expiry date* for the cache, essentially telling it that if the cached data is older than a certain threshold, the function needs to actually be executed and the results re-cached.

We achieve this using the `ttl` argument of `st.cache_data`. `ttl` stands for "time-to-live" and sets the amount of time that the cached data is valid for before Streamlit will re-execute the function during a rerun.

If we assume that our CSV data will change every day, we could set a `ttl` of 1 day like so:

```
@st.cache_data(ttl="1d")
```

This way, the loaded data will be pulled once every day when a user runs the app. That run will take a while, but all subsequent runs in the following 24 hours will use the newly cached data and therefore be quick.

Figure 6.6 illustrates how this works.

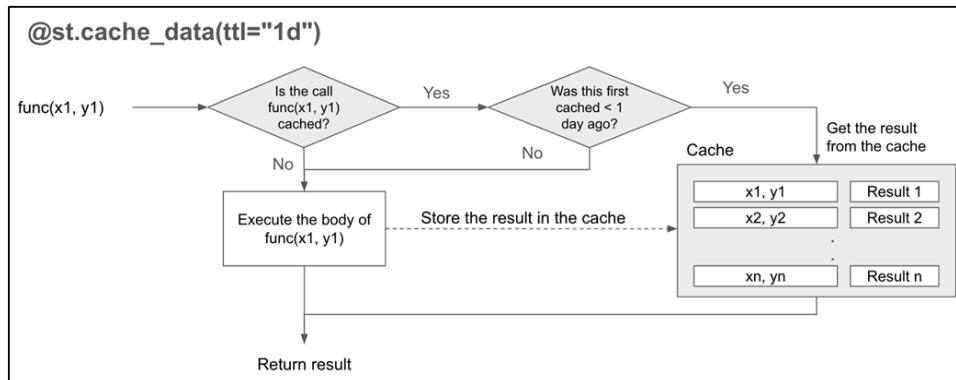


Figure 6.6 How `st.cache_data` works

Another thing we probably want to change is the message displayed with the spinning icon. The name `load_data` is internal to our code and we don't want it exposed to users. We can change this message using the `show_spinner` argument of `st.cache_data` so our code becomes:

```
@st.cache_data(show_spinner="Reading sales data...", ttl="1d")
def load_data():
    return pd.read_csv(LEASES_DATA_PATH)
```

Run the app again and you'll notice the loading indicator has changed to figure 6.7.

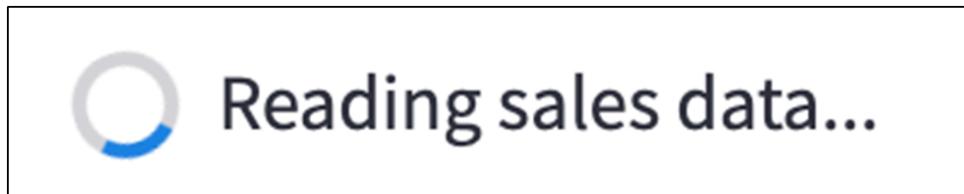


Figure 6.7 Setting the show_spinner parameter in st.cache_data displays a user-friendly message when the data is loading

It's valuable to realize that when a user runs the app and the data is cached, the cached values are available to *all* users of the app, not just the current one. There are times when this can lead to unexpected results (we'll probably encounter some of these in later chapters), but in this case, since we want to show all users the same data, it's desirable behavior.

With the data loaded and available within our app, let's move on to constructing the dashboard itself.

6.3 Prepping and filtering the data

A key requirement our dashboard needs to meet is the ability to inspect various *slices* of the data, as opposed to its entirety. This is quite logical; our source data runs across 5+ years. A user today would probably be more interested in the most recent year of data than older years. Similarly, a user may only be interested in sales related to the "Paper products" category.

Filtering a table of data is the act of considering only the rows of data that we care about, and excluding everything else. Our envisioned dashboard has two areas that enable this: a *filter panel* where users can choose the values of each field they want to consider, and a *date range selector*.

In this section we'll build both of these components visually, and perform the Pandas data wrangling that enables their functionality.

6.3.1 Creating a filter panel

Let's revisit our UI mock, focusing on the filter panel at the top, reproduced in figure 6.8.



Figure 6.8 The filter panel from our UI mock

The panel is essentially a collection of dropdown menus, one for each field that we want to filter by.

Each menu presumably contains the unique values corresponding to each field as options, and the user can choose multiple options (eg. in figure 6.8, the "State" filter has CA, TX and FL all selected). Also, the user can choose not to select *any* option (eg. "Gender" in figure 6.8), which we should probably treat as there not being any filter on that field.

The selections shown in figure 6.8 should cut the data down to rows corresponding to the 18-25 and 26-35 age groups for customers in CA, TX or FL. Since there's no filter on, say, "Product name", the data should include all products sold by Note n' Nib.

OBTAINING THE LIST OF UNIQUE VALUES FOR A FIELD

Clearly, we need a way to populate the options in the dropdowns when we create them. A plausible way to do this might be to hardcode the list of possible values for each field, but this presents some glaring issues: we would need to change our code every time there's a new product or category, or if the way we're grouping ages changes, or for plenty of other reasons.

A better approach is to get the list of options dynamically, from the data *itself*. Start a new Python file called `data_wrangling.py` and add a function to do this, as shown in listing 6.3.

Listing 6.3 `data_wrangling.py`

```
def get_unique_values(df, column):
    return list(df[column].unique())
```

The `get_unique_values` function accepts a Pandas dataframe `df` and a column name `column`. It returns a list of unique values for that column in the dataframe.

When you pass a column name to a Pandas dataframe within square brackets (an operation named *column selection*), you get a Pandas *series*, which is a one-dimensional array-like object (you could think of it as a single-column dataframe).

For instance, `df['age_group']` would return a series with the same number of elements as there are rows in `df`, with each element being the `age_group` corresponding to a row.

Calling `.unique()` on it dedupes the elements and gives you a new series with just the five or six distinct age groups in the data. We finally convert it into a regular Python list with the `list` function.

ADDING THE DROPPDOWN WITH ST.MULTISELECT

As we've seen, the user should be able to select any combination of options in each filter. Streamlit provides this functionality through `st.multiselect`, which is quite similar to `st.selectbox`, which we've come across before. As in the case of `st.selectbox`, the first two parameters (the only required ones) that you pass to `st.multiselect` are the label and the list of options.

For example, you could write `st.multiselect('Color', ['blue', 'green', 'red'])` to display a dropdown labeled "Color" from which you can select one or more colors.

Keeping with our approach of spreading our code into several modules, we'll create a new one for the filter panel and call it `filter_panel.py`.

`Listing 6.4` shows a starting draft of `filter_panel.py`.

Listing 6.4 dashboard.py

```
import streamlit as st
from data_wrangling import get_unique_values

filter_dims = ["age_group", "gender", "category", "segment",
               "product_name", "state"]

def filter_panel(df):
    with st.expander("Filters"):
        filter_cols = st.columns(len(filter_dims))
        for idx, dim in enumerate(filter_dims):
            with filter_cols[idx]:
                unique_vals = get_unique_values(df, dim)
                st.multiselect(dim, unique_vals)
```

`filter_dims` holds the list of fields from the dataframe that we want to filter on.

The `filter_panel` function is what actually displays the dropdowns. It takes a dataframe as input and renders the dropdowns using the unique values for each field.

Some of this code should look familiar by this point. We want to display the dropdowns side by side, so we use `st.columns(len(filter_dims))` to create as many display columns as there are fields we want to filter on.

For each field, we obtain the unique values using our `get_unique_values` function from `data_wrangling.py`, and use them to populate the dropdown:

```
with filter_cols[idx]:
    unique_vals = get_unique_values(df, dim)
    st.multiselect(dim, unique_vals)
```

Listing 6.4 also shows the use of a new Streamlit widget called `st.expander`, which is a collapsible box that users can expand or contract as needed. This makes sense because the user probably doesn't want to see the filters all the time. It's a good idea to have the option of hiding them so they can focus on the actual displayed data.

Let's include the `filter_panel` in our main dashboard by editing `dashboard.py`:

```
...
from filter_panel import filter_panel

data = load_data()
filter_panel(data)
st.write(data.head(5))
```

This should yield the output in figure 6.9.

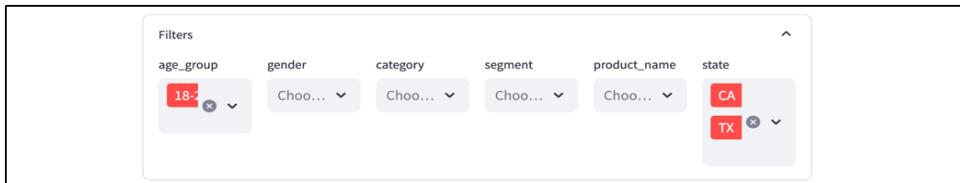


Figure 6.9 By default, our Streamlit app's layout is centered, which can be problematic if there's a lot to display horizontally

One issue we can see immediately is that with six fields to filter on, each individual filter seems to be squished against its neighbors.

Additionally, the filter field names are currently the raw column names from the CSV, including underscores. A more polished design would present these as user-friendly labels, such as "Age group," instead of technical identifiers like "age_group."

FIXING THE WIDTH ISSUE

Notice in figure 6.9 that there's a lot of unused whitespace to the sides of the filter panel. By default, Streamlit apps have a "centered" layout where the body of the app is rendered horizontally at the center of the window, and the sides are blank.

This is usually fine, but with a UI as dense as what we're building, screen real estate comes at a premium. Thankfully, Streamlit allows us to change this and use more of the screen. We can do this using the `st.set_page_config` method in `dashboard.py`.

```
st.set_page_config(layout='wide')
```

Importantly, this only works if it's the very first Streamlit command executed in an app, so make sure it's at the top of `dashboard.py`, right after the imports.

DISPLAYING USER-FRIENDLY LABELS FOR FIELDS

To swap out the raw field names for labels, we could retain a mapping between the raw names and their associated labels in a dictionary, and look up the label every time we needed to display the name of a field. That sounds quite tedious though, so instead let's just rename the fields in the data frame to be user-friendly ones.

There will probably be a bunch of similar "cleaning-up" modifications we'll need to make to the data. We'll bundle these up into a `prep_data` function in `data_wrangling.py`:

```
...
from data_loader import load_data #A

...
def clean_column_names(df):
    df.columns = df.columns.str.replace('_', ' ').str.capitalize()
    return df

@st.cache_data(show_spinner="Reading sales data...", ttl="1d") #B
def prep_data():
    return clean_column_names(load_data())

#A Don't forget to import the functions we need from other modules
#B Move the st.cache_data decorator so it applies to prep_data instead of load_data
```

Here we define a function called `clean_column_names` that replaces the underscores in each column name of a dataframe with spaces and capitalizes it. `df.columns` returns the dataframe column names, and `.str.replace()` and `.str.capitalize()` are the respective string operations. We use `.str` to apply the replace and capitalize operations to each element in `df.columns` in one go, which is something which you'll see quite frequently in Pandas.

For now, the `prep_data` function calls `load_data` internally and then simply returns the result of applying `clean_column_names` to the returned dataframe, but we'll add more logic to it later.

We've also moved `st.cache_data` decorator to the `prep_data` function (remove it from `load_data` in `data_loader.py`) because `prep_data` is supposed to contain basic operations we always want done on the data. After this, Streamlit will cache the result only after prepping the data, not immediately after loading it.

To close the loop, in `dashboard.py`, replace the line `data = load_data()` with `data = prep_data()`.

Finally, in `filter_panel.py`, modify `filter_dims` to use the newly polished field names:

```
filter_dims = ["Age group", "Gender", "Category", "Segment",
               "Product name", "State"]
```

Save and re-run to see the result shown in figure 6.10

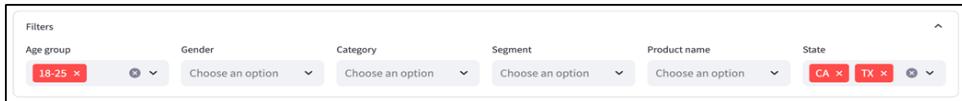


Figure 6.10 Filter panel with user-friendly field names and a wide layout

The dashboard now uses up the full width of the screen, and we're using labels for the filters.

NOTE Due to the amount of code in this chapter and the limited amount of space available on the printed page, while providing code snippets here, we'll focus primarily on how the code in a particular file has *changed* rather than reproducing the entire file as we did in previous chapters. However, at any point, if you're stuck and would like to compare your code to what it should be at that point, you can see complete in-progress snapshots of the code in the GitHub repo for this book (github.com/aneevdavis/streamlit-in-action), under the `chapter_6` folder. For instance, the code files you should be comparing to at the moment are in the `in_progress_4` directory.

APPLYING THE FILTERS TO THE DATA

So far, we've created the UI elements corresponding to the filters, but we haven't actually applied them to the data.

To do this, let's first think of how we should represent the output we get from the filter panel. To be able to apply the filters, we need to look up the values the user has selected in the filter for each dimension. This is a good use case for a dictionary that has each field name as a key and the list of selected options as the value for the key.

Our current `filter_panel` function (from `filter_panel.py`) simply displays the filter bar, but we want to modify it so that it actually returns such a dictionary that we can use for further processing.

```
def filter_panel(df):
    filters = {}
    with st.expander("Filters"):
        filter_cols = st.columns(len(filter_dims))
        for idx, dim in enumerate(filter_dims):
            with filter_cols[idx]:
                unique_vals = get_unique_values(df, dim)
                filters[dim] = st.multiselect(dim, unique_vals)
    return filters
```

Though we weren't making use of this earlier, `st.multiselect` actually returns the list of options a user has selected in the UI. As the highlighted lines show, we're now storing this returned value in the `filters` dictionary using `dim` (the field name) as the key, and returning `filters` at the end.

Next, we need to use the dictionary to produce the slice of the data the user wants. Since this involves data manipulation operations, let's put this functionality in `data_wrangling.py`, in a function called `apply_filters`.

```
def apply_filters(df, filters):
    for col, values in filters.items():
        if values:
            df = df[df[col].isin(values)]
    return df
```

`apply_filters` takes `df`—a Pandas dataframe—and the `filters` dictionary returned by `filter_panel`. It goes through each of the key-value pairs in `filters` and iteratively modifies the dataframe by filtering it using the statement:

```
df = df[df[col].isin(values)]
```

This is worth breaking down a little. The square brackets are quite versatile in Pandas. When you pass a column name in the brackets following a dataframe variable (like `df['Age group']`), it returns that column as a Pandas series—as we've seen previously.

If, instead, you pass a Pandas boolean series (a series where each item is a boolean) instead, it will match the numbered elements of the series against the ordered rows of the dataframe, returning only the rows where the corresponding boolean value is `True`. This is called *boolean indexing*.

You can see both of these usages in the line above:

`df[col]` does column selection, selecting the `col` (the column to filter) in `df`.

The `.isin(values)` applies an element-wise operation on the column, checking whether each value in it is present in values, the list of dropdown options selected by the user. This returns another series with a `True/False` value for every corresponding item in `df[col]`. This is an example of a *vectorized* calculation, which is responsible for much of Pandas' performance.

Finally, we perform boolean indexing, using the boolean series obtained in the previous step to produce a new filtered dataframe and assign *that* to `df`.

Effectively, each time it executes, the line `df = df[df[col].isin(values)]` filters the dataframe to include only rows where the column we're currently looking at contains one of the values the user selected.

The `if values` part ensures that we don't do the filtering if the user hasn't selected any values for a field, thus correctly implementing our requirement.

To see this in action, make the required modifications to `dashboard.py`:

```
...
from data_wrangling import apply_filters, prep_data
...

data = prep_data()
filters = filter_panel(data)    #A

main_df = apply_filters(data, filters)
st.write(main_df.head(5))
```

#A We now put the dictionary returned by `filter_panel(data)` in a variable

We're now capturing the dictionary of filters, using `apply_filters` to create a new dataframe called `main_df`, and displaying that instead of `data`. This should give us the result in figure 6.11.

The screenshot shows a Jupyter Notebook cell. At the top, there is a 'Filters' section with dropdown menus for Age group (18-25), Gender (M), Category (Choose an option), Segment (Staples), Product name (Choose an option), and State (CA). Below this is a table with the following data:

	Date	Product name	Segment	Category	Gender	Age group	State	Sales	Gross margin	Transactions
128,000	2019-01-01	SecureStitch	Staples	Office supplies	M	18-25	CA	12.4	10.04	2
128,050	2019-01-02	SecureStitch	Staples	Office supplies	M	18-25	CA	49.6	40.16	7
128,100	2019-01-03	SecureStitch	Staples	Office supplies	M	18-25	CA	37.2	30.12	5
128,150	2019-01-04	SecureStitch	Staples	Office supplies	M	18-25	CA	37.2	30.12	5
128,200	2019-01-05	SecureStitch	Staples	Office supplies	M	18-25	CA	43.4	35.14	6

Figure 6.11 The selections in the filter panel correctly filter the displayed rows (see chapter_6/in_progress_5 in the GitHub repo for a snapshot of the full code)

As you can see, the displayed dataframe only shows rows corresponding to the user's selections in the filter panel (18-25, M, Staples, CA).

There's still one more kind of filter to apply: the date range! Let's deal with that next.

6.3.2 Creating a date range selector

As usual, Streamlit offers an easy way to enable users to select a date range. In its typical intuitive naming fashion, the widget we want is called `st.date_input`.

`st.date_input` accepts a label, a default value, minimum and maximum values and more. You can have the user select a single date or a range of dates.

For instance, to allow the user to select a single date with today's date as the default:

```
date = st.date_input("Select a date", value=datetime.date.today())
```

To enable a date range selection between a default start and end date:

```
range = st.date_input("Select a date range", value=(datetime.date(2023, 1, 1),
datetime.date(2023, 12, 31)))
```

We'll put our date range selector in a new file called `date_range_panel.py`, shown in listing 6.5.

Listing 6.5 `date_range_panel.py`

```
import streamlit as st
from datetime import date, timedelta

# Hardcode this to the last date in dataset to ensure reproducibility
LATEST_DATE = date.fromisoformat("2024-08-31")
THIRTY_DAYS_AGO = LATEST_DATE - timedelta(days=30)

def date_range_panel():
    start = st.date_input("Start date", value=THIRTY_DAYS_AGO)
    end = st.date_input("End date", value=LATEST_DATE)
    return start, end
```

The `date_range_panel` function displays two date selector widgets—one each for the start and end of the range—and returns the dates selected by the user. We're opting to use two separate single-date inputs here instead of a single range input to prevent temporary errors when the end of the range hasn't been selected yet.

For the default date range values, we show a start date of one month ago and an end date of today, using the variables `THIRTY_DAYS_AGO` and `LATEST_DATE`. Since we're dealing with a static dataset, we hardcode `LATEST_DATE` to a particular date that's available in the CSV. If we were working with real-time or regularly updated data, we would have replaced this with `LATEST_DATE = date.today()`. `THIRTY_DAYS_AGO` is obtained by subtracting 30 days from `LATEST_DATE` using the `timedelta` class from the `datetime` module.

The values `LATEST_DATE` and `THIRTY_DAYS_AGO` are of the type `date`, from the built-in `datetime` module. `st.date_input` understands this type and even returns it. The variables `start` and `end` returned by `date_range_panel` are thus both also of type `date`.

In fact, let's make use of those values presently by editing `data_wrangling.py`:

```
import pandas as pd
import streamlit as st
...

@st.cache_data(show_spinner="Reading sales data...", ttl="1d")
def prep_data() -> pd.DataFrame:
    df = clean_column_names(load_data())
    df['Day'] = pd.to_datetime(df['Date'])
    return df

def get_data_within_date_range(df, start, end):
    if start is not None and end is not None:
        dt_start, dt_end = pd.to_datetime(start), pd.to_datetime(end)
        return df[(df['Day'] >= dt_start) & (df['Day'] <= dt_end)]
    return df

def get_filtered_data_within_date_range(df, start, end, filters):
    df_within_range = get_data_within_date_range(df.copy(), start, end)
    return apply_filters(df_within_range, filters)
```

We've made a few changes here:

In addition to cleaning up the column names, `prep_data` adds a new column called 'Day', the result of applying `pd.to_datetime()` to the existing 'Date' column. Recall that the 'Date' column is currently an opaque "object" type. `pd.to_datetime()` converts it to `datetime[ns]` which can be optimized by Pandas.

We've wrapped a call to `apply_filters` within the `get_filtered_data_within_date_range` function, which first accepts the start and end dates of the date range and uses them to call `get_data_within_date_range`.

`get_data_within_date_range` is the function that actually applies the date range filter on our dataframe. The first line in this function does a date format conversion:

```
dt_start, dt_end = pd.to_datetime(start), pd.to_datetime(end)
```

This is because Pandas uses a `datetime[ns]` type to represent dates. This is different from Python's `date` type that `start` and `end` are in, and is more efficient for use in dataframe operations.

```
return df[(df['Day'] >= dt_start) & (df['Day'] <= dt_end)]
```

This is boolean indexing again, similar to what we encountered in the `apply_filters` function, but we're using a combination of two conditions (`df['Day'] >= dt_start` and `df['Day'] <= dt_end`) to filter the dataframe, joining them with `&`, which is Pandas' element-wise logical `AND` operator.

We also need to update `dashboard.py` again:

```
import streamlit as st
from data_wrangling import get_filtered_data_within_date_range, prep_data
from filter_panel import filter_panel
from date_range_panel import date_range_panel

st.set_page_config(layout='wide')

with st.sidebar:
    start, end = date_range_panel()

data = prep_data()
filters = filter_panel(data)

main_df = get_filtered_data_within_date_range(data, start, end, filters)
st.write(main_df.head(5))
```

We place the date range panel in a sidebar (which you should be quite comfortable with by now).

And since `apply_data` is now contained within `get_filtered_data_within_date_range`, we assign the result of this wrapping function to `main_df`.

Figure 6.12 shows our new date range panel.

The screenshot shows a sidebar with date range inputs and a main area with filters and a data grid.

Start date	2024/07/31	Filters	Age group	Gender	Category
End date	2024/08/31	Choose an option	Choose an option	Choose an option	

	Date	Product name	Segment	Category	Gender	Age group
101,900	2024-07-31	InkStream	Fountain pens	Writing tools	M	18-25
101,901	2024-07-31	InkStream	Fountain pens	Writing tools	M	18-25
101,902	2024-07-31	InkStream	Fountain pens	Writing tools	M	18-25
101,903	2024-07-31	InkStream	Fountain pens	Writing tools	M	18-25
101,904	2024-07-31	InkStream	Fountain pens	Writing tools	M	18-25

Figure 6.12 Date range inputs displayed in a sidebar (see [chapter_6/in_progress_6](#) in the GitHub repo for a full code snapshot)

Our dashboard is slowly starting to take form! It's not very useful to the CEO at the moment, however, since it doesn't show any summary information or metrics. That's up next!

6.4 Calculating and displaying metrics

Imagine you're running a business. How do you know if it's thriving or struggling, whether it's about to go through the roof or come crashing down?

One obvious answer is that you'd look at how much the company is earning and the amount of profit it's making. You'll likely also want to know the rate at which your revenue is growing. If you're a fan of Shark Tank's Mr. Wonderful, you probably also keep track of more esoteric numbers such as how much it costs you to acquire a customer.

All of these numbers are called *metrics*. Metrics are useful because they help you boil down all the complexity of a vast business (or any project, really) into a few figures. If a positive metric (like profit) is going up or a negative metric (like cost) is going down, that means things are going well. If the opposite is happening, something likely needs to change.

In this section, we'll calculate and display the metrics that Note n' Nib's CEO cares about. To do this, we'll first understand what the metrics mean and how to calculate them. We'll then set up a scalable way to define new metrics for our dashboard to use, and display them prominently in the dashboard.

6.4.1 Calculating the metrics

Referring back to our requirements, there are four metrics we care about: *total sales*, *gross margin*, *margin percentage*, and *average transaction value*.

Let's try and understand these in reference to our data. Recall that our source CSV has one row for every combination of date, product (and its associated segment and category), gender, age group and state. For each row, it gives us three numeric fields: *sales*, *gross margin*, and *transactions*.

A transaction refers to a single purchase of some number of items, and the figure "transactions" refers to the number of them represented by a row. Sales is pretty easy to understand: it's the amount in dollars Note n' Nib collected from those transactions. Gross margin is the profit, or sales minus the cost that the company paid to acquire what the items it sold.

Given this, let's figure out how to calculate our metrics for any given slice of our CSV:

- **Total sales** is simply the sum of the sales column
- Similarly, **gross margin** is the sum of the gross margin column
- **Margin percentage** is the gross margin expressed as a percentage of total sales, so it's calculated as $\text{total sales} / \text{gross margin} * 100$
- **Average transaction value** is the what Note n' Nib earned per transaction, so we can calculate it as $\text{total sales} / \text{sum of the transactions column}$

Let's consider a quick example to crystallize this. Imagine you have the following rows:

Date	Product	State	Sales	Gross Margin	Transactions
2024-01-01	Fountain Pen	CA	500	200	10
2024-01-01	Notebook	TX	300	120	5
2024-01-01	Pencil	NY	200	80	8

- *Total sales* is $\$500 + \$300 + \$200 = \1000
- *Gross margin* is $\$200 + \$120 + \$80 = \400
- *Margin percentage* is $\$400 / \$1000 = 0.4 = 40\%$
- *Average transaction value* is $\$1000 / (10 + 5 + 8) = \$1000 / 23 = \$43.48$

6.4.2 Setting up the metrics configuration

A recurring theme you'll come across in software development is that it's a good idea to keep your design reasonably *general*. By this, I mean that if you have a choice between coding something in a very specific way (hardcoding a list of values, for instance) and coding it in a flexible way (populating the list from a configuration file), it's often better practice to do the latter.

A more general design makes your code more adaptable to future changes, and easier to maintain. We saw this when we developed a unit converter app in Chapter 3, where rather than putting the conversion factors directly in our code, we opted to use a configuration file.

This is exactly what we'll do to set up the metrics in our dashboard—define a configuration file that defines how to calculate them.

As we did in Chapter 3, we'll start by creating a dataclass—`Metric`—to hold the object we want to configure. Let's put this in a new file called `metric.py` (see listing 6.6).

Listing 6.6 metric.py

```
from dataclasses import dataclass
@dataclass
class Metric:
    title: str
    func: callable
    type: str
```

The `Metric` class contains a `title`, which is the label we'll display on the interface, and a `type`, which indicates how it should be formatted (eg. "dollars" as the type would instruct our app to prefix a '\$' sign before the number).

It also has a member called `func` which is apparently a *callable*. Callables are essentially just functions, and `func` is meant to be a function that accepts a Pandas dataframe object and calculates the value of the metric.

To truly understand this, let's see how objects of the `Metric` class are defined in our configuration file, `metric_config.py`, shown in listing 6.7.

Listing 6.7 metric_config.py

```
from metric import Metric

def margin_percent(df):
    total_sales = df["Sales"].sum()
    return df["Gross margin"].sum() / total_sales if total_sales > 0 else 0

def average_transaction_value(df):
    total_sales = df["Sales"].sum()
    return total_sales / df["Transactions"].sum() if total_sales > 0 else 0

metrics = {
    "Total sales": Metric(
        title="Total sales",
        func=lambda df: df["Sales"].sum(),
        type="dollars"
    )
}
```

```

),
"Gross margin": Metric(
    title="Gross margin",
    func=lambda df: df["Gross margin"].sum(),
    type="dollars"
),
"Margin %": Metric(
    title="Margin %",
    func=margin_percent,
    type="percent"
),
"ATV": Metric(
    title="Average transaction value",
    func=average_transaction_value,
    type="dollars"
)
}

display_metrics = ["Total sales", "Gross margin", "Margin %", "ATV"]

```

Turn your attention to the variable `metrics`, a dictionary with the names of each of our metrics as keys, and their corresponding `Metric` objects as values. Again, this is reminiscent of `unit_config.py` from Chapter 3, where we did essentially the same thing.

Let's inspect the first item in the dictionary:

```

metrics = {
    "Total sales": Metric(
        title="Total sales",
        func=lambda df: df["Sales"].sum(),
        type="dollars"
),
...

```

This is the "total sales" metric, which has a sensible display label as its `title`, and "dollars" as its `type`. `func` here is a lambda (which, you might remember from Chapter 2, is an anonymous one-line function). It accepts a single argument—`df`, a Pandas dataframe—and calculates total sales using the expression:

```
df["Sales"].sum()
```

`df["Sales"]`, as we've discussed before, selects just the "Sales" column from `df`, and `.sum()` adds up all the values in it to obtain the final value of the metric.

The other metrics defined in the dictionary are fairly similar, and their calculations should make sense after the discussion in the previous section. The "Margin %" and "ATV" metrics *don't* use a lambda for their funcs.

Rather, in each of these cases, `func` points to a regular function (`margin_percent` and `average_transaction_value`) defined above. Since these metrics are both ratios, we need to handle the possibility that the denominator is zero, preventing a division-by-zero error:

```
def margin_percent(df):
    total_sales = df["Sales"].sum()
    return df["Gross margin"].sum() / total_sales if total_sales > 0 else 0
```

The bottom of our configuration file has the following line:

```
display_metrics = ["Total sales", "Gross margin", "Margin %", "ATV"]
```

The variable is meant to hold the metrics we'll actually display in our metric bar. This may seem kind of pointless since we're just listing out all of the keys in the `metrics` dictionary. However, this might come in handy if we ever decided to include more metrics and don't want to display all of them, or want to display them in a specific order. We'll see in a bit how `display_metrics` is actually used.

6.4.3 Formatting the metrics

One of the pieces of information we capture in the `Metric` class is the "type" of the metric, used primarily for formatting.

In our configuration file, three of our metrics are of the "dollars" type, while Margin % is of the "percent" type.

Let's now define the formatting logic in its own module, `formatting.py` (see listing 6.8). For this, we'll be using a third-party library called `humanize` (install it the usual way, i.e. by running `pip install humanize`), which provides some nice formatting features.

Listing 6.8 `formatting.py`

```
import humanize

def format_metric(value, metric_type):
    if metric_type == "dollars":
        return f'${humanize.metric(value)}'
    elif metric_type == "percent":
        return f'{round(value * 100, 1)}%'
    return f'{value}'
```

The `format_metric` function takes a numeric value and a metric type, and returns a formatted display string.

The `humanize` library we just installed comes into play while formatting a dollar-type metric. Since Note n' Nib is such a popular retailer, its sales figures are in the millions. If the raw revenue number we're tasked with displaying is \$25,125,367, we'd rather not show the precise value to the CEO. An abbreviated version like "\$25.1m" will do the trick while reducing cognitive load for the user.

This is exactly what the `humanize.metric` method does. Given a raw number, it will return a rounded number with a sensible lower precision and a suffix ("k" for thousands, "m" for millions, etc). We add the "\$" sign manually to get the f-string `f'${humanize.metric(value)}'`.

For percent-metrics, we multiply the actual value by 100 to convert it from a fraction to a percentage, and round it to a single decimal point before adding the "%" sign.

For anything else, we don't do any formatting and just print the value as-is.

6.4.4 Displaying the metrics

With all the building blocks in place, we can now go ahead and create the metric bar in Streamlit. To do this, create yet another Python module, called `metric_bar.py`, with the code from listing 6.9

Listing 6.9 metric_bar.py

```
import streamlit as st
from metric_config import metrics, display_metrics
from formatting import format_metric

def get_metric(df, metric):
    return metric.func(df)

def metric_bar(main_df):
    with st.container(border=True):
        metric_cols = st.columns(len(display_metrics))
        for idx, metric_name in enumerate(display_metrics):
            metric = metrics[metric_name]
            with metric_cols[idx]:
                value = get_metric(main_df, metric)
                formatted_value = format_metric(value, metric.type)
                c1, c2, c3 = st.columns([1, 3, 1])
                with c2:
                    st.metric(metric.title, formatted_value)
```

The `metric_bar` function iterates through the list of metrics we flagged for display (`display_metrics` from `metric_config.py`), adding an `st.metric` element within a display column for each of them.

That's the gist of it, but there are a few interesting bits here.

The first is the use of `st.container`, a new Streamlit element. Here we're just using it to put the metric bar within a box with a border:

```
with st.container(border=True):
```

There is more to `st.container`, however. One use case for it is when we want to display elements "out of order", or in a different order than they are written in the code. We'll come across this later in the book.

We use the `get_metric` function to extract the value of a metric, given a dataframe and a Metric object. Since each Metric object already has a `func` member that defines how to do this, the body of `get_metric` is as simple as calling it:

```
def get_metric(df, metric):
    return metric.func(df)
```

The last interesting bit is this part:

```
c1, c2, c3 = st.columns([1, 3, 1])
with c2:
    st.metric(metric.title, formatted_value)
```

You may find this rather strange. We've already defined a column for the metric (`metric_cols[idx]`), but it seems like we're splitting that column into three *sub*-columns, and then putting the `st.metric` widget in only the second one! What's the point of this?

Well, this is actually a layout hack. Unfortunately, as of time of writing, Streamlit doesn't offer a great way of horizontally centering items within a column without HTML. So instead, here we're creating three columns within the main column, with the first and third being equal-width blank ones and the second holding the actual content. The overall effect is that the content of the second sub-column appears centered within the main column.

With that, all we need is a quick update to `dashboard.py` and we should be good to move on:

```
...
from metric_bar import metric_bar

...
main_df = get_filtered_data_within_date_range(data, start, end, filters)
if main_df.empty:
    st.warning("No data to display")
else:
    metric_bar(main_df)
```

We've removed the sample dataframe rows from the display and replaced it with the metric bar. Re-run the dashboard to find the output in figure 6.13.



Figure 6.13 Metrics bar showing key metrics in aggregate (see `chapter_6/in_progress_7` in the GitHub repo for the full code)

Let's tackle some of the visualization components of the dashboard next.

6.5 Constructing visualizations

Humans are intuitively visual beings. When you're trying to convey a message through data, it's usually more memorable and *clicks* much faster when you use a graph rather than a table of numbers. This is especially important for busy executives who may have to deal with a breadth of matters and need to develop an understanding of the business so they can make decisions quickly.

In this section, we'll add visualizations to our dashboard in the form of a line chart to show how the metrics we're tracking have changed over time, and a pie chart to show the breakdown of those metrics across a chosen dimension. In each case, we'll use Pandas to wrangle the data into a form we can visualize easily, whip up the actual images using a library called Plotly, and display them using Streamlit.

6.5.1 Creating a time series chart

A time series is simply a sequence of data points recorded at regular time intervals, showing how a particular metric or variable changes over time. This is crucial for spotting trends, seasonality, and outliers, which can inform decision-making.

You can think of a simple time series as a series of data with two variables—one representing a date or time, and another representing the measure we're tracking.

For instance, this is a time series:

Date	Sales
2024-01-01	120
2024-01-02	135
2024-01-03	142
2024-01-04	130
2024-01-05	155

OBTAINING A TIME SERIES FROM OUR DATAFRAME

Recall that the data we're dealing with have many different fields—date, product name, gender, sales, and so on. We're going to have to transform it into the specific shape of a time series before we can pass it to the visualization we'll build.

We have a "Day" field in our data, but one particular row in our dataframe represents the value for a particular combination of gender, age group, product name, etc. What we need is something that, given any particular slice of our full dataframe (which is what our filters give us), *aggregates* the data up to the "Day" level.

For example, a user may have applied the filters "State = CA" giving us the following slice of data (simplified for clarity, excluding the other fields):

Day	State	Gender	Product	Sales
2024-08-01	CA	M	RoyalQuill	1500
2024-08-01	CA	M	GripLink	1300
2024-08-02	CA	M	RoyalQuill	1600
2024-08-02	CA	M	GripLink	1200

Our time series should contain only "Day" and "Total sales", so we need to add up the sales for each date across state, gender and product:

Day	Sales
2024-08-01	2800
2024-08-02	2800

To do this in Pandas, assuming our dataframe is called `df`, we could write the following:

```
grouped = df.groupby('Day')
data = grouped.apply(lambda df: df['Sales'].sum(), include_groups=False).reset_index()
```

Let's break this down. `df.groupby('Day')` would give us a grouped dataframe, which you can imagine as being represented internally like this:

Day	Grouped Rows
2024-08-01	(2024-08-01, CA, M, RoyalQuill, 1500) (2024-08-01, CA, M, GripLink, 1300)
2024-08-02	(2024-08-02, CA, M, RoyalQuill, 1600) (2024-08-02, CA, M, GripLink, 1200)

Next, consider the line `grouped.apply(lambda df: df['Sales'].sum(), include_groups=False)`

The `apply` method is an extremely powerful Pandas construct that lets you apply a function to each row or column of a dataframe or to the values of a series. When used on a grouped dataframe like the one above, it allows you to perform operations on each group separately.

In this case, the `lambda df: df['Sales'].sum()` takes each group (which corresponds to a specific day) and calculates the total sales for that day by summing the `Sales` values. In the example above, the day 2024-08-01 would have a total `Sales` value of 2800, adding up the sales for the RoyalQuill and GripLink rows.

The `include_groups = False` bit indicates that we don't want the function to also operate on group labels (the `Day` values). It's kind of redundant here as our lambda function specifically refers to the "Sales" column, but if you don't include this, Pandas will whine about it.

Finally, the `reset_index()` method converts the results back to a standard DataFrame format. An `index` is a Pandas concept referring to a unique identifier column for each row, enabling efficient data retrieval and alignment; after summing, the `Day` column becomes the index. By calling `reset_index()`, we restore `Day` as a regular column and create a new index that ranges from 0 to n-1, where n is the number of unique days.

We need a slightly more generalized version of the above code for our dashboard, as our line chart may need to show any of our four key metrics, not just sales (see figure 6.14 from our UI mock).

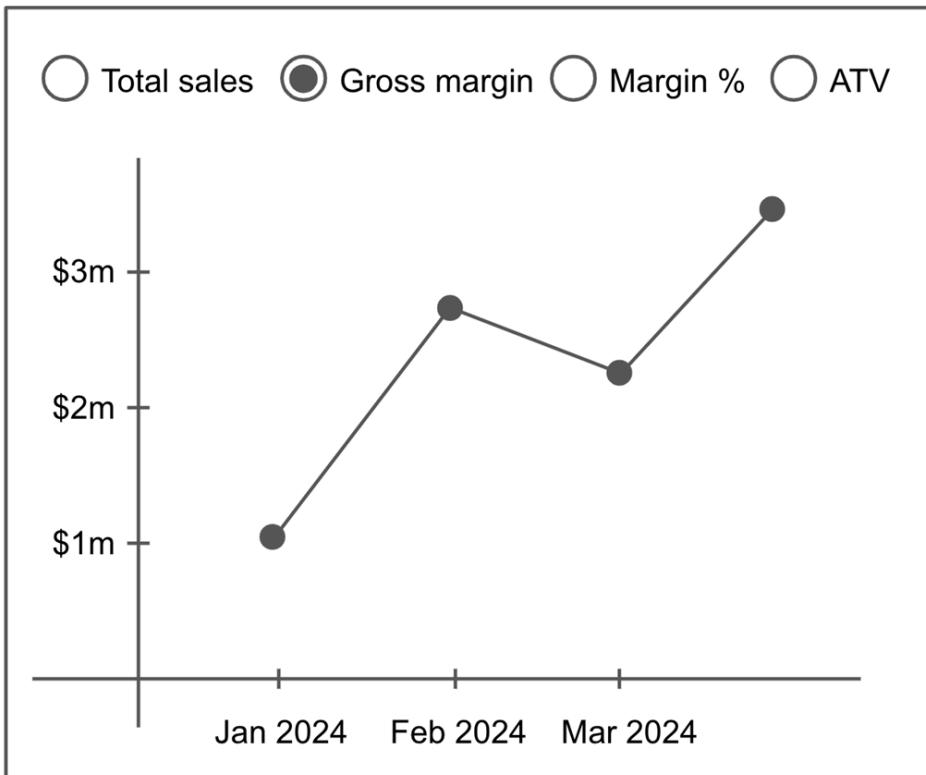


Figure 6.14 The time series chart from our UI mock

So rather than passing in `df["Sales"].sum()` as the function to apply, we'll obtain this function from the `func` attribute of the `Metric` class we defined in `metric.py` and instantiated in `metric_config.py`.

```
Let's add such a function to the bottom of data_wrangling.py:
def get_metric_time_series(df, metric):
    grouped = df.groupby('Day')
    data = grouped.apply(metric.func, include_groups=False).reset_index()
    data.columns = ['Day', 'Value']
    return data
```

Our function accepts a dataframe `df` and `metric`, one of the `Metric` objects from `metric_config.py`. As you can see, we pass `metric.func` to `grouped.apply` to make it general.

The line `data.columns = ['Day', 'Value']` resets the column names of the resultant dataframe to new ones.

Now that we have our time series, let's build our line chart.

USING PLOTLY TO BUILD A LINE CHART WITH OUR TIME SERIES

As we touched upon briefly in Chapter 1, Streamlit supports many different data visualization libraries. Plotly is one such library that tends to be fairly easy to use and offers a range of engaging interactive visualizations.

To use it, you first need to install it with `pip install plotly`.

The flavor of Plotly we'll use is called Plotly Graph Objects (GO for short). GO allows you to construct plots with a high degree of control and customization, making it ideal for interactive charts where you can define every detail.

Create a new Python module named `time_series_chart.py` with the code in listing 6.10.

Listing 6.10 `time_series_chart.py`

```
import plotly.graph_objs as go
from data_wrangling import get_metric_time_series

def get_time_series_chart(df, metric):
    data = get_metric_time_series(df, metric)
    fig = go.Figure()
    fig.add_trace(
        go.Scatter(x=data['Day'], y=data['Value'], mode='lines+markers')
    )

    fig.update_layout(
        title=f'{metric.title}',
        xaxis_title='Day',
        yaxis_title=metric.title
    )
    return fig
```

The import statement at the top makes Plotly Graph Objects available; the abbreviation `go` is conventionally used to refer to it.

The `get_time_series_chart` metric takes our dataframe as well as an object of type `Metric` and returns the line chart (a Plotly `Figure` object) that we can pass later to a Streamlit widget for display.

We first obtain our time series data by calling the `get_metric_time_series` function we defined earlier.

Plotly charts are constructed incrementally. We start with an empty chart and add the components we need bit-by-bit. Here, the line `fig = go.Figure()` initializes the chart and assigns it to `fig`.

The next part adds the actual line to the chart:

```
fig.add_trace(
    go.Scatter(x=data['Day'], y=data['Value'], mode='lines+markers')
)
```

We're creating a `go.Scatter` object here, which represents a scatterplot. A scatterplot simply plots points on a graph with two axes (the x-axis and the y-axis). Each point has a pair of coordinates. In this case, the x-coordinates are supplied by `data['Day']`, or the dates we want to show in the chart, and the y-coordinates are in `data['Value']`, which will be the values of one of our key metrics (depending on which one the variable `metric` contains).

We also pass `mode='lines+markers'`, which makes it so that in addition to plotting the points (using "markers"), Plotly also puts lines between them. The effect is that every (`Day`, `Value`) pair in our dataframe has a marker, and all the markers are connected by lines, forming the line chart we need.

We then add the plot we just created to our `Figure` object by passing it to `fig.add_trace()`.

```
fig.update_layout(
    title=f'{metric.title}',
    xaxis_title='Day',
    yaxis_title=metric.title
)
```

The last part, shown above, simply adds some text to our graph, such as a title (which we get from our `Metric` object's `title` attribute, defined in `metric_config.py`), and the titles of the x- and y-axes.

We now know how to create the chart we require. We still need to actually display it though, so go ahead and update `time_series_chart.py`, adding a `time_series_chart` function at the bottom and the corresponding imports at the top:

```

import plotly.graph_objs as go
import streamlit as st
from data_wrangling import get_metric_time_series
from metric_config import metrics, display_metrics

...
def time_series_chart(df):
    with st.container(border=True):
        chart_tabs = st.tabs(display_metrics)
        for idx, met in enumerate(display_metrics):
            with chart_tabs[idx]:
                chart = get_time_series_chart(df, metrics[met])
                st.plotly_chart(chart, use_container_width=True)

```

Once again, we use `st.container(border=True)` to make a box to put our line chart in.

The next bit, `chart_tabs = st.tabs(display_metrics)`, introduces a new Streamlit UI widget: `st.tabs`.

As the name suggests, `st.tabs` creates a tabbed area in your app that lets users switch between pieces of content by clicking tabs at the top. The argument to `st.tabs` is the list of tab titles to use.

For instance, `st.tabs(["Home", "About us", "Careers"])` would create three tabs with the titles "Home", "About us", and "Careers".

We're trying to create one tab for each metric with its respective line chart, so we can pass the variable `display_metrics` from `metric_config.py`—which, you may recall, is a list of the metrics we care about that we can use as tab titles.

We define the content within a tab in the same way that we would do it for `st.column`: using the `with` context manager. Since `chart_tabs` now contains the list of tabs (returned by `st.tabs`), we can iterate through each list index/metric name pair (`idx, met`) in `display_metrics` and use `chart_tabs[idx]` to refer to the corresponding tab, calling `get_time_series_chart` to create the Plotly line chart for that metric.

Finally, we pass the created chart to the `st.plotly_chart` element to render it on the screen:

```
st.plotly_chart(chart, use_container_width=True)
```

The `use_container_width=True` ensures that the line chart expands to fill the width of the box that contains it. This prevents weird layout issues where the chart ends up being larger than the container or leaves a lot of whitespace around it.

Let's now include the line chart in our main app by updating `dashboard.py`:

```

...
from time_series_chart import time_series_chart

...
if main_df.empty:
    st.warning("No data to display")
else:
    metric_bar(main_df)
    time_series_chart(main_df)

```

If you save and re-run the app now, you'll see your first Streamlit visualization (see figure 6.15)



Figure 6.15 Time series chart created using Plotly (chapter_6/in_progress_8 in the GitHub repo has the full code)

Pretty, ce n'est pas? You can see the tabs and switch between them to see how each metric has changed over the given data range. Visualizations created using `st.plotly_chart` throw in a lot of useful functionality for free, such as the ability to zoom in to specific points in the chart, tooltips when you hover over specific data points, a full-screen mode, and a download button to save the image.

NOTE We could have used radio buttons as our metric selection widget as our UI mock indicates, but I didn't want to pass up the chance to introduce st.tabs. Also, there is one important difference in how selection via st.radio and that via st.tabs works: switching between tabs does not trigger an app rerun while changing between radio button options does. This makes using tabs faster and more efficient with the tradeoff that charts for all the metrics need to be created at once, during the initial load.

6.5.2 Creating a pie chart

The time series chart we just created lets us identify trends over time, but we also need to be able to break a particular data point down and see what contributes to it. For instance, if we know that the total sales for the "Fountain pens" segment are \$500k, it would be helpful to know that 70% of that was driven by the RoyalQuill brand, while Inkstream only accounted for 30%, or that 45% of stapler sales are from the 56+ age group.

Pie charts, which illustrate the percentage breakdown of a whole into its component parts, are a good way to instantly form a picture of the data.

Figure 6.16 shows the pie chart from our UI mock again.

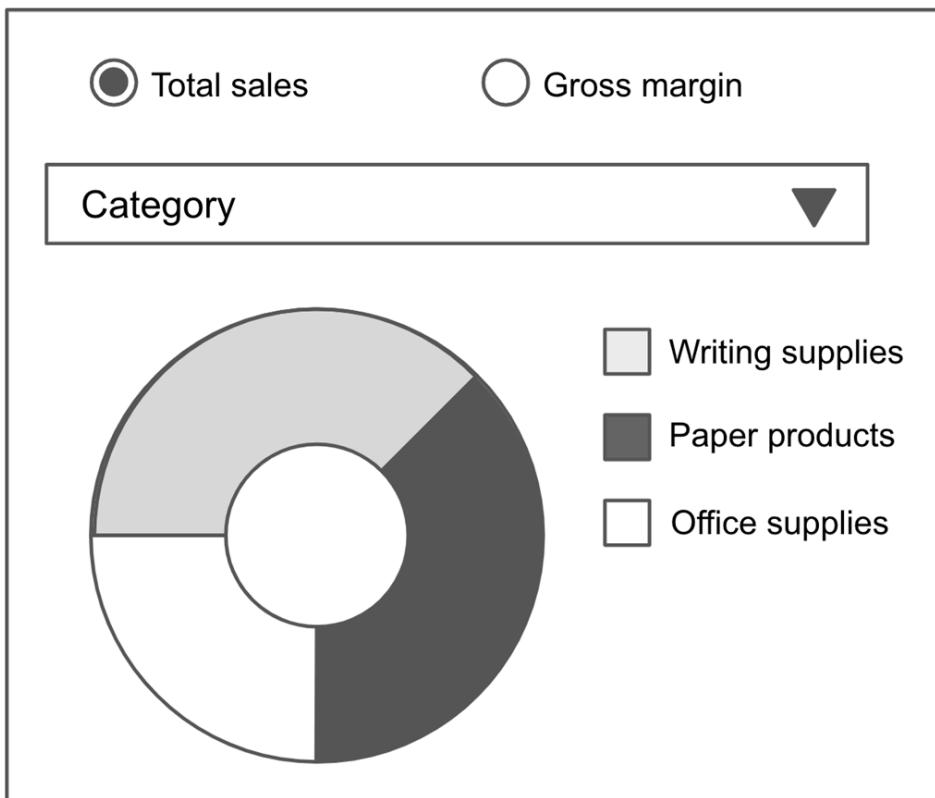


Figure 6.16 The pie chart from our UI mock

There are two selections you can make here: a metric for the pie chart to display, and a breakdown dimension. The metric can only be "Total sales" or "Gross margin", not "Margin %" or ATV. This is because the latter two metrics are ratios, and the corresponding dimension-wise ratios won't add up to 100%—so a pie chart does not apply.

For instance, let's say the average transaction value (sales divided by transaction count) for fountain pens is \$50. We can't break this down by gender and say that 60% (\$30) of that comes from males and 40% (\$20) comes from females. To get the ATV for males, we have to calculate the ratio by dividing total sales for males by the corresponding transaction count.

Let's record this smaller list of pie-chart-applicable metrics in a new variable at the bottom of metric_config.py:

```
pie_chart_display_metrics = ["Total sales", "Gross margin"]
```

GETTING DATA INTO THE RIGHT SHAPE

Just as we wrangled our data into a date/value time series to feed it to the time series chart, we need to prepare our data for the pie chart too. The pie chart needs to know the value of our metric corresponding to each dimension value; it will do the conversion to percentages on its own.

For example, consider the earlier sample data we processed into a time series earlier:

Day	State	Gender	Product	Sales
2024-08-01	CA	M	RoyalQuill	1500
2024-08-01	CA	M	GripLink	1300
2024-08-02	CA	M	RoyalQuill	1600
2024-08-02	CA	M	GripLink	1200

If we want to show a breakdown of the total sales by product, you would group by product and add up the sales:

Product	Sales
RoyalQuill	3100
GripLink	2500

This is remarkably similar to what we did earlier; the only difference is that instead of grouping by the date field, we're grouping by a particular dimension ("Product") instead.

The function we'll include at the bottom of `data_wrangling.py` is thus also very similar to `get_metric_time_series`:

```

...
def get_metric_time_series(df, metric):
    grouped = df.groupby('Day')
    data = grouped.apply(metric.func, include_groups=False).reset_index()
    data.columns = ['Day', 'Value']
    return data

def get_metric_grouped_by_dimension(df, metric, dimension):
    grouped = df.groupby(dimension)
    data = grouped.apply(metric.func, include_groups=False).reset_index()
    data.columns = [dimension, 'Value']
    return data

```

The sole distinction between the newly added `get_metric_grouped_by_dimension` and `get_metric_time_series` is that in the former, we're accepting dimension as an input and grouping by that instead of `Day`.

A PLOTLY PIE CHART

The ever-versatile Plotly Graph Objects can also be used to create the pie chart we want. In fact, the code you'll put in `pie_chart.py` (a new file shown in listing 6.11), is closely related to that in `time_series_chart.py`:

Listing 6.11 pie_chart.py

```

import plotly.graph_objects as go
from data_wrangling import get_metric_grouped_by_dimension

def get_pie_chart(df, metric, dimension):
    data = get_metric_grouped_by_dimension(df, metric, dimension)
    fig = go.Figure()
    fig.add_trace(
        go.Pie(labels=data[dimension], values=data['Value'], hole=0.4)
    )
    return fig

```

The differences should be fairly obvious: we use `get_metric_grouped_by_dimension` in the place of `get_metric_time_series`, and `go.Pie` instead of `go.Scatter`.

`go.Pie` accepts `labels`, which will be displayed in a color legend, `values`, and `hole`, which indicates how large the "donut hole" in the pie chart should be.

We're not using `fig.update_layout()` here to set any text in the chart, since the title will simply be the tab header (which we'll get to in a second), and there are no x- or y-axes.

As we did previously, we also need to write another function in `pie_chart.py` to render the image:

```
import plotly.graph_objects as go
import streamlit as st
from data_wrangling import get_metric_grouped_by_dimension
from metric_config import metrics, pie_chart_display_metrics

...
def pie_chart(df):
    with st.container(border=True):
        split_dimension = st.selectbox(
            "Group by",
            ["Age group", "Gender", "State", "Category",
             "Segment", "Product name"]
        )
        metric_tabs = st.tabs(pie_chart_display_metrics)
        for idx, met in enumerate(pie_chart_display_metrics):
            with metric_tabs[idx]:
                chart = get_pie_chart(df, metrics[met], split_dimension)
                st.plotly_chart(chart, use_container_width=True)
```

The `pie_chart` function, too, is similar to its counterpart—`time_series_chart`—from `time_series.chart.py`.

The key difference is the added `split_dimension` variable (the dimension to break down the metric for) that we need to collect from users using `st.selectbox`.

Everything else stays more or less analogous; we create a tab for each metric in `pie_chart_display_metrics` (that we defined in `metric_config.py`), iterate through those metrics, create the Plotly object with `get_pie_chart`, and display it using `st.plotly_chart`.

In `dashboard.py`, we want to display the line and pie charts side-by-side, so we use `st.columns`:

```

...
from pie_chart import pie_chart

...
if main_df.empty:
    st.warning("No data to display")
else:
    metric_bar(main_df)
    time_series_col, pie_chart_col = st.columns(2)
    with time_series_col:
        time_series_chart(main_df)
    with pie_chart_col:
        pie_chart(main_df)

```

With this, our dashboard's UI is complete! Re-run to see figure 6.17.

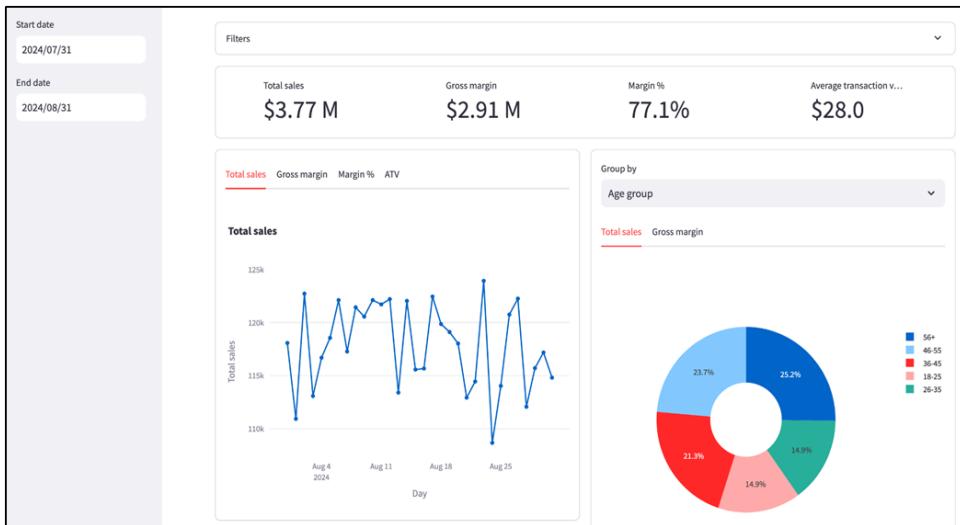


Figure 6.17 Our completed app, with the newly-added pie chart at the bottom right (chapter_6/in_progress_9 in the GitHub repo has the full code)

We've covered a lot of ground in this chapter, and we're ready to launch our dashboard. We're not quite done with Note n' Nib though. In the next chapter, we'll see several ways in which our app can be improved--including usability tweaks and shifting from a static CSV file to a data warehouse.

6.6 Summary

- A metrics dashboard is an essential decision-making tool for executives.

- Pandas is a popular Python library for manipulating tabular data in dataframes. Streamlit can display Pandas dataframes natively.
- Use Pandas' `read_csv` function to load data from a comma-separated values (CSV) file
- The `st.cache_data` decorator can be used to cache the results of functions to improve performance. Its `ttl` parameter sets the time period for which a cached result is valid.
- The dataframe square bracket notation is quite versatile in Pandas; you can use them to select columns, filter rows, and more.
- `st.container` can be used to hold other Streamlit widgets, displaying them out of order or with a border around them.
- `st.multiselect` creates a dropdown menu where you can select multiple options.
- `st.set_page_config` can set app configurations in Streamlit, including switching from a centered layout to a maximized one.
- `st.date_input` can be used to display date selectors in your app.
- The `humanize` library is useful for formatting numbers in a user-friendly way.
- A time series is a sequence of data points, each with a date and a value.
- The `groupby` method on Pandas dataframes can aggregate data across dimensions.
- Plotly Graph Objects (abbreviated to `go`) is a Python library used to create visualizations that can be displayed directly by Streamlit.
- `go.Scatter` can be used to create scatterplots and line charts, while `go.Pie` can make pie charts.
- A data warehouse is a specialized system designed to store and retrieve large amounts of data.
- Google BigQuery—part of GCP—is an example of a data warehouse. To enable an app to connect to it, you need to create a service account with a key, and record the credentials in `st.secrets`.

7 The CEO Strikes Back: Supercharging our dashboard

This chapter covers

- Critically evaluating an app and resolving user feedback
- Adding flexibility to Streamlit visualizations
- Improving usability by making commonly-used features easy to access
- Creating modal dialogs in Streamlit
- Using query parameters to enable deeplinks in a Streamlit app

When Python was first released, it lacked many of the features we rely on today. The Python we know and love has been carefully refined over the years—and that process is still ongoing. A big part of this evolution comes from the feedback of developers who actively build with the language.

Indeed, no software is perfect at launch. Instead, it is refined over time, with a bugfix here and a new feature added there. The projects we build in this book are no exception.

In the previous chapter, we created a metrics dashboard for a company called Note n' Nib. In this one, we'll skip forward and see how users responded. We'll use their opinions and comments to revisit our app in a critical light and improve it. As we go through this process, we'll learn more about Streamlit visualizations, introduce modal dialogs and query parameters, and understand how to program an advanced, flexible dashboard.

If Chapter 6 was about *launching* a dashboard based on user requirements, Chapter 7 is about *landing* it, addressing our users' concerns and iterating on the app to ensure quality and satisfaction.

7.1 Feedback on the dashboard

The dashboard you built in the last chapter has made waves at Note n' Nib. For the first time, the company's executives can look up updated sales numbers, compare performance between products, and analyze trends on their own, without having to enlist help from Engineering.

The CEO has requested that all staff meetings start with a review of the key sales metrics, which means that the top brass is now intimately familiar with your dashboard. Of course, with that kind of attention comes increased scrutiny, and you're not entirely surprised when, one Monday afternoon a few weeks after launch, you find an email from the CEO in your inbox.

The email contains collated feedback about your dashboard from the higher-ups—essentially a wishlist of additional features for you to implement.

Well, there goes the rest of your week. You're excited about the work, though, since you get to play with Streamlit some more! Over the course of this chapter, we'll inspect and resolve each item of feedback.

7.2 Granularity in the time series chart

The first bullet point in the email reads: "The time chart is useful in reviewing a few days of data, but it's hard to make sense of for longer periods".

Recall that our dashboard has a line chart showing a selected metric's evolution over time. For smaller date ranges (about a month or so), it works reasonably well (see the left side of figure 7.1), but for longer ones (say a year or more), it looks like the right side.

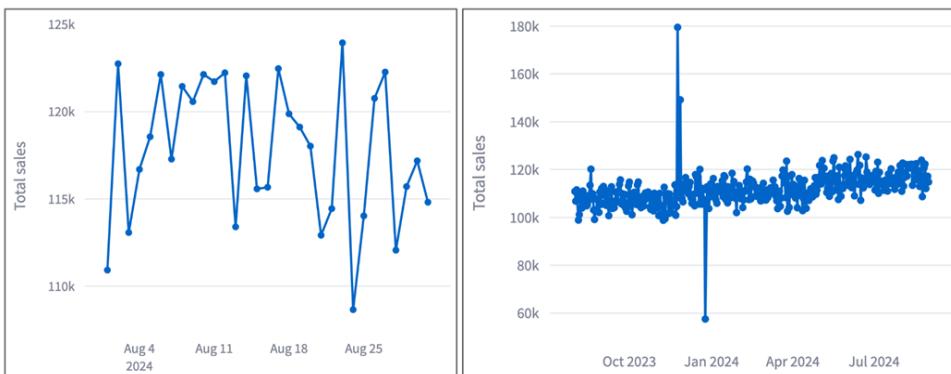


Figure 7.1 The day granularity in the existing time chart works for small time frames like a month (left) but not for longer time frames (right)

All the data is there, but there's so much of it that it's overwhelming. When considering years of data, we don't need to plot a single point for every date in the range.

You can see how doing so increases the number of individual markers in the chart; if we're observing a date range of two years, that's $365 \times 2 = 730$ points—far too many to interpret.

So, how do we address this? A granularity of a single day is too much over a longer time horizon, but if we have a shorter date range, like a week or a month, it's reasonable. For longer ranges, we would probably want one marker for a week, a month, or perhaps even a year.

The easiest solution would be to allow the user to pick the granularity they want. Let's tackle this now.

7.2.1 Enabling different time grains

To enable weekly, monthly, and yearly granularities, we first have to ensure our data *has* those fields, which it doesn't currently. Once we do, we'll be able to aggregate each metric to the right grain.

Recall that the flow of data in our app starts with the `load_data` function in `data_loader.py`, which obtains the data from an external source, currently a CSV file.

This is followed by `prep_data` in `data_wrangling.py` where we rename the columns and add the `Day` field. This is also where we need to make changes to incorporate the other grains we want.

Go ahead and edit `prep_data` so it now looks like this:

```
@st.cache_data(show_spinner="Reading sales data...", ttl="1d")
def prep_data() -> pd.DataFrame:
    df = clean_column_names(load_data())
    df['Day'] = pd.to_datetime(df['Date'])
    df['Week'] = df['Day'].dt.to_period('W').dt.to_timestamp()
    df['Month'] = df['Day'].dt.to_period('M').dt.to_timestamp()
    df['Year'] = df['Day'].dt.to_period('Y').dt.to_timestamp()
    return df
```

We're adding three new columns to the Pandas dataframe: `Week`, `Month` and `Year`. To obtain each field, we start with the `Day` column (`df['Day']`), convert it to a period, and then convert the result to a timestamp. Consider one such statement:

```
df['Month'] = df['Day'].dt.to_period('M').dt.to_timestamp()
```

`.dt` here is used to access the date/time-related properties of the column in an element-wise manner. `to_period('M')` converts the `Day` column into a monthly "period" type internal to Pandas, representing a whole month rather than a specific point in time.

We then use a second `.dt` accessor to get the date/time properties of the transformed column, and finally `.to_timestamp()` to convert each monthly period into a date representing the start of the month.

For instance, if one of the elements we're operating on in `df['Day']` is the date 2024-07-12, we end up with the date 2024-07-01, the start of the corresponding month.

The other two statements that create the `Week` and `Year` columns are analogous, adding dates representing the start of the week and the year respectively.

Elsewhere in the code—specifically within (other) functions in `data_wrangling.py` and `time_series_chart.py`—we've been treating `Day` as a hardcoded column name. Once we have these other columns, all we need to do in the backend is to introduce a variable to represent the grain instead.

So, the function `get_metric_time_series` from `data_wrangling.py` would now look like this:

```
def get_metric_time_series(df, metric, grain):
    grouped = df.groupby(grain)
    data = grouped.apply(metric.func, include_groups=False).reset_index()
    data.columns = [grain, 'Value']
    return data
```

And `get_time_series_chart` (in `time_series_chart.py`) becomes:

```
def get_time_series_chart(df, metric, grain):
    data = get_metric_time_series(df, metric, grain)
    fig = go.Figure()
    fig.add_trace(
        go.Scatter(x=data[grain], y=data['Value'], mode='lines+markers')
    )

    fig.update_layout(
        title=f'{metric.title}',
        xaxis_title=grain,
        yaxis_title=metric.title
    )
    return fig
```

In both cases, we're making the same changes: adding `grain` as a new parameter in the function, and replacing '`Day`' wherever it occurs to `grain`.

7.2.2 Creating a time grain selector

Now that we've wired up the functions that ultimately generate the time series chart to handle `grain` as a variable, we need to provide the user with a way to select what grain they want.

ST.SELECT_SLIDER

Let's use a new Streamlit widget for this: `st.select_slider`, another selection element. `st.select_slider` is a cross between `st.selectbox`, which lets you pick a single value from a dropdown, and `st.slider`, which lets you choose a numeric value.

You use it when you have a list of text options for users to pick from, but also want to impose some ordering on them. For instance, when you're creating options for a survey, "Strongly agree", "Agree", "Neutral", "Disagree", and "Strongly disagree" are strings, but they have a specific order to them—from the most to the least agreement.

In our case, the time grain options we want the user to see—"Day," "Week," "Month," and "Year"—also have an order, from the smallest unit of time to the largest.

For our purposes, we can use `st.select_slider` like this within the `time_series_chart` function in `time_series_chart.py`:

```
def time_series_chart(df):
    with st.container(border=True):
        grain_options = ["Day", "Week", "Month", "Year"]
        grain = st.select_slider("Time grain", grain_options)
        chart_tabs = st.tabs(display_metrics)
        for idx, met in enumerate(display_metrics):
            with chart_tabs[idx]:
                chart = get_time_series_chart(df, metrics[met], grain)
                st.plotly_chart(chart, use_container_width=True)
```

`grain_options` here holds the ordered list of options, and is fed to the second parameter of `st.select_slider`, the first being the label to display. You'll find these parameters very similar to those of `st.selectbox` and `st.radio`. `st.select_slider` returns the option selected by the user, which we store in `grain`, and pass as the new parameter we recently added to `get_time_series_chart`.

Save and run your app with `streamlit run <path to dashboard.py>` to get figure 7.2

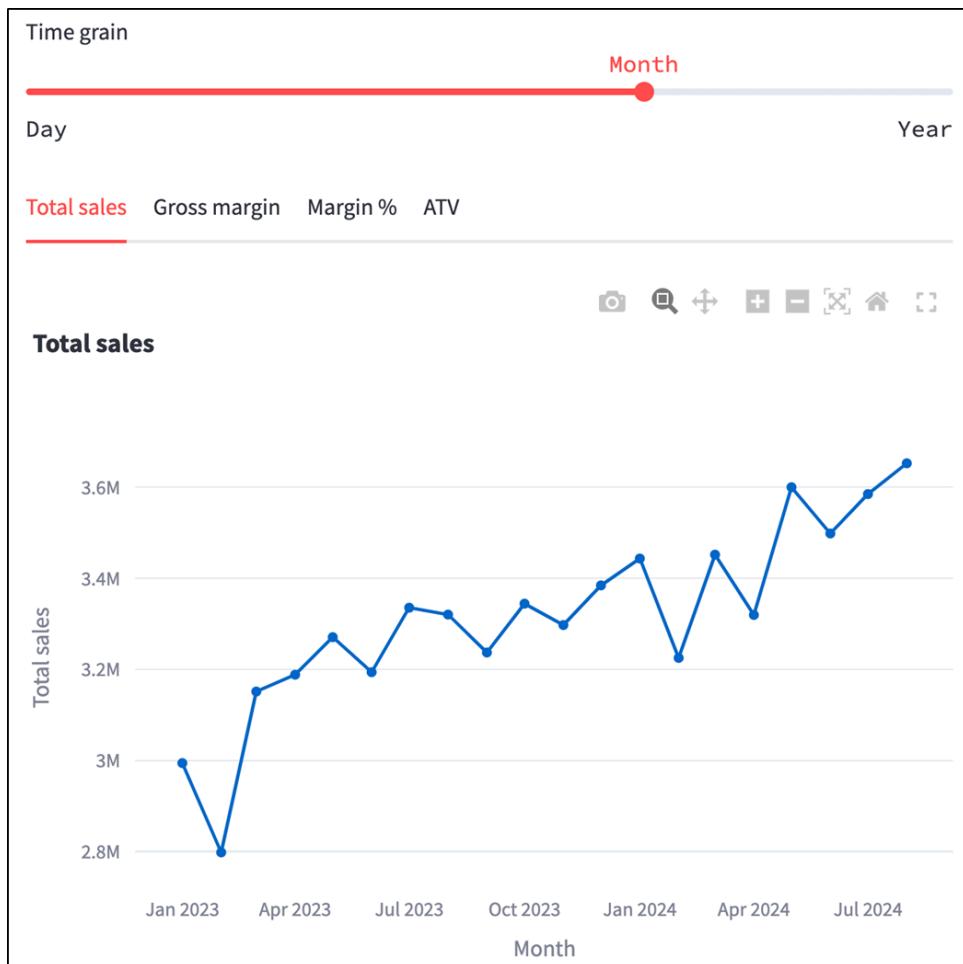


Figure 7.2 The line chart now has a time grain selector (see chapter_7/in_progress_01 in the GitHub repo for the full code at this point)

Play around with the grain selector. Using the month grain makes the chart much more palatable when viewing a long date range spanning multiple years.

7.3 Interdependent filters

"If I've already selected the 'Writing tools' category, why does it still ask me if I want to see staples and calendars?" asks a verbatim comment in the email, reportedly from the CFO, one of the dashboard's more passionate users.

You admit it's a valid question. She's referring to the filter bar in figure 7.3, which doesn't take the existing selections into account while displaying options in the filter dropdowns, leading to nonsensical combinations such as "Writing tools" paired with "Paper clips."

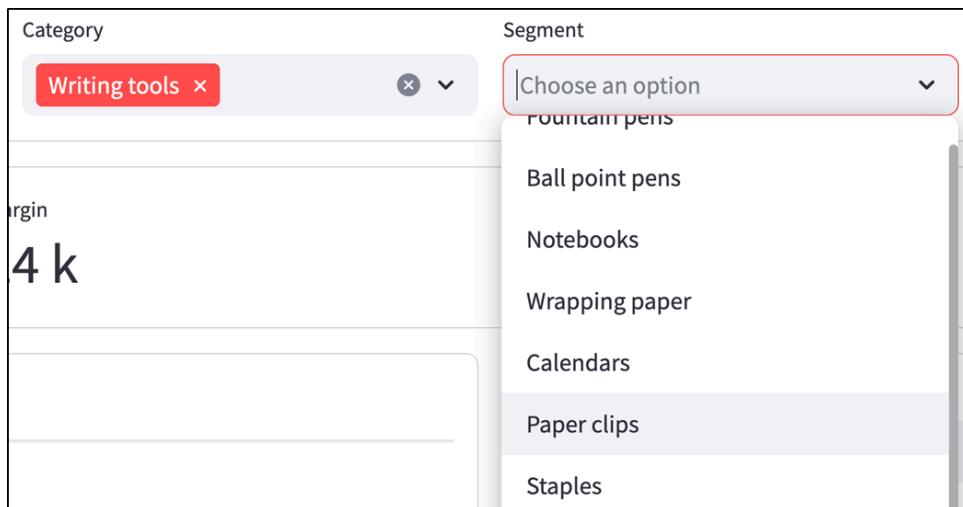


Figure 7.3 The filter bar options can have mismatching combinations

The filter bar isn't "intelligent". Filtering on a product category doesn't update the options available to the user in the segment and product name filters, even if those segments and products don't belong to the selected category.

If a user filters the data for "Writing tools", it's a bit annoying to still see all the other product lines (like "Paper clips") in the segment dropdown. The example data we've been using only has ten products overall, so it isn't a dealbreaker per se, but consider the situation where there are hundreds. At that point, using the higher levels in the product hierarchy (like "category") to filter out irrelevant products from the other dropdowns becomes a necessary feature, not a nice-to-have.

Let's consider how to fix this.

One possibility is to record the interdependencies between the dimensions, and then look up and resolve those dependencies while obtaining the unique values for each field. For example, since `Product name` should depend on the selections for `Category` and `Segment`, we might record that dependency somewhere.

This requires us to maintain a new configuration and the logic could get fairly involved.

There's an easier alternative: rather than obtaining all the unique filter values first and *then* filtering the dataframe (figure 7.4), we could get the unique values for the first filter, apply the filter to get a new dataframe, get the unique values for the second filter, apply that one too, and so on.

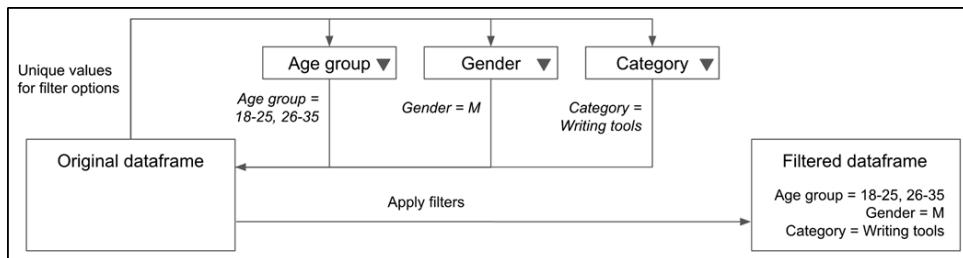


Figure 7.4 Old filtering approach: Get unique values for all filters first, then filter based on selections

In the new approach (figure 7.5), since we filter the data frame before getting the unique values for the next filter, we're guaranteed to show only the available values.

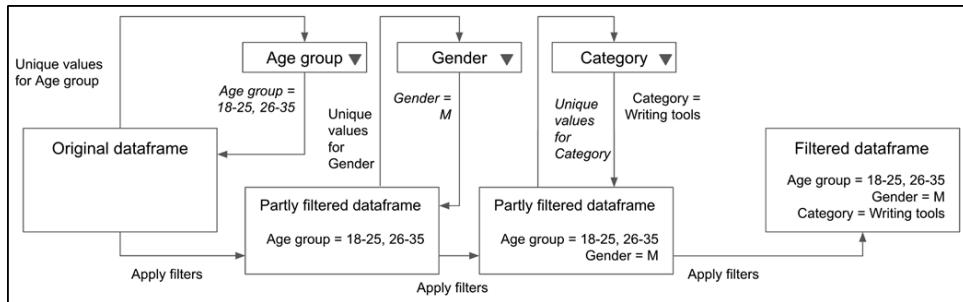


Figure 7.5 New filtering approach: Get unique values for each filter, apply filter based on selection, then repeat for other filters

So when a user selects the category "Writing tools" the dataframe is filtered to only include those rows, and the unique values for the segment filter are drawn from this new set, which won't include staplers and the like.

To implement this, modify `filter_panel.py`:

```

import streamlit as st
from data_wrangling import get_unique_values, apply_filters

filter_dims = ["Age group", "Gender", "Category", "Segment",
               "Product name", "State"]

def filter_panel(df):
    filters = {}
    with st.expander("Filters"):
        filter_cols = st.columns(len(filter_dims))
        effective_df = df
        for idx, dim in enumerate(filter_dims):
            with filter_cols[idx]:
                effective_df = apply_filters(effective_df, filters)
                unique_vals = get_unique_values(effective_df, dim)
                filters[dim] = st.multiselect(dim, unique_vals)
    return filters

```

The changes aren't too complicated. In each iteration through the filter fields, rather than passing `df` directly to `get_unique_values` to get the set of dropdown options to display, we introduce a variable called `effective_df` and pass that.

In line with our explanation of the approach, `effective_df` is recomputed in each loop iteration by applying the filters we have so far (we import `apply_filters` at the top for this purpose).

Go ahead and re-run your app! Figure 7.6 shows what happens when you select "Writing tools" as the only category you're interested in.

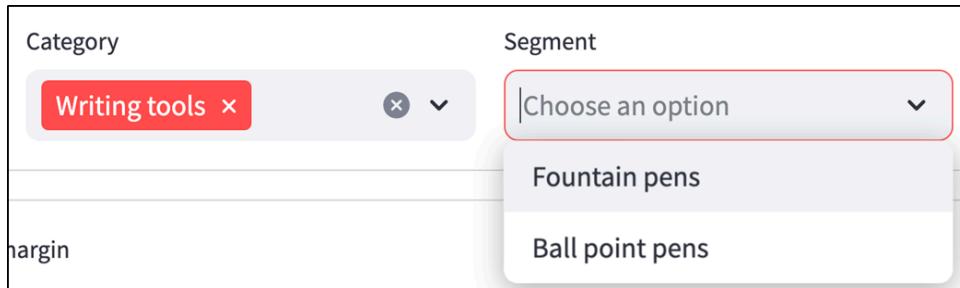


Figure 7.6 The filter bar only shows valid option combinations (see [chapter_7/in_progress_02](#) in the GitHub repo for the full code)

As expected, the `Segment` filter now only shows writing instruments.

NOTE The *order* of the filters is now meaningful. If the Category filter were to be placed *after* Segment, choosing a category would have no effect on Segment because the unique values for Segment would already have been computed by the time the selected Category values are evaluated.

7.4 Date range comparisons

Another piece of feedback comes from one of the product line chiefs, who posted a screenshot (figure 7.7) to illustrate her point: "I can see that sales for RoyalQuill were \$1.32M for July. But is that good or bad? How did we do last year?"

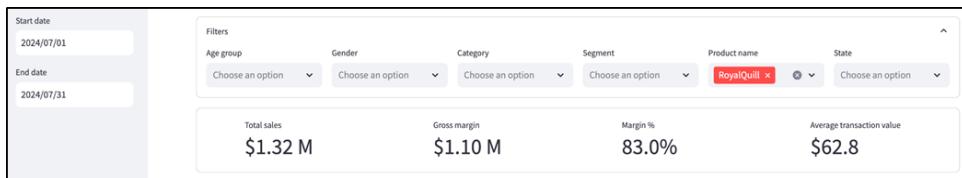


Figure 7.7 Sales for RoyalQuill were \$1.32M but there's no indication of whether that's good, or what the sales were in a previous comparable period

Often, the hard part about analyzing data is not obtaining or transforming it but *contextualizing* it. A metric by itself doesn't mean much. To make it useful, you have to be able to *compare* it to something. If we know sales for a product are \$1m in a year, the decisions we would make if we knew that last year's sales were \$10m are very different from those we'd make if last year's sales were only \$100k.

Our dashboard doesn't currently offer an easy way to make this comparison.

Ideally, when we see a metric for a certain time period, we should also be able to tell how it's *changed* as compared to the past. In this section, we'll explore this requirement more deeply and incorporate it into our dashboard.

7.4.1 Adding another date range selector

What exactly does it mean to compare a metric to its past value? What start date and end date do we use for "the past"? Let's consider some frequently-accessed comparisons that users may be interested in.

For instance, if the user is viewing the total sales for August 1 to August 15, 2024, there's a good chance that they might want to see how that compares to the same dates in the previous month, i.e. July 1 to July 15, 2024. This is called a "month-over-month" comparison, often abbreviated to "MoM".

Other similar comparisons the user might want to make are QoQ ("quarter-over-quarter") and YoY ("year-over-year"), which are both similar to MoM. QoQ means comparing to analogous dates in the previous quarter. For example, August 1 to August 15 represents the first 15 days of the second month of Q3, so QoQ would compare this to the first 15 days of the second month of Q2, or exactly three months earlier: May 1 to May 15.

YoY should be obvious—it's the date range that's exactly a year in the past, so August 1 to August 15, 2023 for our example.

Another type of comparison an executive might want to make is against the immediately prior X days, where X is the number of days in the currently selected main date range.

So, if the main range is August 1 to August 15, the "previous period" would be the 15 days immediately preceding August 1, i.e. July 17 to July 31.

Let's now implement these commonly-accessed comparisons. We start by updating our `date_range_panel` function in `date_range_panel.py` to incorporate a comparison selector and return two more dates to the caller (`dashboard.py`, which we'll edit later):

```
...
def date_range_panel():
    start = st.date_input("Start date", value=THIRTY_DAYS_AGO)
    end = st.date_input("End date", value=LATEST_DATE)
    comparison = st.selectbox(
        "Compare to", ["MoM", "QoQ", "YoY", "Previous period"])
    compare_start, compare_end = get_compare_range(start, end, comparison)
    st.info(f"Comparing with: \n{compare_start} - {compare_end}")
    return start, end, compare_start, compare_end
```

Since the comparison options are discrete values, we use an `st.selectbox` to offer users a choice between them, and call a yet-to-be-defined function, `get_compare_range`, to get the actual start and end dates of the comparison range.

We also expose these comparison dates to the user in an `st.info` box so the user doesn't have to do any calendar math themselves to get that information.

Let's also define the `get_compare_range` function we referenced above (in the same file, `date_range_panel.py`):

```
def get_compare_range(start, end, comparison):
    offsets = {
        "MoM": pd.DateOffset(months=1),
        "QoQ": pd.DateOffset(months=3),
        "YoY": pd.DateOffset(years=1),
        "Previous period": pd.DateOffset((end - start).days + 1)
    }
    offset = offsets[comparison]
    return (start - offset).date(), (end - offset).date()
```

This function accepts three parameters: the start and end dates of the main date range, and `comparison`, a string that holds the type of comparison we want to perform—as discussed above, this could be `MoM`, `QoQ`, `YoY`, or `Previous period`.

Calculating the comparison date range comes down to subtracting the right `offset` from both the start and end dates. For example, for an MoM comparison, we need to subtract one month from both dates. For QoQ, we subtract 3 months, and for YoY, we subtract a year.

For the previous period comparison, we first find the number of days within the main date range using `(end - start).days + 1`, and use that as the offset.

We store these offsets in a dictionary (called `offsets` in the code above) with the comparison name as the key and a Pandas `DateOffset` object as the value. We can then obtain the new start and end dates by subtracting the offset from each:

```
return (start - offset).date(), (end - offset).date()
```

NOTE Why are the `.date()`s necessary here? Well, if you've been paying particularly close attention, you might have realized that `start` and `end` are `datetime.date` objects, not Pandas timestamp objects. Pandas makes sure that `pd.DateOffset` is compatible with `datetime.date` and that the former can be subtracted from the latter, but the result is a Pandas timestamp object. Since we've been trying to keep our date ranges `datetime.date` objects elsewhere, we use the `.date()` method of the Pandas timestamp class to convert `start - offset` and `end - offset` to `datetime.dates`—thus ensuring consistency.

Since the function `date_range_panel` now returns four values (`start`, `end`, `compare_start`, and `compare_end`) instead of just two, we need to update the code that calls it to reflect this.

This code happens to be in `dashboard.py`, within the sidebar. Change it from:

```
with st.sidebar:
    start, end = date_range_panel()

to:

with st.sidebar:
    start, end, compare_start, compare_end = date_range_panel()
```

Your app's sidebar should now look like figure 7.8.

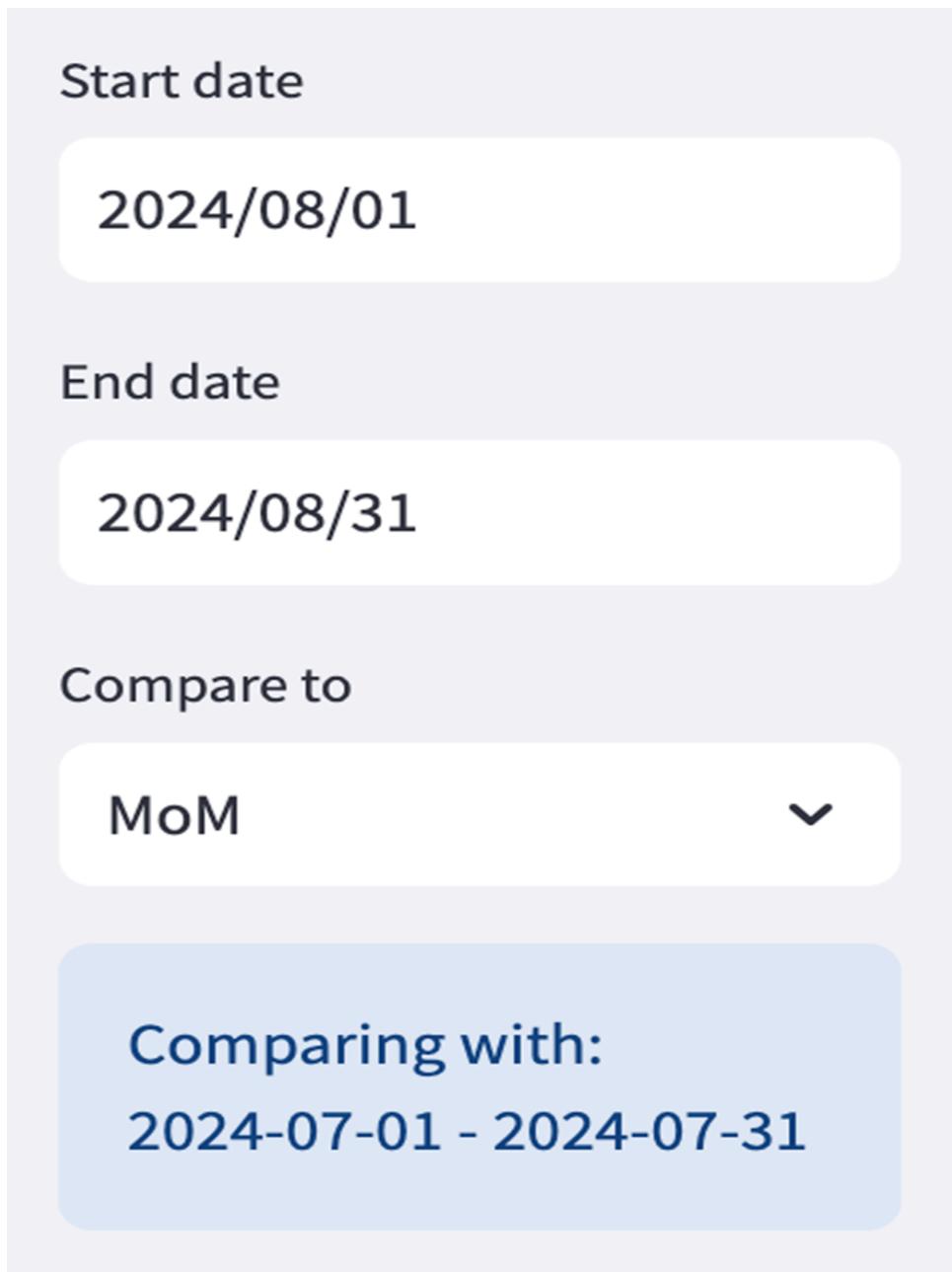


Figure 7.8 Sidebar showing a start/end date selector for the main date range as well as a "Compare to" input (see chapter_7/in_progress_03 in the GitHub repo for the full code)

Though we're not actually doing anything with the comparison date range, you can see the new selectors.

7.4.2 Showing the comparison in the metric bar

Now that we've collected the date range to compare to the main date range, how do we use it to resolve the feedback we received?

Let's consider this with an example. Say we have two date ranges: August 1 to August 31, 2024 (the "main" date range) and July 1 to July 31, 2024 (the comparison date range). If we're comparing the total sales between those ranges, we'll need to calculate them separately for both date ranges and then display the *delta* (difference) between them.

If the August sales are \$5m and those in July are \$4m, we would display a delta of \$1m. Generally speaking, expressing the difference as a percentage of the past number is more useful, so the delta is 20% ($\$4m / \$5m \times 100$). We would show this number alongside the August sales to provide a complete picture: Sales in August were \$5m, up 20% from the previous period.

This approach requires us to do two things:

- Calculate the metric separately for the comparison date frame, keeping everything else (mainly the filter values) constant
- Compute the percentage delta and display it with the main metric.

For the first part, let's modify `dashboard.py`:

```
import streamlit as st
...
with st.sidebar:
    start, end, compare_start, compare_end = date_range_panel()
...
main_df = get_filtered_data_within_date_range(data, start, end, filters)
if main_df.empty:
    st.warning("No data to display")
else:
    compare_df = get_filtered_data_within_date_range(
        data, compare_start, compare_end, filters)
    metric_bar(main_df, compare_df)
...
```

Here, we're obtaining a new Pandas dataframe, `compare_df`, in much the same way that we get `main_df`—by passing the raw prepped data to `get_filtered_data_within_date_range` with the appropriate start and end dates and filters.

The filters are the same as those used to create `main_df`. This is important because if a user has filtered for, say, a particular category and/or gender, the comparison they want to see is with the same category and/or gender, just for a different date range.

We also pass `compare_df` as a second argument to `metric_bar`, which it doesn't support yet, but will when we're done.

To compute and display the percentage delta, the changes we need to make are in `metric_bar.py`.

Let's start by modifying `metric_bar` to accept the extra argument we passed in:

```
def metric_bar(main_df, compare_df):
    with st.container(border=True):
        metric_cols = st.columns(len(display_metrics))
        for idx, metric_name in enumerate(display_metrics):
            metric = metrics[metric_name]
            with metric_cols[idx]:
                value = get_metric(main_df, metric)
                formatted_value = format_metric(value, metric.type)
                formatted_delta = get_formatted_delta(value, compare_df, metric)
                c1, c2, c3 = st.columns([1, 3, 1])
                with c2:
                    st.metric(
                        metric.title, formatted_value, formatted_delta, "normal")
```

Earlier, for each metric we needed to display, we would get the formatted value using `format_metric` and pass that, along with a title, to `st.metric` for display like this:

```
st.metric(metric.title, formatted_value)
```

However, `st.metric` supports showing a delta as well, through its third and fourth argument (internally named `delta` and `delta_color`).

The third argument is the formatted number to show as the change (in this case, the percentage difference), while the fourth argument, `delta_color`, indicates the color scheme to display the delta in.

`delta_color` can take the values "normal," "inverse," or "off." If it's set to "normal," positive deltas are displayed in green, and negative changes will be in red. If it's "inverse," the reverse is true: increases are in red, and decreases are green (this is appropriate for metrics where a lower value is better, like cost). If it's "off," Streamlit just shows everything in gray.

In this case, we're calling `st.metric` like this:

```
st.metric(metric.title, formatted_value, formatted_delta, "normal")
```

"normal" is appropriate for all of our metrics since a higher value is better for all of them (you'd want higher sales, a higher gross margin, a higher margin percentage, and a higher average transaction value). For the third argument, we pass in `formatted_delta`, which we're obtaining above by calling a function we haven't defined yet:

```
formatted_delta = get_formatted_delta(value, compare_df, metric)
```

Let's go ahead and create `get_formatted_delta`, and any associated functions now:

```
def get_delta(value, compare_df, metric):
    delta = None
    if compare_df is not None:
        compare_value = get_metric(compare_df, metric)
        if compare_value != 0:
            delta = (value - compare_value) / compare_value
    return delta

def get_formatted_delta(value, compare_df, metric):
    delta = get_delta(value, compare_df, metric)
    formatted_delta = None
    if delta is not None:
        formatted_delta = format_metric(delta, "percent")
    return formatted_delta
```

We've defined two functions: `get_delta` calculates the actual delta, while `get_formatted_delta` calls it and formats the result.

`get_delta` accepts the value of the main metric, `compare_df`—the comparison dataframe we computed in `dashboard.py`—and `metric`, which is the `Metric` object that represents the measure we're trying to show the change in.

The body of `get_delta` isn't complicated. We use the `get_metric` function on `compare_df` to calculate the metric for the comparison date range, and obtain the percentage delta as follows:

```
delta = (value - compare_value) / compare_value
```

At any point, if we realize that a delta can't be displayed (either because `compare_df` contains no data or because trying to calculate it would cause a divide-by-zero error since the comparison value is zero), we return `None` instead.

In `get_formatted_delta`, we take this returned value and get a formatted version of it by calling `format_metric`:

```
formatted_delta = format_metric(delta, "percent")
```

Recall from Chapter 6 that `format_metric` (defined in the file `formatting.py`) converts a numeric value into a user-friendly string depending on its type. In this case, the metric type is a "percent," so `format_metric` will add a "%" sign at the end.

If there's no `delta` to format (which happens when `get_delta` returns `None`), `get_formatted_delta` returns `None` as well.

When this is eventually passed to `st.metric`, Streamlit handles the `None` value correctly by not displaying anything at all.

You can now re-run the dashboard to view your updated metric bar (remember to choose a comparison date range that we have data for), as shown in figure 7.9.

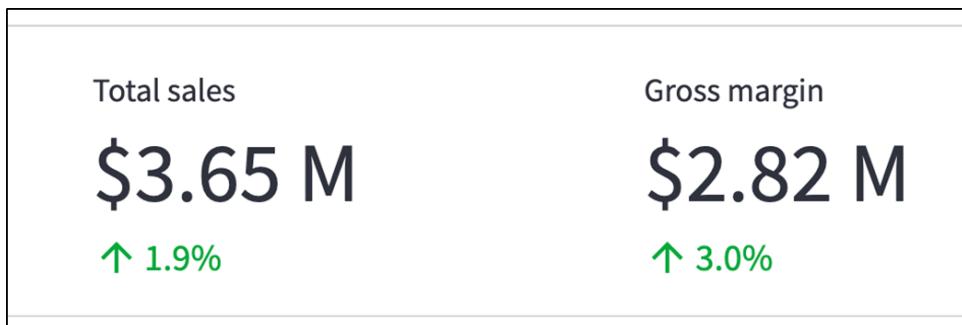


Figure 7.9 Metric bar showing how each metric has changed from the comparison date range (see chapter_7/in_progress_04 in the GitHub repo for the full code)

As you can see, the metric bar now shows how each metric has changed as compared to its value in the comparison date range.

Boom! We've addressed another key piece of feedback and are well on our way to version 2.0 of our dashboard! Let's see what else the email has to say.

7.5 A drilldown view

Note n' Nib's CEO prides himself on being a "details guy," so when he sees a number on the dashboard, he wants to investigate *why*. For instance, if he finds that the average transaction value on ball pens is lower than that on fountain pens, he wants to dig deeper into the data to understand if there's a certain demographic driving the ATV down.

Our dashboard doesn't expose data beyond what's in the metrics bar, line and pie charts, but there's clearly a desire for a more flexible and detailed view, perhaps even showing the individual rows in the source data.

So far in our dashboard design, we've tried to shield users from complexity where possible. We've relied on visualizations to make data easy to grasp and used a clear, friendly metric bar to display key aggregate numbers. The quality of abstracting away complexity is a laudable one most of the time and for most users. Now and then, however, you'll come across a power user who wants to go deeper and interact with your software in more advanced ways.

In our case, Note n' Nib's CEO fits this description—he's comfortable with data and has expressed frustration at not being able to drill down for more detailed insights. Addressing this feedback will likely be the most complex task of this chapter, as we'll need to create a whole new view rather than just improve upon the existing features.

7.5.1 Inserting a modal dialog

Before considering what a drilldown view might contain, let's ponder where to place this functionality. Since we're classifying it as an "advanced" feature, we probably shouldn't place it in the main window of the dashboard. Casual users should be able to ignore the new, more detailed view, while advanced users should be able to find it with no trouble.

Let's use this opportunity to discuss a new UI construct: a modal dialog. A modal dialog is essentially an overlay displayed on top of the main content, temporarily blocking interaction with the underlying interface until it has been dismissed. This overlay would remain focused on a particular task, which makes it ideal for presenting advanced functionality like drilldowns.

ST.DIALOG

Streamlit offers modal dialogs out of the box with `st.dialog`. Let's see this in action now.

For our first iteration on the drilldown view, to keep things simple, when a user wants to drill down into the data, we'll just show them the entire Pandas dataframe. Of course, since users are likely to want to also dive into the comparison date range we recently added, we'll need to show both the main and the comparison dataframes.

`Listing 7.1 drilldown.py` shows a new file, `drilldown.py`, set up to achieve this.

```
Listing 7.1 drilldown.py

import streamlit as st

@st.dialog("Drilldown", width="large")
def drilldown(main_df, compare_df):
    main_tab, compare_tab = st.tabs(["Main", "Compare"])
    with main_tab:
        st.dataframe(main_df, use_container_width=True)
    with compare_tab:
        st.dataframe(compare_df, use_container_width=True)
```

It may surprise you to learn that `st.dialog` is *not* structured the same way as `st.columns`, `st.tabs`, or `st.container`, i.e., as a widget that holds other widgets within.

Instead, it's similar to `st.cache_data` from chapter 6 in that it's a *decorator*. A function decorated with `st.dialog` runs and has its content rendered inside a popup dialog.

```
@st.dialog("Drilldown", width="large")
```

The width parameter simply sets the size of the dialog, which may be "small" (500 pixels wide) or "large" (750 pixels wide).

The function being decorated is called `drilldown`, and it accepts `main_df` and `compare_df` from `dashboard.py` as arguments. The function renders two tabs titled "Main" and "Compare", and uses a new widget, `st.dataframe`, to display the passed Pandas dataframes in their respective tabs.

Using `st.dataframe` like this simply shows the dataframe on the screen, just as `st.write` did in Chapter 6. We'll encounter it later too.

To see the dialog, we need to *trigger* it, so let's focus on that next.

USING ST.CONTAINER TO DISPLAY UI ELEMENTS OUT OF ORDER

As alluded to earlier, the `drilldown` view should be unobtrusive to the casual user, but fairly obvious for a power user to access. One way to achieve this would be to add a button labeled "Drilldown" to the sidebar, and have it trigger the dialog when clicked.

Let's examine the existing code in `dashboard.py`:

```
...
with st.sidebar:
    start, end, compare_start, compare_end = date_range_panel()

...
main_df = get_filtered_data_within_date_range(data, start, end, filters)
if main_df.empty:
    st.warning("No data to display")
else:
    compare_df = get_filtered_data_within_date_range(
        data, compare_start, compare_end, filters)
    ...

```

The sidebar already has the date range panel in it, with four widgets (two date inputs for the main range, a comparison selectbox, and an info box showing the comparison range), all lined up vertically. Since we want the `drilldown` trigger to be easily visible, we probably don't want it to be *below* the date range panel. Cool, so we put it above the panel instead, right?

Except, there's a bit of an ordering issue here. To trigger the drilldown view, we need to call the `drilldown` function we just decorated with `st.dialog`. The parameters to this function are `main_df` and `compare_df`.

If you inspect the `dashboard.py` code again, you'll realize that obtaining `main_df` and `compare_df` requires that we already *have* the `start`, `end`, `compare_start` and `compare_end` values so we can pass them in like this (for `main_df`):

```
main_df = get_filtered_data_within_date_range(data, start, end, filters)
```

But where do we get these values? Why, in the sidebar!

```
with st.sidebar:
    start, end, compare_start, compare_end = date_range_panel()
```

Do you see our dilemma? To place the drilldown button above the date range panel, we'd need to write its code *before* this line, but that code requires values that are only available *after* this line!

This is the perfect place to elaborate on something I mentioned in passing in Chapter 6: the ability to display elements out of order. We need a way to separate the order in which Streamlit renders widgets on the screen from the order in which those widgets are computed.

We'll use `st.container` for this. In chapter 6, we used it to display a border around the metric bar and visualizations. This time, we'll capitalize on a different property—`st.container` can put a *placeholder* widget on the screen that we can populate with other widgets when we can.

For our use case, the placeholder will be above the date range panel—within the sidebar—and we'll only populate it with the actual drilldown button once we have `main_df` and `compare_df` later in the code.

Let's lay this out in `dashboard.py`:

```

...
from drilldown import drilldown

...
with st.sidebar:
    dd_button_container = st.container()      #A
    start, end, compare_start, compare_end = date_range_panel()

...
main_df = get_filtered_data_within_date_range(data, start, end, filters)
if main_df.empty:
    st.warning("No data to display")
else:
    compare_df = get_filtered_data_within_date_range(
        data, compare_start, compare_end, filters)
    if dd_button_container.button("Drilldown", use_container_width=True):
        drilldown(main_df, compare_df)
...
#A This is the placeholder defined within st.sidebar

```

As promised, we use `st.container` above the date range panel to put a placeholder and refer to it by the name `dd_button_container`.

Then, once we have `main_df` and `compare_df`, we create the button that calls the `drilldown` function when clicked. Notice that we're using the syntax `dd_button_container.button` instead of the `with/st.button` structure, just as we could with columns or tabs.

It's finally time to see our dialog come to life! Re-run `dashboard.py` and click the `drilldown` button to get figure 7.10.

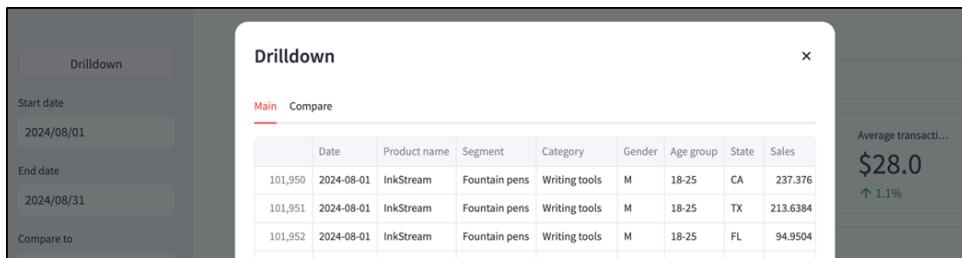


Figure 7.10 A basic raw dataframe view rendered in a dialog using `st.dialog` (see `chapter_7/in_progress_05` in the GitHub repo for the full code)

You may have noticed that the drilldown button took a second to appear before you clicked it. As you probably guessed, that's because we delayed its rendering until a bunch of other stuff was processed.

7.5.2 Designing the content of the drilldown

Turn your attention to the dialog in figure 7.10 for a minute. We're currently just displaying `main_df` and `compare_df` as-is. It's rather ugly, with the horizontal *and* vertical scrollbars indicating that we only see a tiny portion of the data. More to the point, we can't easily use this view to look for specific data points or view a specific subset of the data. The boss would not be pleased if he had to use this.

No, we need to think carefully to get the experience right.

WHAT DO USERS NEED FROM A DRILLDOWN VIEW?

It's clear that the content of the drilldown page needs to change, but how? What do our users need from this view? Well, what better way to understand than to talk to a user?

So, you book a slot on the CEO's calendar—it's a sign of his enthusiasm for the dashboard that he readily accepts. In your interview with him, he lays out the original motivation for the feedback he provided about the drilldown view.

Note n' Nib has two separate products in its best-selling line of fountain pens: Inkstream and RoyalQuill. InkStream is supposed to be a chic modern take on the fountain pen, while RoyalQuill, reminiscent of the classic elegance of vintage pens, is targeted at older customers.

Recently, the company ran an advertising campaign for RoyalQuill, specifically targeting women in the 46-55 and 56+ age groups. The CEO wanted some data on the assumptions and results of this campaign. Specifically, he wanted to know how sales broke down for InkStream and RoyalQuill by age *and* gender.

Our current dashboard shows users a breakdown of a metric by age *or* gender but not by both, so he can't easily access this information.

You may see how this feedback can be generalized to *any* combination of dimensions in the data, not just age group/gender. Also, it would be tricky to present this kind of detail in a coherent repeatable visualization.

What we need, then, is a highly flexible tabular form to show the data, similar to pivot tables that you may be familiar with from spreadsheet programs like Microsoft Excel. This table should:

- enable us to view numbers for any combination of dimensions we choose
- allow us to focus on just the fields we care about, hiding irrelevant rows and/or columns
- show aggregate numbers so we can see a full breakdown

A mock UI created with these requirements in mind is shown in figure 7.11.

Drilldown fields

Age group, Gender, Category						
Main		Compare				
Age group	Gender	Category	Total sales	Gross margin	Margin %	ATV
Total			\$1.7M	\$1.3M	79%	\$36
18-25	F	Writing tools	—	—	—	—
26-35	F	Writing tools	—	—	—	—
36-45	F	Writing tools	—	—	—	—
...

Figure 7.11 A mock UI for the drilldown view

The mock shows us a fairly flexible table, that's quite similar to a pivot table. The "Drilldown fields" box is a multi-select that lets us choose the dimensions we care about. Below that is a table that only shows the dimensions we've selected. It *aggregates* the data over those dimensions, showing the metrics for every combination of the selected dimensions. There's also a total row that adds up everything.

We've also retained the main/comparison tabs so that users can switch between them to see a past/present view of the metrics. This format is quite flexible, and meets our requirements, so it's time to build it out!

7.5.3 Implementing the drilldown view

The drilldown view in figure 7.11 is fairly complex, so we'll assemble it piece by piece, starting with the drilldown field selector, and ending with some formatting and styling.

AGGREGATING BY A FEW CHOSEN DIMENSIONS

Building the dimension selector (the top widget in figure 7.11) is just a matter of passing the possible dimension options in an `st.multiselect`. Add a new function to `drilldown.py` that does this and returns the list of the user's selections:

```
def drilldown_dimensions():
    return st.multiselect(
        "Drilldown fields",
        ["Age group", "Gender", "Category", "Segment", "Product name", "State"]
    )
```

As our mock in figure 7.11 shows, we want to display all key metrics simultaneously. Let's put another function (also in `drilldown.py`) that takes in a dataframe (or a slice thereof), calculates all the metrics by aggregating it, and returns the results. Remember to import all the modules we need!

```
import pandas as pd
import streamlit as st
from metric_config import metrics

...
def get_metric_cols(df):
    metrics_dict = {met: metric.func(df) for met, metric in metrics.items()}
    return pd.Series(metrics_dict)

...
```

The expression `{met: metric.func(df) for met, metric in metrics.items()}` is a *dictionary comprehension*, which is shorthand to create a dictionary by iterating through something. Here it's saying "iterate through the metrics dictionary (from `metric_config.py`) and return a new dictionary where each key is the name of the metric, and the corresponding value is the result of applying the metric function `metric.func` on `df`, i.e. the value of that metric".

We use a Pandas series here because, as you'll see shortly, it's a versatile data type that integrates seamlessly with various dataframe operations.

To prepare the aggregated table given a dataframe and a list of dimensions, we introduce a new function called `get_aggregate_metrics` in the same file:

```
def get_aggregate_metrics(df, dimensions):
    if dimensions:
        grouped = df.groupby(dimensions)
        return grouped.apply(get_metric_cols).reset_index()
    metric_cols = get_metric_cols(df)
    return pd.DataFrame(metric_cols).T
```

If `dimensions` is not an empty list, i.e. if the user has indeed selected some drilldown dimensions, `get_aggregate_metrics` groups `df` by those fields, and applies `get_metric_cols` to each of the groups (using `grouped.apply`, which you should be familiar with from Chapter 6), thus obtaining metric values for each group.

If no dimensions are selected, then we call `get_metric_cols` directly on `df` to get a Pandas series object with the aggregated metrics for the entire dataframe. Finally, we convert this series into a dataframe and return its *transpose*:

```
return pd.DataFrame(metric_cols).T
```

The transpose (referenced using a Pandas dataframe's `.T` property) of a dataframe is another dataframe whose rows and columns are interchanged. In this case, `metric_cols` is a `pd.Series` object, and calling `pd.DataFrame` on it would return a one-column dataframe where each of the metrics is a row.

The `.T` is required to turn this into a one-row dataframe where each metric is a column, a format that's more handy.

Next, we write a function that returns our full drilldown table. For now it's pretty thin since we're only doing some aggregations:

```
def get_drilldown_table(df, dimensions):
    aggregated = get_aggregate_metrics(df, dimensions)
    return aggregated
```

We'll add more logic to `get_drilldown_table` later.

To wrap up, we also need a function that displays the drilldown table (which is currently a Pandas dataframe):

```
def display_drilldown_table(df):
    if df is None:
        st.warning("No data available for selected filters and date range")
    else:
        st.dataframe(df, use_container_width=True, hide_index=True)
```

This should be simple; we display a warning if there's no data, or use `st.dataframe` to display the aggregated table otherwise. Note the use of `hide_index=True`. By default, Streamlit displays the index field (which, as you may recall from Chapter 6, is a unique identifier for the row, defaulting to a simple serial number) alongside each row. You can see this in figure 7.10 (they're the numbers to the extreme left in the dataframe). We don't want the index displayed to the user, so we hide it.

With these changes made, we can update the `drilldown` function too:

```

@st.dialog("Drilldown", width="large")
def drilldown(main_df, compare_df):
    dimensions = drilldown_dimensions()
    main_data = get_drilldown_table(main_df, dimensions)
    compare_data = get_drilldown_table(compare_df, dimensions)

    main_tab, compare_tab = st.tabs(["Main", "Compare"])
    with main_tab:
        display_drilldown_table(main_data)
    with compare_tab:
        display_drilldown_table(compare_data)

```

The order of operations is logical: First we get the drilldown dimensions from the user (`dimensions = drilldown_dimensions()`), then we compute the aggregated dataframes (for both `main_df` and `compare_df`) using `get_drilldown_table`, and eventually display them in separate tabs using `display_drilldown_table`.

If you re-run the dashboard now, you should see a much more palatable version of the drilldown view, as shown in figure 7.12.

Gender	Category	Total sales	Gross margin	Margin %	ATV
F	Office supplies	45,492.9304	35,777.5738	0.7864	10.1253
F	Paper products	310,673.1373	222,725.9512	0.7169	18.2181
F	Writing tools	1,696,765.754	1,333,765.6899	0.7861	36.4449
M	Office supplies	50,145.1769	39,111.8811	0.78	9.9712
M	Paper products	358,841.4117	260,572.6291	0.7261	18.2682
M	Writing tools	1,190,424.4791	923,964.7547	0.7762	31.7091

Figure 7.12 Drilldown view with a dimension selector and aggregated for selected dimensions (see [chapter_7/in_progress_06](#) in the GitHub repo for the full code)

The user can now pick whichever dimensions they want, and see the metrics for every combination of those dimensions. This effectively lets the user drill into the required level of detail in the data, but perhaps a summary "total" row would be warranted to understand the whole that's being broken down here.

ADDING A "TOTAL" ROW

Adding a summary row to the drilldown table is relatively complicated for a couple of reasons:

- Pandas dataframes do not natively have a way of designating a row as a summary of all the other rows. When we want totals, we have to wrangle them together using various operations.
- The dimension values are meaningless in a total row and should be blank.

To take an example, let's say we have the following drilldown dataframe (after the filtering and aggregation):

Gender	Segment	Product name	Total sales	Gross margin	Margin %	ATV
M	Fountain pens	InkStream	\$100,000	\$60,000	60%	\$10
M	Fountain pens	RoyalQuill	\$200,000	\$150,000	75%	\$40

With the total row added to the top, we would have a dataframe that looks like this:

Gender	Segment	Product name	Total sales	Gross margin	Margin %	ATV
Total			\$300,000	\$210,000	70%	\$20
M	Fountain pens	InkStream	\$100,000	\$60,000	60%	\$10
M	Fountain pens	RoyalQuill	\$200,000	\$150,000	75%	\$40

Let's implement this with a new function, `add_total_row`, in `drilldown.py`:

```
def add_total_row(df, all_df, dimensions):
    total_metrics = get_metric_cols(all_df)
    if dimensions:
        dim_vals = {dim: '' for dim in dimensions}
        dim_vals[dimensions[0]] = 'Total'
        total_row = pd.DataFrame({**dim_vals, **total_metrics}, index=[0])
        return pd.concat([total_row, df], ignore_index=True)
    total_row = pd.DataFrame({'': 'Total', **total_metrics}, index=[0])
    return total_row
```

`add_total_rows` takes three arguments: `df`, `all_df`, and `dimensions` (the same list of dimension names we've been passing around). `df` is the drilldown dataframe we have so far (e.g. the first table above), while `all_df` is the dataframe with the original granular columns *before* aggregation.

Why do we need both `df` and `all_df` here? Recall that we have a `get_metric_cols` function that can calculate all the metrics we need for a given dataframe—in other words, the numeric values for the "total" row we're trying to build. `get_metric_cols` expects a raw non-aggregated dataframe, not the aggregated version. This means we need to pass it `all_df`, not `df`.

That is indeed what the first statement in the function does, storing the results in `total_metrics`.

The next part builds the total row if the dimensions list is non-empty (`if dimensions:`), i.e. if the user has selected some drilldown dimensions.

The following two lines are associated with populating the dimension values:

```
dim_vals = {dim: '' for dim in dimensions}
dim_vals[dimensions[0]] = 'Total'
```

The first line is another dictionary comprehension that has a blank value for every dimension key from `dimensions`. We then set the value for the first dimension to "Total". This effectively creates the text display values for our total row as we saw in our example above—"Total" in the first field and blanks for everything else.

We have the dimension values for the total row in `dim_vals` (a dictionary) and the metric values in `total_metrics` (a `pd.Series`). All we need to do is to put them together! That's what the next line does:

```
total_row = pd.DataFrame({**dim_vals, **total_metrics}, index=[0])
```

There's some interesting syntax here, so let's break it down.

The character sequence `**` here is called a *dictionary unpacking operator*. It unpacks the items from a dictionary so they can be combined with other items to form a new dictionary, or even passed as function arguments.

The former is what's happening here. For instance, if `dim_vals` is something like `{'Gender': 'Total', 'Segment': '', ...}` and `total_metrics` is `{'Total sales': 300000, ...}`, `{**dim_vals, **total_metrics}` gives you a *combined* dictionary `{'Gender': 'Total', 'Segment': '', ..., 'Total sales': 300000, ...}`. The `index=[0]` sets the index of the only row in this single-row dataframe to 0.

You might notice one issue with this though: Didn't we just say that `total_metrics` is a `pd.Series` and *not* a dictionary? Well, though that's true, a Pandas series actually has many of the properties of a regular Python dictionary—among them support for the `**` operator.

The next line concatenates this total row to the rest of the drilldown dataframe and returns it:

```
return pd.concat([total_row, df], ignore_index=True)
```

Now, if the user has *not* selected any dimensions and `dimensions` is empty, getting the dataframe with a total row becomes easier; we just need to add a blank column that says "Total" to `total_metrics`, and there's nothing to concatenate the row to—the dataframe consists only the total row:

```
total_row = pd.DataFrame({'': 'Total', **total_metrics}, index=[0])
```

We can now add the act of obtaining the total row to the transformations in `get_drilldown_table` like so:

```
def get_drilldown_table(df, dimensions):
    aggregated = get_aggregate_metrics(df, dimensions)
    with_total = add_total_row(aggregated, df, dimensions)
    return with_total
```

Re-run the dashboard to see what your total row looks like (see figure 7.13):

The screenshot shows a dashboard interface with a title 'Drilldown'. Below the title is a section labeled 'Drilldown fields' with two buttons: 'Category' and 'Age group', both of which have a red 'X' icon. Underneath this is a navigation bar with 'Main' and 'Compare' tabs, where 'Main' is underlined. The main content is a table with the following data:

Category	Age group	Total sales	Gross margin	Margin %	ATV
Total		3,652,342.8895	2,815,918.4796	0.771	28.0266
Office supplies	18-25	14,578.0991	11,505.5842	0.7892	9.7317
Office supplies	26-35	12,638.6563	9,988.5907	0.7903	9.6626
Office supplies	36-45	13,335.8745	10,443.9369	0.7831	10.1336

Figure 7.13 Drilldown dataframe view with a total row (see chapter_7/in_progress_07 in the GitHub repo for the full code)

This is *almost* perfect, but wouldn't it be nice if the total row were highlighted or shaded to set it apart from the other rows?

FORMATTING AND STYLING THE DRILLODOWN TABLE

While we have the content of our drilldown table ready to go, the presentation leaves a couple of things to be desired:

- As figure 7.17 shows, the numbers in the table are user-unfriendly raw ones, with hardly any formatting. Ideally, we'd want these to be shown in the same way as in the metric bar (e.g. "\$1.2m" instead of "1200000").
- No shading distinguishes the total row from the rest of the table.

Let's tackle the former first. Formatting the numbers in the table should be quite straightforward because we've already defined the actual formatting rules in the `formatting.py` file from Chapter 6.

All we need is a function to apply the formatting to an entire Pandas dataframe rather than the individual numbers displayed in the metric bar.

Spin up a new function for this in `formatting.py`:

```
import humanize

def format_metric(value, metric_type):
    ...

def format_dataframe(df, metrics):
    cols = df.columns
    for col in cols:
        if col in metrics:
            df[col] = df[col].apply(format_metric, metric_type=metrics[col].type)
    return df
```

The `format_dataframe` function should be simple to wrap your head around. After accepting two parameters (`df`, the dataframe to format, and the `metrics` dictionary from `metrics.py`), we simply iterate through the columns in `df`, and apply `format_metric` (a function we wrote in chapter 6) element-wise to each column.

Notice how we're passing `metric_type` to `format_metric` as *another* parameter to `.apply()`!

Essentially the following expression:

```
df[col] = df[col].apply(format_metric, metric_type=metrics[col].type)
```

is saying: "issue the function call `format_metric(element, metric_type=metrics[col].type)`" for every element in `df[col]`, and save the result, which should be our formatted dataframe.

Let's turn to the shading problem next: let's say we want to give the total row a gray background so it stands out from the rest of the table.

The key to this lies in the `style` property of a Pandas dataframe, which enables us to apply *conditional formatting* (i.e. formatting based on certain rules) to the dataframe.

To achieve this, we would use the `.apply` method of the `style` property, along with a custom function that defines the conditional style to apply.

Let's create a new function in `drilldown.py` to implement this logic:

```
def style_total_row(df):
    def get_style(row):
        first_col = row.index[0]
        return [
            'background-color: #d9d9d9' if row[first_col] == 'Total' else ''
            for _ in row
        ]
    return df.style.apply(get_style, axis=1)
```

The `style_total_row` function accepts the drilldown dataframe `df`, and applies the shading we need. To achieve this, it does something interesting: it defines *another* function called `get_style` within its body!

In Python, a function defined within another is called a *nested function* or an *inner function*. Python considers a nested function to be local to the enclosing function's scope. In other words, any code outside of `style_total_row` cannot call `get_style`.

Getting to the logic of the `get_style` function, it operates on an individual row of a Pandas dataframe, and so takes in `row` as a parameter.

It then identifies the name of the first column of the dataframe using `first_col = row.index[0]`. The `index` property of a dataframe row is a list-like object containing the names of its columns, so `index[0]` gives the name of the first column.

The next line defines (and returns) the actual conditional style we want to apply:

```
return [
    'background-color: lightgray' if row[first_col] == 'Total' else ''
    for _ in row
]
```

The expression we're returning is a *list comprehension* which builds a new list by iterating through something (similar to how the dictionary comprehensions we've seen build new dictionaries).

In this case, we're iterating through the fields in the dataframe row, using `for _ in row`. We don't actually need to refer to the fields themselves, which is why we use `_`—a perfectly valid Python identifier, by the way—as the loop index here.

For each field, if the passed row is the total row (which we verify by checking if the value of the first column is "Total"), we add a peculiar string, `'background-color: lightgray'`, to the list we're constructing.

This notation comes from CSS, the language used to style web pages. I know I promised you don't need to learn CSS to read this book, but this particular piece of it should be obvious enough: we're telling Pandas to give a light gray background to every field in a total row.

We've now defined the conditional style we want to apply, but we still need to do the applying. The last line in `style_total_row` does this:

```
return df.style.apply(get_style, axis=1)
```

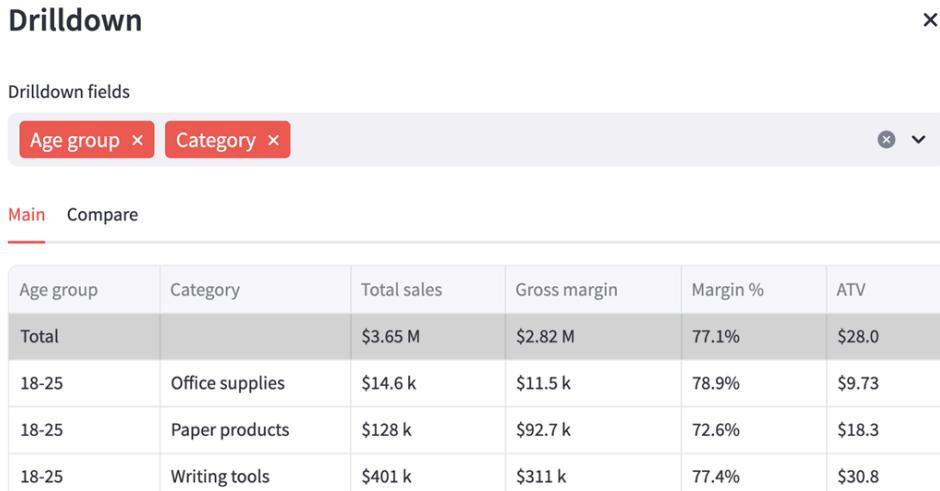
`.apply` here expects a function that accepts a dataframe row, so it can call it on every row (as we've seen before).

To complete our drilldown view, the final thing we need is to add the formatting and styling to `get_drilldown_table`:

```
...
from formatting import format_dataframe

...
def get_drilldown_table(df, dimensions):
    aggregated = get_aggregate_metrics(df, dimensions)
    with_total = add_total_row(aggregated, df, dimensions)
    formatted = format_dataframe(with_total, metrics)
    styled = style_total_row(formatted)
    return styled
```

And that's it! Our drilldown view is fully formed now. Check it out in figure 7.14.



The screenshot shows a 'Drilldown' interface with the following details:

- Drilldown fields:** Age group, Category.
- Main tab selected:** Main.
- Table Data:**

Age group	Category	Total sales	Gross margin	Margin %	ATV
Total		\$3.65 M	\$2.82 M	77.1%	\$28.0
18-25	Office supplies	\$14.6 k	\$11.5 k	78.9%	\$9.73
18-25	Paper products	\$128 k	\$92.7 k	72.6%	\$18.3
18-25	Writing tools	\$401 k	\$311 k	77.4%	\$30.8

Figure 7.14 Completed drilldown view with shaded total row and formatted values (see chapter_7/in_progress_08 in the GitHub repo for the full code)

Whew! That was a lot of work! However, we have more feature requests to address before we're done.

THE FRAGMENT-LIKE BEHAVIOR OF ST.DIALOG

If you paid especially close attention in Chapter 4 where we learned about Streamlit's execution model, there's one aspect of how we implemented the drilldown view that may be puzzling you.

To show the dialog, we nested it under a button like this (in `dashboard.py`):

```
if dd_button_container.button("Drilldown", use_container_width=True):
    drilldown(main_df, compare_df)
```

with `drilldown` being a function decorated with `st.dialog`. Within the drilldown, we can perform many interactions, such as selecting a dimension, or setting a filter.

But in previous projects, we've seen that the "clicked" state of `st.button` only holds for a single re-run, and that whenever we interact with something nested under a button, the app gets re-run again, and the button click gets cleared. In fact, we had to jump through a bunch of hoops in chapter 4 using `st.session_state` to get the behavior we wanted.

But we didn't need to do any of that in this case. Shouldn't interacting with the drilldown have caused a re-run with the button-click getting reset and the drilldown disappearing?

This doesn't happen because of some special behavior that `st.dialog` exhibits. When a user interacts with a widget within an `st.dialog`-decorated function, only the decorated function gets re-run, *not* the entire app!

In the above case, when someone selects a drilldown dimension, only the drilldown function gets re-run, and the button remains in the "clicked" state. `st.dialog` gets this behavior from a more general decorator called `st.fragment`.

7.6 Enabling deeplinks

Next up on the feedback list is a complaint about not being able to share views on the dashboard with other people. Naturally, the CEO frequently emails his reports about data she sees on the dashboard after applying a variety of filters and selections.

When receiving one of these, the subordinate spends a good few minutes trying to recreate what his boss saw on the dashboard, sometimes using trial-and-error to get the filters and date ranges right. "This," the CEO writes, "amounts to a collaboration tax."

Decision-making through data is not an isolated activity, or shouldn't be, at any enterprise. You generally want at least a few other pairs of eyes to validate the decisions you intend to make. This presents a real problem for users of our dashboard currently.

Using the dashboard, they may identify a trend or data point that is key to a decision the company is evaluating. However, if they are to share it with someone else, they have two options: either screenshot the app or give the sharee instructions on how to recreate the view.

Neither of these is ideal. A screenshot prevents the other person from interacting with the app further, and the other way is decidedly low-tech (imagine a user telling someone, "You're doing it wrong. You need to apply a date range of last year, filter for the 18-25 age group, and choose a monthly granularity!").

Wouldn't it be nice if the user could just copy-paste the URL they're looking at over chat, and the recipient could go to the URL to see exactly what the first user was seeing? After all, this works for many other websites. For instance, when you use a search engine like Google or DuckDuckGo, you can send someone directly to the search results page by sending them your search URL.

This functionality is called *deeplinking*, in the sense of linking someone "deep" into your website.

How do deeplinks work? Let's take an example from the search engine DuckDuckGo. If you search for "streamlit" on duckduckgo.com, the URL of the search results page will be something like:

```
https://duckduckgo.com/?t=h_&q=streamlit&ia=web
```

Copy-and-paste this URL in your browser and it'll take you directly to the search results page for the query "streamlit". The part of the URL where it says `q=streamlit` is what makes this possible. The URL has embedded information about the inputs entered by the first user, and DuckDuckGo uses this information to direct the second user to the right page.

If we apply this logic to our app, we need two things to implement deeplinks:

- A way to embed the inputs entered by a user in the app's URL
- Given such a URL, a way to repopulate these inputs in the app automatically

7.6.1 Using `st.query_params`

The part of the URL that contains the "extra" information after the actual address is called a *query string*. It is separated from the rest of the URL by a question mark (?) character. The query string is made up of several key-value pairs called *query parameters*, separated in turn by the ampersand (&) character.

For example, in the URL we discussed above, i.e. `https://duckduckgo.com/?t=h_&q=streamlit&ia=web`:

- The query string is `t=h_&q=streamlit&ia=web`
- The query parameters are: `t=h_` (key `t` and value `h_`), `q=streamlit` (key `q` and value `streamlit`), and `ia=web` (key `ia` and value `web`).

If our app were to have query parameters, what would they look like? Well, since the query string needs to capture the inputs entered by the user, it might be something like the following:

```
start_date=2024-08-01&end_date=2024-08-31&product_name=RoyalQuill
```

Essentially, the user's selections need to be part of the query string (and, therefore, the URL).

Streamlit allows you to manage query parameters through `st.query_params`, which is another dictionary-like object similar to `st.session_state`.

At any point, `st.query_params` contains whatever key-value pairs are in your app's URL query string. You can also modify the query string in your browser's address bar by modifying `st.query_params`.

The syntax for getting and setting parameters in `st.query_params` is identical to using a dictionary. For example, the code `st.query_params["pie_chart_dimension"] = "Gender"` would set the `pie_chart_dimension` parameter, updating the URL to include `pie_chart_dimension=Gender` somewhere.

You could also read in the value of the parameter like this:

```
dimension = st.query_params["pie_chart_dimension"]
```

Our solution, then, would involve something like figure 7.15.

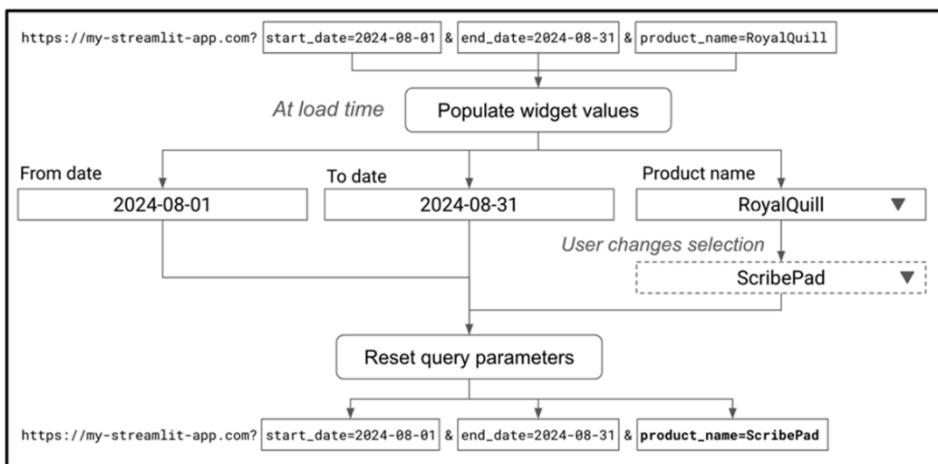


Figure 7.15 Approach for implementing deeplinking

When someone first navigates to our app, we should extract the query parameters from the URL if there are any.

We then set the values of the widgets based on those parameters. For instance, if we have `start_date=2024-08-01` as one of the query parameters, we would set the start date in the date range selector widget to 2024-08-01. If there are no query parameters, or no value specified for a particular widget, we don't set the value of the widget; instead we let the default behavior take over.

Then, when someone changes a selection in a widget, we *update* the query parameters to reflect that change, thus also changing the URL in the address bar. This way, if a user copies the URL, it always has the latest selections they've made in the app!

7.6.2 Setting widget defaults through `st.session_state`

There's one part of our scheme that hasn't been covered in any previous chapters. How do we programmatically set the value of an input widget?

Recall from chapter 4 that every Streamlit widget (or UI element) has a unique identifier called a widget key. The key is usually created automatically by Streamlit, but if you have two identical widgets, you have to give each one a key manually so Streamlit can tell them apart. One neat thing I didn't mention earlier is that every time you provide a widget a key, its value becomes accessible in `st.session_state`.

So if you have a dropdown input coded like this:

```
st.selectbox("Pick a field", ["Gender", "Product name"], key="select_dim")
```

You can access its value using `st.session_state["select_dim"]`

Importantly for us, we can also set its value, simulating a user selection, like this:

```
st.session_state["select_dim"] = "Product name"
```

One caveat is that we can only do this *before* the widget code is run. In other words, we can set the value of the widget key in `st.session_state` *first* and have the widget take on that value when it renders later, but you can't render the widget with a key *first* and *then* overwrite its value by setting the value of the key in `st.session_state`.

7.6.3 Implementing deeplinks

We now have all the information needed to build the deeplink functionality. Create a new file called `query_params.py` with the content shown in listing 7.2.

Listing 7.2 query_params.py

```
import streamlit as st

def get_param(key):
    return st.query_params.get(key, None)

def set_widget_defaults():
    for key in st.query_params:
        if key.startswith('w:') and key not in st.session_state:
            st.session_state[key] = get_param(key)

def set_params():
    query_params_dict = {}
    for key in st.session_state:
        if key.startswith('w:'):
            value = st.session_state[key]
            query_params_dict[key] = value
    st.query_params.from_dict(query_params_dict)
```

The `get_param` function gets the value of a particular query parameter given its name. It does so by using the `.get()` method of `st.query_params`, which—identically to that of a regular dictionary—returns a default value of `None` if the key does not exist.

`set_widget_defaults` populates the values of various widgets in the app from the query parameters by iterating through them and setting the value of each widget key in `st.session_state`.

Why do we have the following condition?:

```
if key.startswith('w:') and key not in st.session_state:
```

We don't necessarily want *every* key stored in `st.session_state` to appear in the URL, just those representing widgets. To ensure this, later on we'll prefix the string '`w:`' to every widget key we want in the query parameters. This gives us the flexibility of using `st.session_state` for other purposes should we need to while still being able to autopopulate widget values with it.

We also don't want Streamlit to try to set the widget value from `st.query_params` in every re-run of the app because then *users* wouldn't be able to change the value. Instead, we only want to set each widget value once, when the app is being loaded for the first time from the parameter-embedded URL. That's why we have the sub-condition `key not in st.session_state`.

`set_widget_defaults` fulfills the first part of what we need for deep links—the ability to populate widget inputs from the URL. However, we still need to change the query parameters whenever a user makes a selection.

That's what the `set_params` function does. It loops through `st.session_state`, gets the value of every widget key and stores them in a dictionary, `query_params_dict`. It then populates all of `st.query_params` directly from this dictionary using the `from_dict` method. As I illustrated earlier, we could also have set each value in `st.query_params`, but I wanted to show you this way too.

Of course, for this to work, all of the widgets we want in the query parameters must have keys defined, starting with '`w:`'. Therefore, we'll need to go through all of our code and add widget keys to each widget. Not a lot of fun, I'm afraid, but it has to be done.

Here are the changes we'll need to make, if you're following along:

CHANGES TO DATE_RANGE_PANEL.PY

In `date_range_panel.py`, there are three date selection widgets we show the user. We need to add keys to each of them. These would become:

```
start = st.date_input("Start date", value=THIRTY_DAYS_AGO, key="w:start")
end = st.date_input("End date", value=LATEST_DATE, key="w:end")
comparison = st.selectbox(
    "Compare to", ["MoM", "QoQ", "YoY", "Previous period"], key="w:compare")
```

The exact names for the keys don't matter as long as they start with `w::`. There is an additional wrinkle here though.

Notice that we currently assign default values `THIRTY_DAYS_AGO` and `TODAY` to the start and end date selectors respectively, through the `value` parameter in `st.date_input`.

When we use `st.session_state` to set widget values—as we're doing in the `set_widget_defaults` function we created earlier—Streamlit will throw an error if we *also* try to set the value using the `value` parameter. We can't use both methods, we have to choose one.

```

if 'w:start' not in st.session_state:
    st.session_state['w:start'] = THIRTY_DAYS_AGO
if 'w:end' not in st.session_state:
    st.session_state['w:end'] = LATEST_DATE
start = st.date_input("Start date", key="w:start")
end = st.date_input("End date", key="w:end")

```

Here, we've removed the `value` parameter from both widgets and added some logic at the beginning to set the same values using `st.session_state`. These lines must come before the widgets are defined.

For reference, the overall `date_range_panel` function is now:

```

...
def date_range_panel():
    if 'w:start' not in st.session_state:
        st.session_state['w:start'] = THIRTY_DAYS_AGO
    if 'w:end' not in st.session_state:
        st.session_state['w:end'] = LATEST_DATE
    start = st.date_input("Start date", key="w:start")
    end = st.date_input("End date", key="w:end")
    comparison = st.selectbox(
        "Compare to", ["MoM", "QoQ", "YoY", "Previous period"], key="w:compare")
    compare_start, compare_end = get_compare_range(start, end, comparison)
    st.info(f"Comparing with: \n{compare_start} - {compare_end}")
    return start, end, compare_start, compare_end

```

CHANGES TO FILTER_PANEL.PY

`filter_panel.py` has multiselects we need to add a key to:

```

...
def filter_panel(df):
    ...
    with st.expander("Filters"):
        ...
        for idx, dim in enumerate(filter_dims):
            with filter_cols[idx]:
                ...
                filters[dim] = st.multiselect(
                    dim, unique_vals, key=f'w:filter|{dim}')
return filters

```

In this case, since multiple widgets are populated through a loop, we use the f-string `f'w:filter|{dim}'` as the key, using the dimension name `dim` to differentiate between the keys.

CHANGES TO PIE_CHART.PY AND TIME_SERIES_CHART.PY

In `pie_chart.py`, add a key to the `st.selectbox` assigned to `split_dimension`:

```
...
    split_dimension = st.selectbox(
        "Group by", pie_chart_dims, key="w:pie_split")
```

Similarly, in `time_series_chart.py`, add keys to `grain` and `split_dimension` in the `time_series_chart` function:

```
...
def time_series_chart(df):
    with st.container(border=True):
        ...
        grain = grain_col.select_slider(
            "Time grain", grain_options, key="w:ts_grain")
        split_dimension = split_col.selectbox(
            "Group by", ["None"] + time_chart_dims, key="w:ts_split")
    ....
```

With the widget keys in place, we can now call the relevant functionality we defined earlier in `query_params.py` from `dashboard.py`:

```
...
from query_params import set_widget_defaults, set_params

st.set_page_config(layout='wide')
set_widget_defaults()

...
set_params()
```

Pay attention to exactly where we've placed the calls to `set_widget_defaults` and `set_params`. As mentioned earlier, we can only use `st.session_state` to set widget key values *before* any of the widgets are created, so the call to `set_widget_defaults()` needs to go right at the top (just after `st.set_page_config(layout='wide')` which needs to be the first command).

On the other hand, the query parameters need to capture changes to *any* widget that the user has changed, so the call to `set_params` has to go at the very *end* of `dashboard.py`, after all the widgets have been created.

Let's test out our deeplinks! Save everything and re-run the app. Then try making the following selections in the app:

- Set "Start date" to 2024/07/01, and "Compare to" to YoY.
- Set the "Time grain" slider in the line chart to Week.

If you now check the URL in your browser's address bar, it should look something like:

```
http://localhost:8501/?w%3Ats_grain=Week&w%3Acompare=YoY&w%3Aend=2024-08-31&w%3Astart=2024-07-01&w%3Apie_split=Age+group
```

When a web URL contains certain special characters, such as a colon (:) or a space, it is converted into *percent-encoded* characters to ensure that browsers and web servers interpret them correctly. Each special character is usually replaced by a % sign followed by a two-digit hexadecimal code representing the original character in the ASCII standard. One exception is the space character, which, when it appears in the query parameters part of the URL, is encoded as a + sign.

In our case, the following substitutions have occurred:

- The colon character has become %3A, so `w:ts_grain` becomes `w%3Ats_grain`
- The space character has become +, so `Age group` becomes `Age+group`

Reverse those substitutions and the URL becomes:

This is pretty much what we expected—the selections we made are reflected in the URL (along with the value of the pie chart dimension selectbox, which gets a non-blank value—`Age group`—by default, which is automatically captured in the URL).

NOTE The URL starts with `http://localhost:` since we're currently developing locally. When we deploy our app, the `localhost` part will be replaced by whatever the address of the app is. For instance, if we deploy to Streamlit Community Cloud under <https://ceo-dashboard.streamlit.app>, our URL with query parameters would look something like https://ceo-dashboard.streamlit.app?query_param1=value1&query_param2=...

Next, paste the original URL you copied into another browser tab and navigate to it. Unfortunately, the app throws an error (see figure 7.16).

```
StreamlitAPIException: DateInput value should either be an date/datetime or a list/tuple of 0 - 2 date/datetime values
Traceback:
File "/Users/aneevdavis/projects/streamlit_book/streamlit-in-action/chapter_7/in_progress_09/dashboard.py"
    start, end, compare_start, compare_end = date_range_panel()
                                         ^^^^^^^^^^^^^^^^^^
File "/Users/aneevdavis/projects/streamlit_book/streamlit-in-action/chapter_7/in_progress_09/date_range_pa
    start = st.date_input("Start date", key="w:start")
                                         ^^^^^^^^^^
```

Figure 7.16 We get an error when we inadvertently pass a string to an `st.date_input` (see `chapter_7/in_progress_09` in the GitHub repo for the full code)

The error claims that we tried to pass the wrong kind of value to a "DateInput", presumably the "Start date" and/or "End date" widgets.

The issue here is that in the (parsed) URL above, the value given to the start date widget (with the key `w:start`) is the string "2024-07-01":

```
w:start=2024-07-01
```

When Streamlit tries to assign this value to the date input widget when it's eventually defined, it results in an error because `st.date_input` expects a date object, not a string.

There's a similar problem with our filter inputs: these widgets expect lists (as you can select multiple values), but we're passing strings.

We need some special handling logic for when the value to be set is not a string but a date or a list.

Firstly, when setting the value of such a widget in `st.query_params`, let's add a prefix to denote that the value we're placing is a list or a date—say `L#` for a list and `D#` for a date. Here's the `set_params` function in `query_params.py` with this modification:

```

import streamlit as st
from datetime import date

...
def set_params():
    query_params_dict = {}
    for key in st.session_state:
        if key.startswith('w:'):
            value = st.session_state[key]
            if value:
                if isinstance(value, list):
                    value = f'L#{','.join(value)}'
                elif isinstance(value, date):
                    value = f'D#{value.isoformat()}''
                query_params_dict[key] = value
    st.query_params.from_dict(query_params_dict)

```

Here, before adding a value to `query_params_dict`, we check its type using `isinstance`. If it's a list, we convert it to a string in a specific format (e.g. `['M', 'F']` becomes `L#M,F`). If it's a date, we convert it into a different format (e.g. `2024-08-01` becomes `D#2024-08-01`).

We also need the reverse logic to decode these string formats and convert them into the original values. This part goes in the `get_param` function, which we'll rewrite entirely:

```

def get_param(key):
    if key not in st.query_params:
        return None
    value = st.query_params[key]
    if value.startswith('L#'):
        return value[2:].split(',')
    if value.startswith('D#'):
        return date.fromisoformat(value[2:])
    return value

```

As you can see, for a particular key, if the value is a string that starts with one of our special prefixes—either `L#` or `D#`—we do the reverse transformation, converting the string to the original list or date respectively.

When we use this returned value in `set_widget_defaults`, it will thus be in the expected type, identical to the value that the user originally set, removing the error.

You can see for yourself by retrying the earlier steps. You should see figure 7.17 now, demonstrating that you can now copy and paste the current URL of your dashboard to show others exactly what you're seeing.

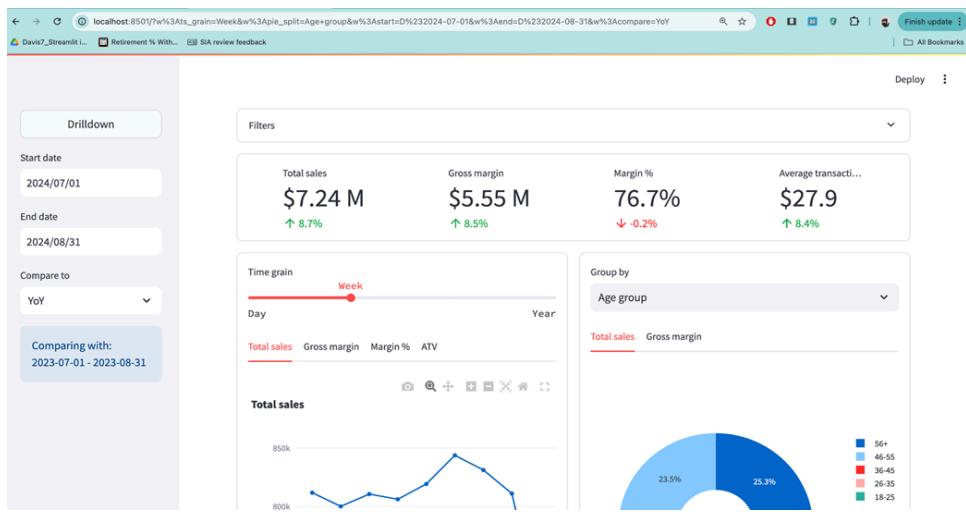


Figure 7.17 The widgets in the dashboard are populated based on the URL values (see chapter_7/in_progress_10 in the GitHub repo for the full code)

Explore the deeplinks some more. Note n' Nib's execs can now spend less time fiddling with the dashboard and more time making decisions that benefit the company!

7.7 Sourcing data from a data warehouse

We've addressed all of the user feedback on the dashboard, but there's one glaring practical problem with it we have yet to talk about: the data we display in the dashboard is sourced from a static CSV file.

I chose this approach as I wanted us to focus primarily on what we do with the data once we have it. Reading a static CSV file is probably the easiest way to ingest data in our app. However, there are multiple issues with this:

- While a CSV is manageable when dealing with a small amount of data, it quickly becomes inefficient when handling large datasets.
- We can't query the data flexibly at the source, and instead have to load it in memory to perform operations like filtering, aggregations, and joins.

In the real world, we would generally store this data in a *data warehouse*, which is a specialized system designed to manage large volumes of structured data. In this section, we'll swap out our CSV for a table in a data warehouse—specifically Google BigQuery.

7.7.1 Getting our data into BigQuery

Google BigQuery is a cloud-based data warehouse service that's part of the Google Cloud Platform (GCP). It allows you to efficiently store and analyze massive datasets using a language called *Structured Query Language* (SQL), without needing to manage infrastructure or worry about scaling.

To begin, you'll need to set up a GCP account, which you can do at cloud.google.com. You'll probably need to enter details of a payment method such as a credit card, but you won't get charged since we're only going to be using free resources for this exercise.

When you create a new account, Google will also create a *GCP project* for you. In GCP parlance, a project is a container for organizing and managing your Google Cloud resources. You need one to use BigQuery; feel free to use the default one created for you or to create a new one. A project has a unique ID; you can choose what this is when you create a project, but the default one is a randomly generated string. For instance, my default project ID was `dauntless-brace-436702-q0`.

Next, let's go to BigQuery itself. Google Cloud is so vast and offers so many products and services that its UI may be intimidating to a beginner. The most reliable way to find BigQuery is probably to enter the search string "bigquery" in the search box at the top (see figure 7.18)

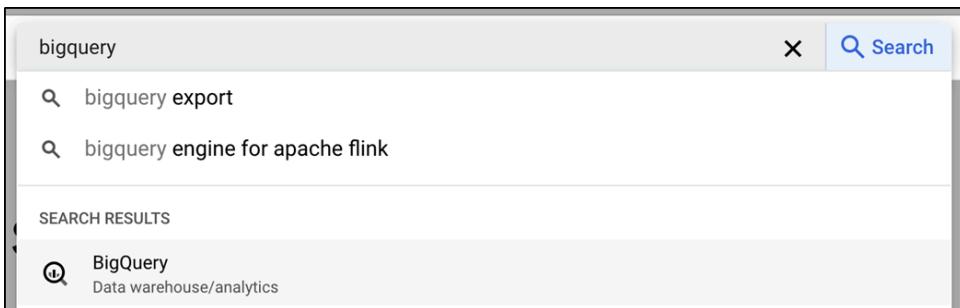


Figure 7.18 The most reliable way to find something on GCP is to use the search bar.

Once on the BigQuery page, you should be able to see your BigQuery resources categorized under your GCP projects in an Explorer side panel to the left (see figure 7.19). "Resources" here means things like "queries", "notebooks", "workflows" etc., all of which you may ignore.

What we're trying to do is to create a BigQuery table by uploading our CSV file. Before we can do this, we need to create a dataset. A BigQuery dataset is just a way to organize your tables within a project.

Make your first dataset by clicking the three dots next to your project ID in the Explorer panel and then "Create dataset" (figure 7.19).

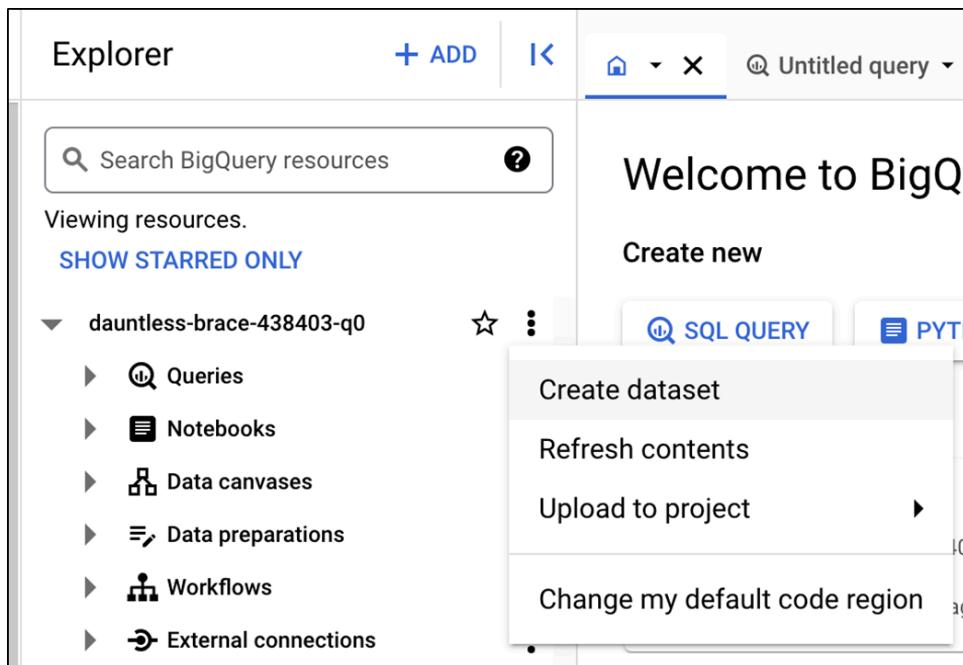


Figure 7.19 The Explorer pane in BigQuery

This will open a screen where you can configure the dataset. All you need to enter is a name (I chose `sia_ceo_dashboard`); you can use the defaults for the remaining options.

Once created, your dataset should appear in the Explorer panel. Click the three dots next to it and "Create table" to get to the table creation screen where we can upload our file.

Select "Upload" under the "Create table from" options, and "CSV" as the file format. You can then select the `sales_data.csv` file from your local disk. You'll need to pick a name for the table (`sales_data` works). The remaining options should be straightforward and will likely be auto-populated: your project ID, and the name of the dataset you just created. Figure 7.20 shows what the screen looks like.

Create table

Source

Create table from —
Upload

Select file * —
sales_data.csv X BROWSE ?

File format —
CSV

Destination

Project * —
dauntless-brace-438403-q0 BROWSE

Dataset * —
sia_ceo_dashboard

Table * —
sales_data

Maximum name size is 1,024 UTF-8 bytes. Unicode letters, marks, numbers, connectors, dashes, and spaces are allowed.

Table type —
Native table

Schema

Auto detect

ⓘ Schema will be automatically generated.

CREATE TABLE CANCEL

Figure 7.20 The table creation screen in BigQuery

Check the "Auto detect" box under "Schema" so you don't have to enter it manually. Then click the button at the bottom to actually create your table.

At this point, your data is in BigQuery, and the table should appear under your dataset in Explorer. If you like, you can click into it and go to the "Preview" tab to see the data.

7.7.2 Setting up the Python-BigQuery connection

We can now access our data in the BigQuery interface, but we also need to be able to connect to it from our Python code.

ENABLING THE BIGQUERY AND BIGQUERY STORAGE APIs

First, we need to enable a couple of BigQuery-related APIs in our GCP project: the BigQuery and BigQuery Storage APIs. The BigQuery API is what enables us to connect to BigQuery in the first place, while the Storage API makes it faster to ingest data into a Pandas dataframe.

In each case, the corresponding result should lead you to a page when you can enable the API.

CREATING A SERVICE ACCOUNT

Since our app will connect to BigQuery programmatically, we need a *GCP service account* to handle authentication. A service account is a special type of account that belongs to your application instead of to an individual user. It enables your app to authenticate and interact with Google Cloud services, including BigQuery.

To create a service account, first find "IAM & Admin" and then "Service accounts" in the Google Cloud navigation menu (or better still, search for "service accounts" and click the first result).

In the "Service Accounts" page, click the option to create one. This screen will ask you for a service account name (I used `sia_service_account`) and description. Once you've created the account, you'll also need to grant it access to your project on the same screen. Choose the role "Viewer" when you do this.

Your service account should now appear in the Service Accounts page.

CREATING A SERVICE ACCOUNT KEY

We have a service account that can access our BigQuery resources, but we still need to obtain the credentials that will let our Streamlit app act as the service account. For this, we require a service account key.

Find the account you just created on the Service Accounts page, click the three dots under "Actions" next to it, and then click "Manage keys."

Click "ADD KEY" > "Create new key" and select "JSON" as the key type. When you click "Create", your computer should automatically download a JSON file. Inspect this file in a text editor. It should contain the credentials you need to access BigQuery from your app, as well as additional details such as your project ID.

GENERATING SECRETS.TOML

The credentials we just obtained must be kept secret as they allow anyone who has them to read your BigQuery data. Recall from Chapter 5 that the optimal way to maintain confidential info in Streamlit is to use a `secrets.toml` file in conjunction with `st.secrets`.

Unfortunately, the credential file we have is in JSON, so we need to convert it to TOML. You can do this manually, but let's use Python instead.

First create a `.streamlit` folder to hold your `secrets.toml`.

Rename your JSON file to `sia-service-account.json`, then open a Python shell from the same folder and type in the following commands:

```
>>> import json
>>> import toml
>>> with open('sia-service-account.json') as json_file:
...     config = json.load(json_file)
...
>>> obj_to_write = {'bigquery': config}
>>> with open('.streamlit/secrets.toml', 'a') as toml_file:
...     toml.dump(obj_to_write, toml_file)
...
'[bigquery]\ntype = "service_account"\nproject_id = "dauntless-..."'
<Rest excluded for brevity>
```

NOTE You may first need to run "pip install toml" to get this to work.

All we're doing here is opening the JSON file we got from GCP, reading it into a Python dictionary, and writing it back to `secrets.toml` under the key "bigquery". If you now open `secrets.toml`, you should be able to see the credentials in TOML format.

```
[bigquery]
type = "service_account"
project_id = "dauntless-brace-436702-q0"
private_key_id = ...
...
```

7.7.3 Updating the dashboard to load data from BigQuery

It's time to update our code to source data from BigQuery instead of a static CSV file. We need to install three new Python modules to enable this, so go ahead and enter the following commands into a terminal window:

```
pip install google-cloud-bigquery
pip install google-cloud-bigquery-storage
pip install db-dtypes
```

The first two are needed to access the BigQuery and BigQuery Storage APIs. `db-dtypes` is required to enable converting the data returned by BigQuery to a Pandas dataframe.

Since we wrote our code in a modular way, the only thing we need to change is the implementation of the `load_data` function in `data_loader.py`, and the rest of our app should work as before. This is an advantage of the "separation of concerns" principle we discussed in chapter 3.

Listing 7.3 shows the new `data_loader.py` with `load_data` re-implemented to use BigQuery.

Listing 7.3 `data_loader.py` reimplemented to use BigQuery

```
import streamlit as st
from google.cloud import bigquery, bigquery_storage

DATASET = "sia_ceo_dashboard"
TABLE = "sales_data"

def load_data():
    service_account_info = st.secrets["bigquery"]
    client = bigquery.Client.from_service_account_info(service_account_info)
    creds = client._credentials
    storage_client = bigquery_storage.BigQueryReadClient(credentials=creds)
    project_id = service_account_info["project_id"]
    query = f"SELECT * from `{project_id}. {DATASET}. {TABLE}`"
    query_job = client.query(query)
    result = query_job.result()
    return result.to_dataframe(bqstorage_client=storage_client)
```

We keep a couple of constants at the top (`DATASET` and `TABLE`) to hold the names of the dataset and table we created in BigQuery.

Within `load_data`, we first save the credentials from the `bigquery` key in `st.secrets` to `service_account_info`. We then pass in these credentials to create a BigQuery client (essentially an object that contains the methods and abstractions needed to interact with BigQuery):

```
client = bigquery.Client.from_service_account_info(service_account_info)
```

We want to use the same credentials in the BigQuery Storage API client too, so we extract the credentials from the BigQuery client and use them to initialize `storage_client`:

```
creds = client._credentials
storage_client = bigquery_storage.BigQueryReadClient(credentials=creds)
```

Our connection is now established.

Tables in BigQuery are referred to using a dot-separated combination of the project ID, dataset name, and table name. For instance, the table I created would be referenced as:

```
dauntless-brace-436702-q0.sia_ceo_dashboard.sales_data
```

We obtain the project ID from the credentials (`project_id = service_account_info["project_id"]`) and the dataset and table names from the constants we created above.

We use the table reference to construct a SQL query like this:

```
query = f"SELECT * from `{project_id}.{DATASET}.{TABLE}`"
```

We'll encounter more SQL in chapter 8, but for the moment, all you need to understand is that "SELECT * from <table>" means "get me all the columns from <table>". Essentially, we're telling BigQuery to return all the data in the table.

Though we're not doing it here, we could have used a different SQL query to obtain some *subset* of the data; we couldn't have done this if we were still using a CSV file.

The next two lines execute the query itself, wait for it to finish, and save the results to `result`:

```
query_job = client.query(query)
result = query_job.result()
```

Lastly, we convert the result to a Pandas dataframe, utilizing the BigQuery Storage client for optimized performance, and return the dataframe.

If you execute `streamlit run dashboard.py` again (you can't just re-run the app in the browser since we're using `st.cache_data` and simply re-running would return a previously cached version), the app will now pull data from BigQuery!

7.7.4 Notes on deployment to Streamlit Community Cloud

In chapter 5, we explored how to deploy our apps to Streamlit Community Cloud. The process to do so remains the same for our metrics dashboard, but I want to call out a couple of things.

The first is related to where we store the data. When deploying, if you're using the static CSV approach to source the data, you need to commit the CSV file in git, essentially storing it in your GitHub repo.

If you're using BigQuery instead, the CSV is obviously not required, and you don't have to check it into your repository. However, you do need to configure your GCP credentials in Streamlit Community Cloud using the same process we used in Chapter 5.

You'll also need to create a `requirements.txt` with all the modules we're using and need Community Cloud to install. As discussed in chapter 5, you can use the `pip freeze` command to identify the specific versions of the libraries we're using.

An example `requirements.txt` for the dashboard is provided in listing 7.4.

Listing 7.4 requirements.txt

```
humanize>=4.10.0
streamlit>=1.38.0
plotly>=5.23.0
google-cloud-bigquery==3.25.0
google-cloud-bigquery-storage==2.26.0
db-dtypes==1.3.0
```

We're finally all set to release version 2.0 of our dashboard! Without a doubt, there will be more feedback later, and each iteration will refine our dashboard further.

For now, however, it's time to bid farewell to Note n' Nib and its data needs. In the next chapter, we're shifting gears from data insights to interactive tools, as we dive into a web app for creating, storing, and sharing haikus.

7.8 Summary

- *Launching* an app is only the first part of making it successful. You also have to *land* it, making sure that it serves your users' needs. For this, it is critical to hear from users directly.
- `st.select_slider` is a cross between `st.selectbox` and `st.slider`, used when you want to impose a logical order between options.
- `st.metric` can show the delta associated with a metric, i.e. how a value has changed over time.
- `st.dialog` is a decorator that lets you create a modal dialog, an overlay that blocks interaction with the rest of the app.
- You can use `st.container` to create placeholders in your app, only rendering the content you want to show when it's available.
- Pandas dataframes have a `style` property that can be used to set conditional rules that modify how they are displayed on the screen.
- `st.query_params` is a dictionary-like object that lets you read and update URL query parameters—this can be used to enable deeplinks in an app.

- A data warehouse is a specialized system designed to store and retrieve large amounts of data.
- Google BigQuery—part of GCP—is an example of a data warehouse. To enable an app to connect to it, you need to create a service account with a key, and record the credentials in `st.secrets`.

8 Building a CRUD app with Streamlit

This chapter covers

- Setting up a relational database for persistent storage
- Performing CRUD operations using SQL
- Developing a multi-page Streamlit app
- Creating shared database connections in a Streamlit app
- Authenticating users

In 1957, a science fiction author named Theodore Sturgeon famously said, "ninety percent of *everything* is crud". While this was originally a cynical defense of the genre of science fiction—the point being that it was no different from anything else in that regard—the adage has since taken on a different meaning, becoming the worst-kept secret in software engineering: Ninety percent of everything is CRUD.

By CRUD, I'm referring to Create, Read, Update, and Delete, the four mundane operations that appear repeatedly in almost any notable piece of software.

Think about it. Social media platforms like Facebook revolve around creating posts, reading feeds, updating profiles, and deleting content. E-commerce sites manage products, orders, customer accounts, and reviews through similar operations. Even something as simple as Notepad on Windows centers around creating, reading, updating, and deleting text files.

Mastering CRUD operations is essential for building a strong foundation in software design, as their implementation often involves tackling non-trivial challenges. In this chapter, we'll create a CRUD application with Streamlit, implementing these operations from scratch while covering related topics such as user authentication.

8.1 Haiku Haven: A CRUD app in Streamlit

For our excursion into CRUD, we're going with the Japanese art of *haiku*—specifically, we will create a website that lets users write and share their own haikus.

For those not intimately familiar with Japanese literature, a haiku is a short three-line poem conforming to certain rules: the first and the third lines must have five syllables, while the second must have seven. For instance, here's one I wrote about Streamlit:

So many web apps!

With Python can I make one?

Then I tried Streamlit.

I know, right? I sometimes wonder if I missed my calling too. Regardless, you'll notice that the poem adheres to the 5-7-5 syllable rule I mentioned above, and is therefore a valid haiku.

Haiku Haven will be a place where budding poets can author, refine, and manage haikus. It will enable users to *create* haikus from scratch, *read* what they've created, *update* a haiku once they've created it, and *delete* it if they decide it doesn't pass muster.

8.1.1 Stating the concept and requirements

As usual, we'll start by stating the concept of our app succinctly:

CONCEPT Haiku Haven, a website that allows users to create, edit, and manage haikus

This concept and our earlier discussion about CRUD should give you a basic idea of what we want, but let's spell out the concrete requirements so we're on the same page about the Haiku Haven vision.

REQUIREMENTS

A user of Haiku Haven should be able to:

- create and log in to an account with a username and password
- create haikus under their username
- view the haikus they've created
- update haikus
- delete their haikus

Hopefully you'll realize how frequently you encounter this kind of app. If you replace the word "haiku" in the requirements with "image", you get a barebones version of Instagram. Substitute "task" to get a productivity tool like Asana or "post" to get Twitter or WordPress.

The point is that these requirements aren't just about haikus—they represent a universal pattern in software design. Almost every app revolves around managing some kind of data, founded on CRUD operations.

By building Haiku Haven, you're not just creating a playful app for poetry lovers; you're learning how to construct the essential workflows of modern software. You'll tackle user authentication, data storage, and retrieval—skills that apply to nearly any app or system you might create in the future.

WHAT'S OUT OF SCOPE

We could fit a lot of functionality into Haiku Haven (think of everything you can do on Twitter), but we only have one chapter, so we'll focus on the absolute core stuff. That means we *won't* concern ourselves with:

- Making haikus visible to and searchable by other users—haikus are private to the author
- "Social" features such as liking, commenting, and sharing
- Auxiliary functionality like pagination
- Advanced security features (though we'll get the basics right)

8.1.2 Visualizing the user experience

Haiku Haven will be our first multi-page app. We'll need to craft multiple experiences or flows—the account creation flow, login and logout, and the actual CRUD part (creating, reading, updating, and deleting haikus).

Figure 8.1 attempts to sketch out what the different portions of our app might look like.

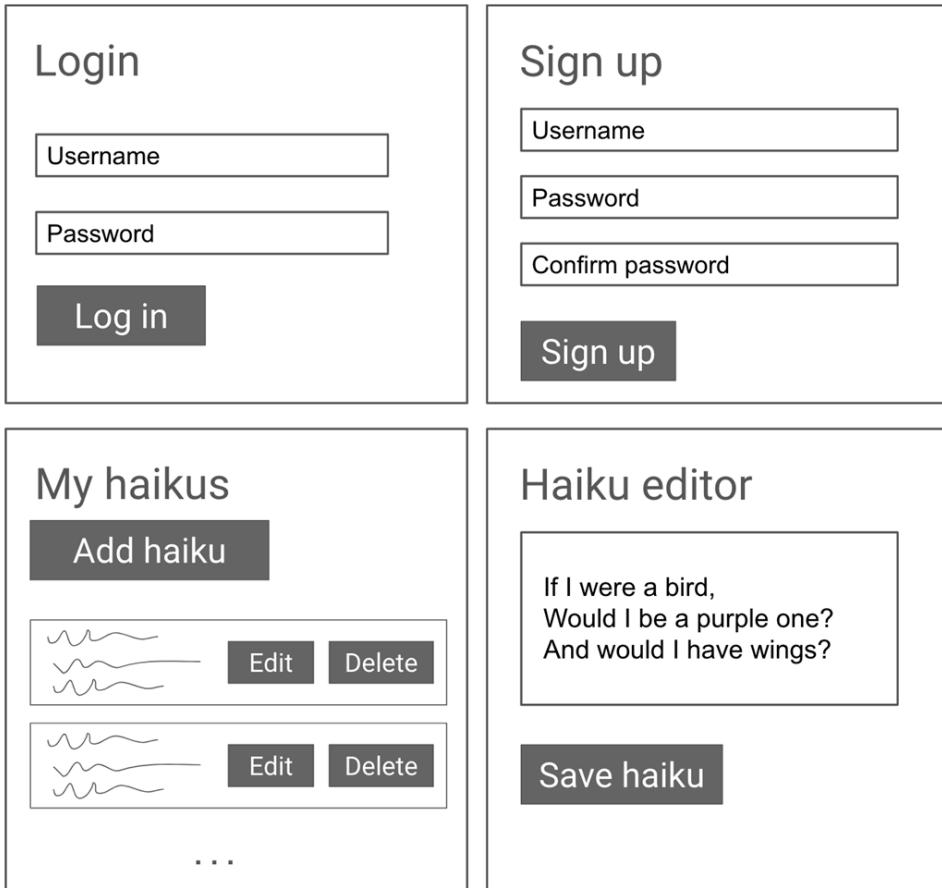


Figure 8.1 Rough sketches of the pages we want in Haiku Haven

Because of how common these flows are in various common apps, I won't take too much space here to explain them in detail, but here are a few highlights:

- The login page uses a password for authentication, which you can set in the signup page
- A "My haikus" page represents the logged-in experience within the app, letting users create, view, edit, or delete their haikus.
- There's also a haiku editor page where the authoring of haikus takes place.

8.1.3 Brainstorming the implementation

Given that Haiku Haven represents CRUD web apps—and thus, by Sturgeon's law, 90% of *all* web apps—it stands to reason that its implementation should involve some very common patterns.

Indeed, the design we'll use consists of three components that are seen in most live online applications: a frontend, a backend, and a *database*. Figure 8.2 lays out this approach.

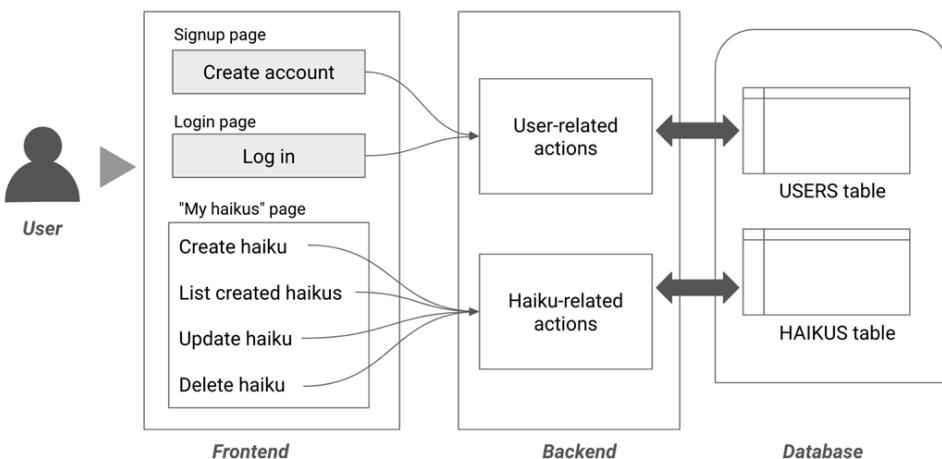


Figure 8.2 Design for our app showing a frontend, backend, and database

The frontend, as we've seen in prior apps, consists of the widgets that the user interacts with. Each major action, such as creating an account, or updating a haiku, calls a corresponding function in the backend.

We'll divide the functions available in the backend into two groups: one that includes actions related to users, like creating an account or authenticating a user, and another that includes haiku-related actions—creating, reading, updating, or deleting haikus.

The interesting section here—one that's new to us—is the database, which is used to permanently store information related to users and haikus in *tables*. As we'll soon see, the database also makes it easy to retrieve the information we've stored.

While our app is intentionally simple, it captures the core elements of most web applications: a frontend for user interaction, a backend for handling requests, and a database for storing and retrieving data. These three pillars work together seamlessly, forming the foundation of countless applications, whether simple or complex.

8.2 Setting up persistent storage

One of the most important deficiencies in the apps we've created thus far is the lack of persistent storage. Essentially, in all of our apps so far, if the user closes their browser window, they lose their data and progress. This won't do at all for Haiku Haven; we need users to be able to save and access their haikus after they log out and log back in again at some point in the future—we need to store data *outside* of the app itself.

There are several different approaches to solving the problem of data storage, but we'll use a fairly common technique: a relational database.

8.2.1 Relational database concepts

A *relational database* is a type of data storage system that organizes data into structured *tables*, where each table consists of *rows* (also called *records*) and *columns* (also called *fields*). A row represents some kind of entry or entity, and a column is an attribute of the entity. A *schema* defines the structure of these tables, including the *data types* of each column and the *relationships* between different tables.

Relational databases rely on a language called *Structured Query Language* (SQL) to create, manage and *query* tables.

If some of this rings a bell, it's likely because we've been dealing with this kind of thing for a while now. In Chapters 6 and 7, we worked with Pandas dataframes that also handle tabular data—except that dataframes are stored in memory while a program is running, while a database is used for *persistent* storage, i.e. storage that exists even when a program finishes running.

We also encountered SQL briefly in Chapter 7, where we used it to fetch the rows of the sales data that we stored in Google BigQuery. Indeed BigQuery is often considered to be a relational database itself, though of a different kind than the one we'll use in this chapter.

8.2.2 Haiku Haven's data model

To understand all of this better, let's try and figure out how we can *model* Haiku Haven's data in a relational database.

Broadly speaking, modeling data for an app consists of the following steps:

- Identifying the *entities* involved in the app
- Defining the *relationship* between those entities
- Listing the *attributes* of each entity
- Converting the entities, attributes and relationships into a relational database schema

IDENTIFYING THE ENTITIES

Generally speaking, a good way to identify the entities involved in an app is to list all the *nouns* that represent core concepts in the app. For instance, if you take Twitter, the following might all be considered as entities: users, tweets, retweets, direct messages, mentions, followers, hashtags, etc.

Haiku Haven is way simpler, of course. We can fairly easily identify the two key entities our app will need to handle: **haikus** and **users**.

DEFINING THE RELATIONSHIP BETWEEN THE ENTITIES

The relationship between any two entities should be defined in terms of the *nature* and *cardinality* of the possible interaction between them. In English, that means you should lay out *how* one entity is related to the other, and *how many* of each entity can be on each side of this relationship.

For instance, haikus and users are related because a user **can write** a haiku (the "nature" we spoke of above). Also, one user can write many haikus, while a particular haiku can only be written by one user. So the relationship between a user and a haiku is **one-to-many** or 1:n (the "cardinality").

LISTING THE ATTRIBUTES OF EACH ENTITY

The attributes of an entity are the fields that describe it. In our case:

- A user has a **username** and a **password**. In real life, we'd probably also want to capture the name of the user or the time a user's account is created, but let's keep things simple.
- A haiku has its **text**, and an **author** (who happens to be a user). We should also give each haiku a **numeric ID** for easy reference. The **creation time** for a haiku may be important to display in the app, so let's consider that too.

All of this can be represented in an entity-relationship diagram (or ER diagram) as shown in figure 8.3.

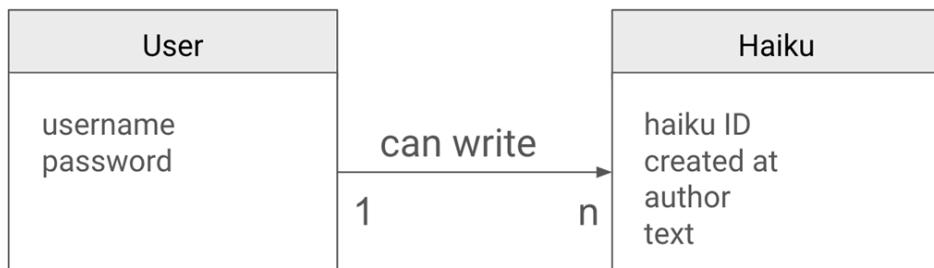


Figure 8.3 Entity-Relationship (ER) diagram showing the User and Haiku entities

Of course, ER diagrams for apps in the real world tend to be much, *much* more complex than this, but I hope this serves to illustrate the concept.

CONVERTING EVERYTHING INTO A DATABASE SCHEMA

The exercise of coming up with an ER diagram is helpful in visualizing the data model, but the end outcome we actually want is a schema that we can use in our relational database.

There's no hard-and-fast rule to convert entities, attributes, and relationships into tables in a database, but *generally speaking*: entities become tables, attributes become columns, one-to-many relationships become *foreign keys* (more on this in a moment), and many-to-many relationships become their own tables.

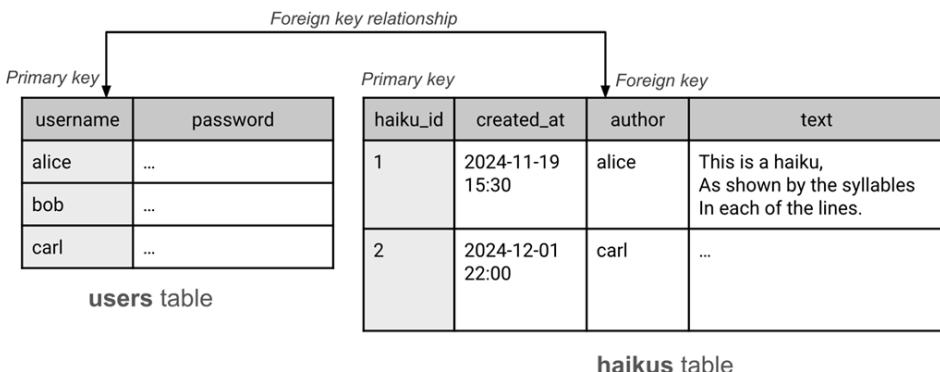


Figure 8.4 Database schema with a foreign key relationship between the `users` and `haikus` tables

In our case, as figure 8.4 shows, we'll have two tables, `users` and `haikus`, with the attributes we discussed earlier as columns. Each row in `users` represents a single user, and each row in `haikus` is a single haiku. Additionally, each table has a *primary key*, which is a field that can be used to uniquely identify any row in the table. For `users`, the primary key is the `username` field (which makes sense since every user has a `username` and no two users can have the same `username`). For `haikus`, it's `haiku_id`.

The relationship between `users` and `haikus` is reflected in the `author` column in the `haikus` table, which contains a `username` that must appear in the `users` table. Such a column (`author`) is called a *foreign key* as it points to the primary key (`username`) of another ("foreign") table.

8.2.3 PostgreSQL: A real relational database

Enough theorizing! Let's now get our hands dirty with a real relational database. The one we'll use in this chapter is PostgreSQL (pronounced "post-gress-cue-ell"), one of the oldest, most robust, and most popular databases in the industry.

NOTE Google BigQuery, which we encountered in Chapter 7, can also be considered a relational database (though it's better described as a cloud-based data warehouse). While BigQuery is optimized for analytical use cases (like querying a large amount of data to generate reports or uncover trends), PostgreSQL is better suited for transactional use cases, such as handling frequent small updates to individual records, and maintaining data consistency across concurrent operations.

INSTALLING POSTGRESQL

Towards the end of this chapter, when we deploy Haiku Haven to production, we'll use a free cloud service to set up PostgreSQL. However, for local development, we first need a local installation.

To install PostgreSQL, download the installer for your operating system from <https://www.postgresql.org/download/> and run it, following the on-screen instructions. For most options, you can accept the defaults. During the installation, you'll be prompted to set a password for the database superuser. Make sure to note this password—you'll need it in the next section.

Once installation is complete, you should have access to the PostgreSQL shell, called `psql`, which is located in the `bin/` directory of your installation folder. If you kept the default options, this would typically be:

- `/Library/PostgreSQL/17/bin/psql` on macOS
- `C:\Program Files\PostgreSQL\17\bin\psql` on Windows

To simplify access, you should configure your system so you can run the `psql` command directly from your terminal. This requires adding the `bin/` directory to your system's environment variables. You may already have done something similar for the `streamlit` command, which you can find detailed steps for in Appendix A (section A.4).

If adding the path to your environment variables isn't feasible for any reason, you can still use `psql` by typing its full path instead of just the command. For example:

- macOS: `/Library/PostgreSQL/17/bin/psql`
- Windows: `"C:\Program Files\PostgreSQL\17\bin\psql"` (include quotes)

Throughout the rest of this chapter, substitute the full path whenever you see `psql`, if necessary.

CREATING A DATABASE FOR HAIKU HAVEN

Once you've set up the `psql` command, log in to your local PostgreSQL instance by running the following command:

```
psql -U postgres -d postgres
```

When prompted for a password, enter the one you configured during installation. This command logs you into PostgreSQL as the default user (`-U`) `postgres`—which is an administrative user that's allowed to do whatever they want. The `-d` specifies that you want to connect to the default database, which is `~`—rather confusingly if you ask me—also called `postgres`. You should see now see the `psql` prompt that looks like this:

```
postgres=#
```

When your app talks to PostgreSQL, you don't want it to be running with broad administrative privileges—that would be a security nightmare. Instead, let's create a more narrowly-scoped user. Enter this into the `psql` prompt, replacing the quoted string with a password of your choice:

```
CREATE USER haiku_lord WITH PASSWORD '<Pick a password you like>';
```

Obviously, you can use whatever username you like, but I'm going to assume we're going with `haiku_lord`. If that works (don't forget the ending semicolon!), you should get an output line that just says `CREATE ROLE`.

Before you can create tables in PostgreSQL, you need to first make a **database** (which you can think of here as a container for tables). The `haiku_lord` user you just created can't do that yet, so enter this command to let it:

```
ALTER USER haiku_lord CREATEDB;
```

Now that we have an appropriately-privileged Haiku Haven-specific user, we're done with the default `postgres` user, so exit out of the `psql` shell by typing `exit`, and then re-run it like so:

```
psql -U haiku_lord -d postgres
```

Assuming you entered the password you selected for `haiku_db` when prompted, you're again connected to the `postgres` database, but now you're acting under the capacity of `haiku_lord` (if you like, you can verify this by typing `SELECT current_user;`).

To create a database called `haikudb` to hold the tables for Haiku Haven, enter:

```
CREATE DATABASE haikudb;
```

You can list all the databases available in your local PostgreSQL instance by typing `\l`. You should now be able to see `haikudb` in there, along with `postgres` and a couple of others.

To start using our database, we need to "connect" to it. To do this, type:

```
\c haikudb;
```

A confirmation message—You are now connected to database "haikudb" as user "haiku_lord"—should let you know that this worked.

Keep this terminal window with the `psql` shell open. We're going to keep coming back to it throughout this chapter. If you do end up closing it, you can get back to this state by typing:

```
psql -U haiku_lord -d haikudb
```

8.2.4 A crash course in SQL

The SQL in PostgreSQL stands for Structured Query Language. Variously pronounced as "sequel" and "ess-cue-ell", SQL is the language of databases, used to create and update tables, and most importantly, to *query* them to get exactly the data we need. It's also one of the most popular and useful languages to know if you're a developer or work in the field of data in any capacity.

If you don't know SQL, today's your lucky day because we're going to breeze through the basics in this section.

CREATING TABLES

Previously in this chapter, we came up with Haiku Haven's database schema that contains two tables: `users` and `haikus`.

We're now going to actually create these tables in PostgreSQL. Go back to the `psql` shell where you're connected to the `haikudb` database (or re-run `psql` as indicated earlier), and enter the following SQL command:

```
CREATE TABLE users (
    username VARCHAR(100) PRIMARY KEY,
    password_hash VARCHAR(128)
);
```

The `CREATE TABLE` command creates a table with a specific schema. In the above, the name of the table is `users`, and it has two columns: `username` and `password_hash`—which, if you recall, are the two fields we settled on earlier as being important attributes of a user.

Why does it say `password_hash` instead of just `password`? Bear with me, please—I'll explain this a little later in the chapter. For now, let's just think of it as the user's password.

Turn your attention to the lines where we've defined the columns:

```
username VARCHAR(100) PRIMARY KEY,
password_hash VARCHAR(128)
```

These sequences, separated by commas, are *column specifications* that let you configure each column—including specifying the data type.

`VARCHAR` is one such data type in PostgreSQL; it's a string type that can have a varying number of characters. `VARCHAR(100)` means that the column can have any number of characters up to 100. If you try to store more than 100 characters, PostgreSQL will throw an error.

You'll also notice `PRIMARY KEY` against the `username` column. This denotes that the `username` column will be used to uniquely identify a row in the `users` table. Among other things, this means that every row in `users` must have a `username`, and that only one user can have a particular `username`.

To check that this worked, you can use the `psql` command `\dt`, which lists all the tables in the current database. Doing so now should give you:

List of relations			
Schema	Name	Type	Owner
public	users	table	haiku_lord
(1 row)			

Next, let's create our second table, `haikus`, with another `CREATE TABLE` command:

```
CREATE TABLE haikus (
    haiku_id SERIAL PRIMARY KEY,
    created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    author VARCHAR(100),
    text TEXT,
    FOREIGN KEY (author) REFERENCES users(username)
);
```

There are a few new things here:

- `haiku_id`, the primary key, has a data type of `SERIAL`. This means that PostgreSQL will automatically provide the value for this column as an auto-incrementing integer. The first row inserted will have `1` as its `haiku_id`, the second will have `2`, and so on.
- `created_at` has the data type `TIMESTAMPTZ`, which is a timestamp with time zone information. The `DEFAULT` keyword specifies the value to put in this column if one is not explicitly provided while inserting a row in the table. In this case, we want the current timestamp (`CURRENT_TIMESTAMP`) to be provided as this default value.
- `text` has a data type of `TEXT`, which is like `VARCHAR` but with no maximum character length. This makes sense for the actual content of the haiku.
- The last line, `FOREIGN KEY (author) REFERENCES users(username)`, says that the `author` column must have a value that exists in the `username` column of some row in the `users` table. This is called a *foreign key constraint*.

INSERTING ROWS INTO TABLES

Next, let's populate these tables with rows using SQL's `INSERT INTO` statement.

We'll first create a new user by adding a row to the `users` table:

```
INSERT INTO users (username, password_hash) VALUES ('alice', 'Pass_word&34');
```

The `INSERT INTO` command inserts a row into the table by specifying the column values. The `(username, password_hash)` after the table name `users` says that the column values we're about to specify correspond to the columns `username` and `password_hash`, in that order.

The part after `VALUES` gives the actual values to enter. The effect here is that `users` now has one row—with a `username` of `alice` and `password_hash` set to `Pass_word&34`.

```
INSERT INTO haikus (author, text) VALUES
('alice', E'Two foxes leap high\nOne lands safely on the earth\nWhere is the other?');
```

Note here that though `haikus` has four columns, we're only specifying the values for two of them—`author` and `text`. PostgreSQL automatically provides the `haiku_id` as well as a default value for `created_at`.

Also notice the `E` before the starting quote of the haiku. This marks the value as an *escape string*, correctly translating the `\n` escape sequence into a newline character.

Before we query these tables, let's add another unsettling haiku under `alice` for good measure:

```
INSERT INTO haikus (author, text) VALUES
('alice', E'Five frogs are jumping\nFour come down as expected\nBut one goes missing.'');
```

QUERYING TABLES

The real strength of SQL lies in the versatile ways in which we can *query* tables (read data from them) once they have been populated with rows.

To retrieve data, we use the `SELECT` statement. This command allows us to specify the columns we want to see and filter rows based on certain criteria. Let's start with a simple query to fetch all the rows and columns from the `users` table:

```
SELECT * FROM users;
```

The `*` indicates that we want to retrieve all columns from the table. As a result of this, we'll see a list of all users along with their associated `user_id` and `password_hash`. We only currently have one user, so we get:

```
username | password_hash
-----+-----
alice   | Pass_word&34
(1 row)
```

NOTE If you're horrified to see passwords so easily queried, don't worry! As we'll soon see, we're not actually going to be storing passwords like this.

We could also choose to only retrieve certain columns from the table. For example, if we only want the `haiku_id`, `created_at`, and `author` fields from the `haikus` table, we could write:

```
SELECT haiku_id, created_at, author FROM haikus;
```

This would give us:

```
haiku_id | created_at | author
-----+-----+-----
1 | 2024-12-10 16:12:11.71654-08 | alice
2 | 2024-12-10 16:12:16.364669-08 | alice
(2 rows)
```

We can also filter the rows we want to see based on a condition (or set of conditions) using a `WHERE` clause. For example, if we want to see the `haiku_id` and `text` for only haikus that are written by `alice` and contain "`fox`", we could use the following:

```
SELECT haiku_id, text FROM haikus WHERE author = 'alice' AND text LIKE '%fox%';
```

to get:

```
haiku_id | text
-----+-----
1 | Two foxes leap high +
| One lands safely on the earth+
| Where is the other?
(1 row)
```

We've filtered the `haikus` table using two conditions separated by the keyword `AND` here: `author = 'alice'`, and `text LIKE '%fox%'`.

The `LIKE` keyword is used to perform text-matching. The `%` symbol means "zero or more characters," making `%fox%` a pattern that matches any text containing the word 'fox' anywhere within it.

The `SELECT` statement can do a *lot* more than this, such as calculating summary statistics, aggregating rows, fetching data from multiple tables by joining them and more, but these are out of the scope of this chapter.

UPDATING TABLES

You can update rows in a table after you've inserted them, using the `UPDATE` statement. For example, let's say you want to add the string "[By Alice] " to the start of the haiku with `haiku_id` 1:

```
UPDATE haikus SET text = '[By Alice] ' || text WHERE haiku_id = 1;
```

Here we're setting the column `text` in the `haikus` table for rows where the `haiku_id` is 1. The `||` operator is used in SQL to concatenate strings, so `SET text = '[By Alice] ' || text` adds "[By Alice]" to the beginning. If we now `SELECT` the text for that haiku with:

```
SELECT text FROM haikus WHERE haiku_id = 1;
```

we'll get:

text
[By Alice] Two foxes leap high+ One lands safely on the earth + Where is the other? (1 row)

Though we've only updated one row here, `UPDATE` will update all the rows that match the `WHERE` clause. If we omit the `WHERE` clause, the `UPDATE` will apply to every single row in the table.

DELETING ROWS (AND TABLES)

To delete rows from a table, we use the `DELETE FROM` command. Just as in the case of `UPDATE`, it uses a `WHERE` condition to determine the rows to delete.

So we could delete the haiku with ID 2 by running:

```
DELETE FROM haikus WHERE haiku_id = 2;
```

Be careful with this command! If you omit the `WHERE` clause, `DELETE FROM` will delete *every single row!*

Finally, we can delete a table itself (as opposed to just the rows in the table), by using the `DROP TABLE` command.

For instance, if we wanted to remove the `haikus` table (which we don't, for now), we could have entered:

```
DROP TABLE haikus;
```

Of course, there's much more to SQL than what we've learned—indeed, entire careers can be forged by knowing it well enough—but this is all we'll need for this chapter.

8.2.5 Connecting PostgreSQL to our app

We've now set up the database for Haiku Haven. We've also learned to insert rows and query our database *manually*. What's left is to enable our app to query and make changes to the database *programmatically*, through Python.

For this, we'll employ a third-party Python module named `psycopg2`, which provides us with a way to talk to PostgreSQL in our app.

Go ahead and install it now by running `pip install psycopg2` in a new terminal window.

HOW DATABASE CONNECTIONS WORK

Before using `psycopg2`, let's understand how PostgreSQL queries work in Python.

We learned in prior chapters that a Streamlit app is served by a Streamlit server process that listens to a particular port (usually 8501 or something near that) on a machine. Similarly, a PostgreSQL database runs on a PostgreSQL server that listens to a different port—5432 by default, unless you set a different port number when you installed it.

To execute SQL commands, the app must establish a *connection* to the PostgreSQL server. A single connection can handle only one command at a time.

But what if multiple users try to create or update a haiku at once? We could queue up the SQL commands on a single connection, but that could cause delays. Using multiple *connections* is better, but setting up each one can be resource-intensive—especially in production—as it involves network initialization and authentication.

`psycopg2`'s answer to this is a *connection pool* that manages multiple *re-usable* database connections efficiently, illustrated by figure 8.5.

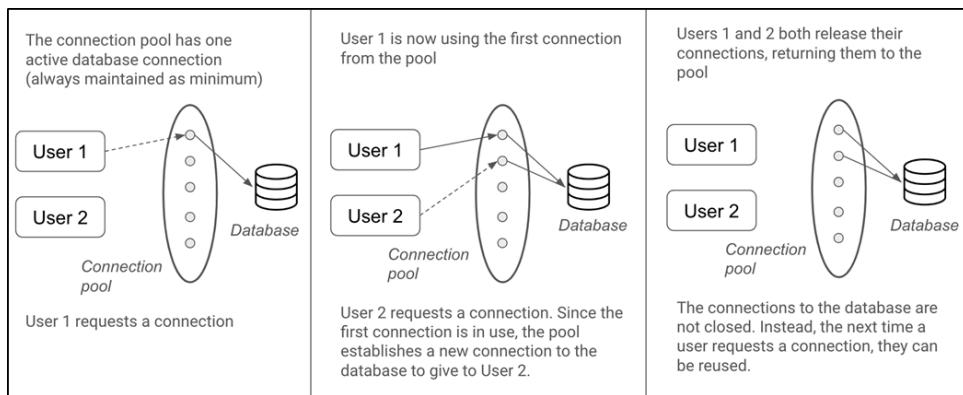


Figure 8.5 Database connection pool with 1 minimum and 5 maximum connections

A connection pool maintains a set of connections that can be quickly reused by different parts of the application or different users accessing the application. When a part of the application needs to run a database query, it can:

- Request a connection from the pool
- Use the connection to execute the query
- Return the connection to the pool for reuse

USING PSYCOPG2 TO CONNECT TO OUR DATABASE

With all that in mind, let's try setting up a live database connection and running a query! Open a Python shell by typing `python` (or `python3`) into your terminal.

Once you're in, import what we need from the module:

```
>>> from psycopg2.pool import ThreadedConnectionPool
```

`ThreadedConnectionPool` is the connection pool class we'll use. To instantiate it, we'll need the address where our PostgreSQL server is running, the port number it's running on, the username and password of the PostgreSQL user we created, and the name of our database.

We can combine all of these into a *connection string* that takes the form:

```
postgresql://<PostgreSQL username>:<Password>@<Address of server>:<Port number>/<Database name>
```

In our case, we'll type:

```
>>> connection_string = 'postgresql://haiku_lord:<password>@localhost:5432/haikudb'
```

Obviously, you'll need to replace `<password>` with the actual password you created for `haiku_lord`. Since we're running PostgreSQL locally right now, the server address is simply `localhost`, but this will change when we switch to a managed PostgreSQL service during deployment later in the chapter.

We can now create our connection pool:

```
>>> connection_pool = ThreadedConnectionPool(1, 5, connection_string)
```

1 and 5 here are the minimum and maximum connections in the pool. This means 1 connection is always kept available even before we receive any requests, and a maximum of 5 simultaneous connections can be maintained.

Let's grab a connection from the pool using the `getconn` method:

```
>>> connection = connection_pool.getconn()
```

To execute a query, we need a `cursor`, which is a pointer that allows us to execute SQL commands and fetch results from the database in flexible ways. We create one by calling the `cursor()` method on our `connection` object:

```
>>> cursor = connection.cursor()
```

Let's now write the query itself. We'll try to fetch the `haiku_id` and `author` fields for a given author:

```
>>> query = 'SELECT haiku_id, author FROM haikus WHERE author = %s'
```

Note the `%s` here. This makes the username of the author a parameter of the query. By plugging in different usernames, we could get the haikus of different authors. For now, we want `alice`'s haikus, so we'll create a `params` tuple to pass along with the query.

```
>>> params = ('alice',)
```

`('alice',)` is the literal notation for a single-element tuple containing the string '`alice`'.

```
>>> cursor.execute(query, params)
```

Once the execution is done, let's fetch all the results.

```
>>> cursor.fetchall()
[(2, 'alice'), (1, 'alice')]
```

The results are in the form of a list of tuples, where each tuple represents a single row in the database. Since our `SELECT` clause said `haiku_id`, `author`, the first element of each result tuple is the `haiku_id`, while the second is the `author` field.

Since we're done with the query, let's return the connection to the pool so other parts of the app can use it:

```
>>> connection_pool.putconn(connection)
```

That concludes our illustration of `psycopg2`. As a final step, let's clean up the connection pool:

```
>>> connection_pool.closeall()
```

This closes all connections and no more can be requested from the pool.

CREATING A DATABASE CLASS

Now that we know how to run SQL queries in Python, we're ready to begin writing the code for our app.

In this chapter, we're going to organize our code in two folders: `backend` and `frontend`, and a main entrypoint script (the one we'll use with `streamlit run`) that lies outside of either.

The database connection stuff we just covered is pretty technical and we'd rather not have to deal with it in the remainder of our app's code. It would be nice if we had a database object that we could just ask to execute the queries we need, without having to worry about the gory details of connection pools and cursors. Whenever we want to run a particular query, we should be able to write `database.execute_query(query, params)`, passing in the query we want to execute and the parameters we want to give it.

To set this up, let's create a `Database` class.

Create a new Python file called `database.py` within the `backend` folder and copy the code in listing 8.1.

Listing 8.1 backend/database.py

```
from psycopg2.pool import ThreadedConnectionPool

MIN_CONNECTIONS = 1
MAX_CONNECTIONS = 10

class Database:
    def __init__(self, connection_string):
        self.connection_pool = ThreadedConnectionPool(
            MIN_CONNECTIONS, MAX_CONNECTIONS, connection_string
```

```

    )

def connect(self):
    return self.connection_pool.getconn()

def close(self, connection):
    self.connection_pool.putconn(connection)

def close_all(self):
    print("Closing all connections...")
    self.connection_pool.closeall()

def execute_query(self, query, params=()):
    connection = self.connect()
    try:
        cursor = connection.cursor()
        cursor.execute(query, params)
        results = cursor.fetchall()
        connection.commit()
        return results
    except Exception:
        connection.rollback()
        raise
    finally:
        self.close(connection)

```

By this point, you should be used to the simple dataclasses we've used in prior chapters. Dataclasses simplify the syntax used to create classes in Python, but for more complex use cases, we need to peel back that layer and write a traditional class definition.

A class is essentially a blueprint that can be turned into a concrete Python object, which can have attributes—these are the object's properties or data associated with the object—and methods—or functions that define the object's behavior and can interact with its attributes.

Let's study the class definition for our `Database` class, shown in listing 8.1. We'll start with the `__init__` method:

```

def __init__(self, connection_string):
    self.connection_pool = ThreadedConnectionPool(
        MIN_CONNECTIONS, MAX_CONNECTIONS, connection_string
    )

```

`__init__` (pronounced "dunder init") is a special method in Python. When an object is first created from a class, the `__init__` method is executed automatically. The two arguments our `__init__` takes are `self` and `connection_string`.

You'll see `self` extremely often in Python class definitions. The first argument to a method within a class is a special one that's not explicitly passed while calling it (as we'll see in a bit). It always refers to the object on which the method is being called. By convention, this argument is named `self`—though you can technically call it whatever you like.

`connection_string` is meant to hold a string of the type we formed earlier, containing the database configuration.

The line `self.connection_pool = ThreadedConnectionPool(MIN_CONNECTIONS, MAX_CONNECTIONS, connection_string)` creates a `psycopg2 ThreadedConnectionPool` as we demonstrated in the earlier section, and assigns it to `self.connection_pool`, making it an attribute of the object created from the class.

`MIN_CONNECTIONS` and `MAX_CONNECTIONS` are defined at the top of the file where they can be easily configured or changed later.

We're effectively initializing a connection pool as soon as the database object is created.

We also define methods other than `__init__`, such as `connect`, which obtains a connection from the pool, `close`, which returns a given connection to the pool, and `close_all`, which closes the connection pool itself (and prints a message to the terminal window). The code for these should be familiar as we discussed it when we were trying out `psycopg2`.

However, we generally won't call these methods directly. Recall that what we really want is a simple `execute_query` method that'll take care of all the underlying database connection logic. Let's presently turn our attention to that method:

```
def execute_query(self, query, params=()):
    connection = self.connect()
    try:
        cursor = connection.cursor()
        cursor.execute(query, params)
        results = cursor.fetchall()
        connection.commit()
        return results
    except Exception:
        connection.rollback()
        raise
    finally:
        self.close(connection)
```

`execute_query` naturally accepts a SQL query we want to execute and any params (set to an empty tuple, `()`, by default) we want to give the query.

The body of the method starts by obtaining a connection from the pool. We then see a `try-finally` block.

In Python, `try-except-finally` is a construct that's used for error handling. The idea is to write your regular code in the `try` block. If an exception occurs while running the `try` code, Python stops execution and jumps to an `except` block that "catches" the exception, allowing you to log a sensible error message, for example, or use some other kind of handling logic.

Regardless of whether there's an exception or not, the code in the `finally` block is always executed.

The code in the `try` block simply gets a cursor, executes the query and returns the results, just as we saw when we were exploring `psycopg2`. The only new thing you might notice is the line `connection.commit()`.

This line exists because we're not just going to be using `execute_query` to run `SELECT` queries. We're also going to run `INSERT INTO`, `UPDATE`, and `DELETE FROM` commands—all of which *modify* tables, not just *read* from them.

In PostgreSQL, when you modify data, the changes aren't permanently saved until you `commit` them.

What if some kind of error occurs (for instance, the given query might have incorrect SQL syntax) while Python's trying to execute this code? We enter the `except` block marked by `except Exception`, which contains the line `connection.rollback()`. This undoes any temporary changes made so that the database is left in a pristine state.

We also print the error we encountered and then re-raise the exception, letting the regular exception flow take over—such as printing the exception message to the screen. What we've achieved with the `try-except` construct is to inject ourselves into the flow when an exception occurs and make sure any partial changes are rolled back.

What about the `finally` block? Well, that calls the `close` method further up in the file, returning the connection to the pool. Regardless of what happens in the `try` block, the connection is always freed up. If we didn't have this code, the connection would remain allocated even after the function exits, and eventually the pool would run out of new connections. Putting this code in the `finally` block ensures the connection is returned even if an error occurs.

USING THE DATABASE CLASS IN OUR APP

With a `Database` class set up, all that's left to do to enable persistent storage and retrieval in our app is to actually use the class in our Streamlit app.

Creating an instance of the `Database` class requires providing it with a connection string. Connection strings are sensitive because they contain PostgreSQL credentials, so we don't want to be putting the string in our code.

You probably know where this is heading—we need to use `st.secrets` as we've done in the past! Without further ado, go ahead and create a `.streamlit` folder in your app's root folder (the parent folder where you created the `backend` directory), and create a `secrets.toml` within it, with contents similar to what's shown in listing 8.2.

Listing 8.2 .streamlit/secrets.toml

```
[config]
connection_string = "postgresql://haiku_lord:password@localhost:5432/haikudb"
```

Don't forget to replace `password` with your actual password. Also, as we learned before, you shouldn't check this file into Git.

Next, create our app's entrypoint file—say, `main.py`—in the app's root directory, with the code in listing 8.3.

Listing 8.3 main.py

```
import streamlit as st
from backend.database import Database

st.title('Haiku Haven')

connection_string = st.secrets['config']['connection_string']
database = Database(connection_string)
query_results = database.execute_query('SELECT * FROM haikus')
st.write(query_results)
```

This is where it all comes together! Firstly, we import the `Database` class we just created using:

```
from backend.database import Database
```

This line means "import the `Database` class from `backend/database.py`". Notice how, in module import paths, the path separator becomes a dot.

For this line to work, Python needs to recognize the parent folder of `backend` as a starting point from which it can look for modules. We can do this by adding the path to the parent folder of `backend` to `sys.path`, which is the list of paths that determines this in Python.

Fortunately, when you run the `streamlit run <script.py>` command, the parent folder of `<script.py>` is automatically added to `sys.path`. In this case, since the parent folder of `main.py` is also the parent folder of the `backend` directory, we don't need to do anything extra.

Next, after displaying a title, we extract the connection string from `st.secrets` as we've done in prior chapters:

```
connection_string = st.secrets['config']['connection_string']
```

We use this to create an instance of the `Database` class:

```
database = Database(connection_string)
```

You'll probably recognize this syntax from when we've used dataclasses before, but this is the first time in this book that we've instantiated a traditional class we wrote, so a deeper explanation is warranted.

What's going on here is that by passing `connection_string` to the class `Database` as though it were a function, we're actually passing it to `Database`'s `__init__` method, which has the signature line `def __init__(self, connection_string)`. As I mentioned earlier, `self` is automatically set to the object that's being created, so we don't need to pass it explicitly.

Instead, we pass only `connection_string`, thus letting `__init__` set up the connection pool for us, and getting the resulting `Database` instance in the `database` variable (note that `__init__` will return the instance even though we didn't write an explicit `return` statement).

Once we have the instance `database`, we use it to execute a simple `SELECT` query:

```
query_results = database.execute_query('SELECT * FROM haikus')
```

Again, even though `execute_query` takes `self` as its first argument, we don't need to pass it explicitly. Instead `database` itself is passed to `self`.

This query has no parameters, so we let the `params` argument have the default value (an empty tuple) by not specifying it.

Finally, we use `st.write` to display the results of the query:

```
st.write(query_results)
```

Take a look at the page now by typing `streamlit run main.py` (make sure you're in `main.py`'s containing folder first so that Streamlit can find the `.streamlit` folder). You should see something similar to figure 8.6.

Haiku Haven

```

[{"id": 0, "author": "alice", "text": "Five frogs are jumping\nFour come down as expected\nBut one goes missing.", "date": "2024-12-10T16:12:16Z"}, {"id": 1, "author": null, "text": "A small red bird perched on a branch,\nSinging a joyful tune, filling the air.\nNature's melody.", "date": "2024-12-10T16:12:11Z"}]
  
```

Figure 8.6 The contents of the haiku table as read by executing a `SELECT` query using the `Database` class and displayed using `st.write` (see [chapter_8/in_progress_01](#) in the GitHub repo for the full code)

One interesting thing here is how `st.write` formats the list of tuples within `query_results` in an easy-to-read form.

In any case, our app is now connected to a database! Next up, let's use this to allow users to create accounts!

8.3 Creating user accounts

Since our users can create their own haikus, there needs to be a way for them to create Haiku Haven accounts to hold their haikus. In this section, we'll wire up our app to enable this, taking care to store passwords securely.

Before we get to that though, let's take a minute to talk about code organization.

8.3.1 Splitting our app into services

In Chapter 3, we discussed the principle of *separation of concerns*—the idea that each component of our app should focus on a specific thing and be independent of the other components, interacting with them only in ways that are specified by a contract or API.

We'll do something similar here, separating the frontend and backend as we did in that chapter. Since we're using classes and *object-oriented programming* this time, we could define a backend class—let's call this `Hub`—that can be the single point of contact for frontend code to call backend code. Any function that the frontend can call should be a method in the `Hub` class. This is analogous to `backend.py` in Chapter 3 where every backend function called from the frontend code was defined in `backend.py`.

We might expect the `Hub` class to have methods that fulfill actions a frontend user might want to take, such as `create_user`, `create_haiku`, `update_haiku`, etc. Over time, though, as our app grows more and more complex, the `Hub` class would have an increasing number of methods, slowly making it unwieldy and difficult to manage.

Rather than take this *monolithic* approach, it might be a better idea to divide up the actions offered by `Hub` into individual *service classes*, each pertaining to a specific kind of action, and using the `Hub` class as merely a coordinator. For instance, we could have a `UserService` class that offers methods pertaining to users, such as `create_user`, and a `HaikuService` class that offers those related to haikus, such as `create_haiku` and `update_haiku`.

This would make our app more modular and easier to extend and maintain. We can add more user-related functionality to `UserService` independently of the haiku-related actions in `HaikuService`, and if we wanted to add limericks to our app later, we could introduce a `LimerickService` without touching either.

With our overall code organization strategy in mind, let's turn our attention towards building one of the components—the user service.

8.3.2 Creating the user service

Just as in prior chapters, we'll use a dataclass to represent the fundamental objects we're concerned with. In Chapters 3, 4, and 6, we had `Unit`, `Task`, and `Metric` classes. Here, we'll start with a `User` class.

Create a new Python file called `user.py` in the `backend` folder, with the text shown in listing 8.4.

Listing 8.4 `backend/user.py`

```
from dataclasses import dataclass

@dataclass
class User:
    username: str
    password_hash: str
```

You'll see that this directly mirrors the `users` table in our database, with fields for `username` and `password_hash`.

Let's now talk about the latter field, and why we called it `password_hash` instead of `password`.

STORING PASSWORDS SECURELY

Passwords are naturally some of the most sensitive pieces of information software developers need to deal with, and much of the field of cybersecurity focuses on keeping them secret.

We know that we shouldn't store passwords in our code, resorting to the construct of `st.secrets` to avoid this. But what about storing them in a database?

Obviously, our app needs to be able to compare a user-entered password with the one actually associated with the user, so passwords need to be stored in some form. However, storing them directly as plain text in a database introduces a security vulnerability.

That's because anyone who gains access to our database—through some kind of security breach—will be able to see passwords in their raw form. How do we avoid this though?

The answer is: with *one-way cryptographic hash functions*, or in other words, a bit of fancy math. It turns out that there are certain mathematical operations that are easy to perform normally but extremely difficult to perform in reverse. As a trivial example, consider multiplying two prime numbers a and b to get c . Multiplying a and b to get c is easy, but if you're only given c , identifying a and b is difficult, especially when c is very, very large (think hundreds of digits long).

Similarly, you can think of a cryptographic hash function as an operation performed on a password that's very difficult to reverse.

Let's say someone's password is `SomePassword123`. If you apply a hash function H to it, you might get a password hash that looks like a random sequence of characters:

Rather than store the string `SomePassword123` directly in the database, we store `g53jkd1gfee09ded8d33rr45t5y5y43f2eff`. Then when someone enters a password in our app, we apply the hash function to *that* password and compare the result to `g53jkd1gfee09ded8d33rr45t5y5y43f2eff`. If the two are the same, the user is authenticated.

How does this help with security? Well, if a hacker now manages to get into our database, they don't have the actual password, only the password hash. As stated, it's very difficult to obtain the password from the password hash.

The password hash itself is useless to the hacker as there's no point entering it in the app—if you did, the app would just apply the hash function on it and come up with a completely different hash that's compared against the real one.

How do we implement this in our app? Fortunately, we don't have to do it from scratch. There are third-party libraries that do it for us. We'll use `bcrypt`, which you should install now with `pip install bcrypt`.

Let's add two more methods to our `User` class so it now looks like this:

```

from dataclasses import dataclass
import bcrypt

@dataclass
class User:
    username: str
    password_hash: str

    @staticmethod
    def hash_password(password):
        return bcrypt.hashpw(password.encode(), bcrypt.gensalt()).decode()

    def authenticate(self, password):
        return bcrypt.checkpw(password.encode(), self.password_hash.encode())

```

We've decorated the `hash_password` method with `@staticmethod`. This makes it belong to the class itself rather than to any specific instance of the class. We generally use `@staticmethod` for utility functions that are logically related to a class but don't actually need to access anything from a particular instance.

`hash_password` is a good fit for this since it doesn't need to access any of the instance's attributes or methods (note the absence of a `self` parameter). Rather, it simply accepts a password entered by a user, converts it into a hash using `bcrypt`, and returns it.

I won't go into the details of how this works, but at a high level, we're taking an extra measure of security here by adding a random "salt" to the password (`bcrypt.gensalt()`) before hashing it. This salt helps protect the password against hackers simply looking up the password associated with a password hash from a huge pre-computed table of such hashes (called a *rainbow table*).

We also have an `authenticate` method that we'll use when a user enters a password. `bcrypt.checkpw` compares the entered password (`password.encode()`) and the password hash stored in the `User` object (`self.password_hash.encode()`), returning `True` if they match.

THE USERSERVICE CLASS

We're now ready to create the `UserService` class, the one we determined would have methods for user-related operations. Create a `user_service.py` file under `backend/` with the content in listing 8.5.

Listing 8.5 backend/user_service.py

```

from backend.user import User

class UserService:
    def __init__(self, database):
        self.database = database

    def get_user(self, username):
        query = "SELECT username, password_hash FROM users WHERE username = %s"
        params = (username,)
        results = self.database.execute_query(query, params)
        return User(*results[0]) if results else None

    def create_user(self, username, password):
        existing_user = self.get_user(username)
        if not existing_user:
            query = '''
                INSERT INTO users (username, password_hash)
                VALUES (%s, %s)
                RETURNING username, password_hash
            '''
            password_hash = User.hash_password(password)
            params = (username, password_hash)
            results = self.database.execute_query(query, params)
            return User(*results[0]) if results else None
        return None

```

UserService has a `__init__` method that accepts an instance of the Database class we created earlier and assigns it to a `database` attribute (`self.database`) of the object.

When a user enters a username and password to create a user, we first need to check if a user of that username already exists, so we have a `get_user` method that returns the user if one exists or `None` if it doesn't.

The `get_user` method executes a parameterized SQL query (`SELECT username, password_hash FROM users WHERE username = %s`) on the database, passing the given username as the only parameter (`(username,)`).

As we've seen before, this returns a list of tuples. Due to the `SELECT username, password`, these will be of the form `(<username>, <password_hash>)`. Consider the last line in `get_users`:

```
return User(*results[0]) if results else None
```

If there's no user with the given username, results will be an empty list, so it'll evaluate to `False`, causing `get_user` to return `None`.

If there *is* such a user, `results[0]` will be a tuple of the form `(<username>, <password_hash>)`. In Python, the `*` operator when applied to a tuple (or a list), *destructures* it for use in things like function calls.

So `User(*results[0])` is equivalent to `User(<username>, <password_hash>)` which creates a new instance of the `User` dataclass (which you'll recall has two corresponding members: `username` and `password_hash`).

The `create_user` method first uses `self.get_user(username)` to see if a user already exists with the given username. If it does, it simply returns `None`.

If it doesn't, it issues the following query to the database:

```
INSERT INTO users (username, password_hash)
VALUES (%s, %s)
RETURNING username, password_hash
```

This is an `INSERT` query, which we've seen previously. The only new thing here is the line `RETURNING username, password_hash`. An `INSERT` query doesn't generally need to return any results, as it's a modify operation, not a read operation.

Adding the `RETURNING` clause makes it return the specified fields in the same way that a `SELECT` query would. In this case, the `username` and `password_hash` of the newly created row are returned.

Once again, `create_user` uses the same approach as `get_user(User(*results[0]))` to create and return a `User` object if everything is successful.

THE HUB CLASS

When we spoke of code organization earlier, we mentioned the `Hub` class, which would be the single point that our frontend code would access. Let's write that class now. Create `backend/hub.py` with the code from listing 8.6.

Listing 8.6 backend/hub.py

```
from backend.database import Database
from backend.user_service import UserService

class Hub:
    def __init__(self, config):
        database = Database(config['connection_string'])
        self.user_service = UserService(database)
```

The `Hub` class's `__init__` is quite simple: it accepts a `config` object (a dictionary of configuration options, such as the one obtained by parsing the `secrets.toml` file we created earlier), creates a `Database` object, and passes it to `UserService` to create an instance of that class.

`Hub` has no other methods. This makes sense because, as we've emphasized, `Hub` is simply a coordinator class that our frontend can use to access the various service class objects (of which `user_service`, an instance of `UserService`, is the only one we've created so far).

THE SIGNUP PAGE

Working our way from the bottom up, we've created a `User` class, a `UserService` class that accesses the `User` class, and a `Hub` class that accesses the `UserService` class—which currently only has a `create_user` method.

The part of our Streamlit app that accesses `create_user` will be the signup page, which we'll define—for now—in `main.py`.

Our previous `main.py` initialized the `Database` object directly and executed a sample query. Since the database is now initialized in the `Hub` class, we'll rewrite `main.py` entirely, as shown in listing 8.7.

Listing 8.7 main.py revised

```
import streamlit as st
from backend.hub import Hub

hub = Hub(st.secrets['config'])

with st.container(border=True):
    st.title("Sign up")
    username = st.text_input("Username")
    password = st.text_input("Password", type="password")
    confirm_password = st.text_input("Confirm password", type="password")

    if st.button("Create account", type="primary"):
        if password != confirm_password:
            st.error("Passwords do not match")
        else:
            user = hub.user_service.create_user(username, password)
            if user:
                st.success("Account created successfully")
            else:
                st.error("Username already exists")
```

By this point in the book, you should be able to read the code in listing 8.7 fairly easily.

It starts by creating an instance of `Hub` using the `'config'` entry of `st.secrets` (note here that `Hub` accepts the entire `config` object rather than just the connection string in case there are other configurations that need to be considered).

Next, it displays the usual username-password-confirm password set of inputs I'm sure you've seen in various websites before. On clicking the "Create account" button, if the passwords in the two inputs don't match, an error is displayed.

If they do match, we call the `create_user` method defined in the `UserService` class to create the user in the database:

```
user = hub.user_service.create_user(username, password)
```

We then show a success or error message based on the return value.

At this point, you should be able to see figure 8.7 if you rerun the app.

The screenshot shows a "Sign up" form. It has three input fields: "Username" (filled with "bob"), "Password" (filled with "*****" and an eye icon), and "Confirm password" (filled with "*****" and an eye icon). A red "Create account" button is below the inputs. At the bottom, a green box displays the message "Account created successfully".

Figure 8.7 Haiku Haven's signup page (see `chapter_8/in_progress_02` in the GitHub repo for the full code)

Try creating an account with the username `bob`. To verify that it worked and that a user has indeed been created, you can issue the query `SELECT * from users where username = 'bob'` in your `psql` prompt (which you've hopefully kept open). This should give you something like:

username	password_hash
bob	<code>\$2b\$12\$hVzjZJN7QMTM94H7ZL.ZJe3PFFgyAPgDOH1F2b38IovcuvKrNAu3G</code>
(1 row)	

As you can see, due to the hashing, `bob`'s password is no longer directly visible. But can `bob` log in? Not until we've completed the next part!

8.4 Setting up a multi-page login flow

We saw earlier—when we sketched out the user experience—that Haiku Haven is meant to be a multi-page app with different pages for signup, login, and the haiku-related functionality, something we haven't encountered before.

8.4.1 Multi-page apps in Streamlit

Streamlit has built-in support for multi-page apps. In this scheme, you define your individual pages separately, and let your entrypoint file (the one you run with `streamlit run`) act as a "router" that identifies the page to load and runs it.

The entrypoint file is quite crucial here; it's loaded as usual in every re-run, and it's the one that picks the "current page" to load.

Let's see an example. So far in our app, we've created a signup page for users to create their accounts, but not a login page. Once we do create the login page, there needs to be a way to tie the two together, making them part of the same app.

We'll do that by revising the `main.py` file one more time, using the multi-page approach discussed above. The signup flow we included in `main.py` will have to move to a different file (`frontend/signup.py`). The new `main.py` is shown in listing 8.8.

Listing 8.8 main.py revised (yet again)

```

import streamlit as st
from backend.hub import Hub

pages = {
    "login": st.Page("frontend/login.py", title="Log in",
                     icon=":material/login:"),
    "signup": st.Page("frontend/signup.py", title="Sign up",
                      icon=":material/person_add:"),
}

if 'hub' not in st.session_state:
    config = st.secrets['config']
    st.session_state.hub = Hub(config)

page = st.navigation([pages['login'], pages['signup']])
page.run()

```

The first new thing you'll notice here is the `pages` dictionary. The keys in `pages` are `"login"` and `"signup"`, which are the names of the pages we want in our app. The values are `st.Page` objects. Let's inspect the first one:

```
st.Page("frontend/login.py", title="Log in", icon=":material/login:")
```

`st.Page` is Streamlit's way of defining a single page in a multi-page app. The first argument you pass it is the path to the Python script for that page—in this case, `frontend/login.py`, which doesn't exist yet. We've also passed it a sensible title.

The last argument is an icon for the page. It has a curious value: `:material/login:.`

This demonstrates a neat way to display icons in Streamlit. The syntax `:material/<icon_name>:` is accepted by most widgets that accept displayable text, and is converted to an image when rendered to the screen.

You can see the supported icons in Google's Material Symbols library at <https://fonts.google.com/icons?icon.set=Material+Symbols>. In this case, we've chosen the "Login" icon. Whenever you need to show an icon, you can go to that URL, click the icon you want, identify its icon name from the sidebar that opens to the right, and substitute it within the text `:material/<icon_name>:.`

Now turn your focus to the following lines further below:

```
page = st.navigation([pages['login'], pages['signup']])
page.run()
```

Here, we're feeding the two `st.Page` objects in the `pages` dictionary (`pages['login']` and `pages['signup']`) to `st.navigation`, a new Streamlit widget.

`st.navigation` is used to configure the available pages in a multi-page Streamlit app, displaying a navigation bar that users can use to select the page they want to go to.

It accepts a list of `Page` objects that form the navigation options, and returns a single `Page` object from the list. This returned item is the page selected by the user, or the first item in the list if nothing has been selected yet.

Once a page has been returned, it can be loaded using its `.run()` method.

You'll also see that we're saving the `Hub` instance (`hub`) to `st.session_state`, but not doing anything else with it. This is because the session state is shared between the pages in a multi-page app. So if you save something to `st.session_state` in any page, it will be accessible in the other pages too. In this case, we're going to be using the saved `hub` object in the other pages.

What about the signup flow we had earlier? Well, now that our app is multi-page, we'll move it to its own page, `signup.py`, within a new folder called `frontend`. As you'll see from listing 8.9, the content has mostly just been copied directly from our earlier `main.py` with no changes.

Listing 8.9 `frontend/signup.py`

```
import streamlit as st

hub = st.session_state.hub

with st.container(border=True):
    st.title("Sign up")
    username = st.text_input("Username")
    password = st.text_input("Password", type="password")
    confirm_password = st.text_input("Confirm password", type="password")

    if st.button("Create account", type="primary"):
        if password != confirm_password:
            st.error("Passwords do not match")
        else:
            user = hub.user_service.create_user(username, password)
            if user:
                st.success("Account created successfully")
            else:
                st.error("Username already exists")
```

The only change we've made (as compared to listing 8.7) is that we obtain the value of the `hub` variable from `st.session_state` where we saved it in the new `main.py`.

8.4.2 Implementing login

With our multi-page app infrastructure in place, it's time to build out the login feature. Before we set up the login page, let's make sure our backend has the functionality we need.

AUTHENTICATING A USER IN USERSERVICE

As we've discussed, all user-related functionality needs to live in `UserService`. Currently, that class has `create_user` and `get_user` methods. We'll implement a new `get_authenticated_user` method:

```
from backend.user import User

class UserService:
    ...

    def get_user(self, username):
        ...

    ...

    def get_authenticated_user(self, username, password):
        user = self.get_user(username)
        if user and user.authenticate(password):
            return user
        return None
```

`get_authenticated_user` accepts a `username` and `password` as arguments. It first calls the `get_user` method we defined earlier to see if a user exists with that `username`. If there is, it calls the `authenticate` method on the returned `User` object. Recall that the `authenticate` method in the `User` class compares the hash of the given `password` with that of the actual `password`.

If the authentication succeeds, the `User` object is returned. If it doesn't, the method returns `None`, which the calling code can interpret in two ways: either no such user exists or the `password` is incorrect. To keep things simple, we won't distinguish between these in the return value.

CREATING THE LOGIN PAGE

That's all we need in `UserService`. We can now go ahead and create a login page to complement the signup page we made earlier.

Create a new file in `frontend/` called `login.py`, with the content shown in listing 8.10.

Listing 8.10 frontend/login.py

```
import streamlit as st

hub = st.session_state.hub

with st.container(border=True):
    st.title("Log in")
    username = st.text_input("Username", key="login_username")
    password = st.text_input("Password", type="password")

    if st.button("Log in", type="primary"):
        user = hub.user_service.get_authenticated_user(username, password)
        if user:
            st.session_state.logged_in = True
            st.session_state.user = user
            st.success("Logged in successfully")
        else:
            st.error("Invalid username or password")
```

This page is fairly similar to `signup.py`, and should be straightforward to follow with your current understanding of Streamlit.

The part to focus on here is what happens when the "Log in" button is clicked. We first call the authentication method we defined in `UserService`:

```
user = hub.user_service.get_authenticated_user(username, password)
```

As we saw, if the method returns a `User` object, authentication has succeeded; if it returns `None`, authentication has failed.

We write this condition as `if user::`. For the actual logging in, we'll use a very simple approach—storing a boolean variable called `logged_in` under `st.session_state`, along with the `User` object for the logged-in user (named simply as `user`).

We also display a success or error message depending on whether the login succeeds.

At this point, you should re-run your app using `streamlit run main.py`. Try logging in with the account you created previously. You should see something similar to figure 8.8.

The screenshot shows a mobile-style login interface. On the left, there is a sidebar with a "Log in" button and a "Sign up" link. The main area has a large "Log in" title, fields for "Username" (containing "bob") and "Password" (containing "*****"), and a "Log in" button. Below the button is a green success message: "Logged in successfully".

Figure 8.8 Haiku Haven's login page (see `chapter_8/in_progress_03` in the GitHub repo for the full code)

Note the navigation panel created by `st.navigation`, which takes up the sidebar and contains links to navigate to either page (along with the icons we added!).

8.4.3 Navigating between pages

While we currently have the bare minimum that we need in a signup/login flow, there's definitely room for improvement. For instance, if the user is on the login page but doesn't have an account, there should be a helpful link right there to sign up, or vice versa.

Also, when the user logs in, we should take them to some kind of logged-in page and give them the ability to log out.

Figure 8.9 lays out the ideal signup/login/logout flow we want to design.

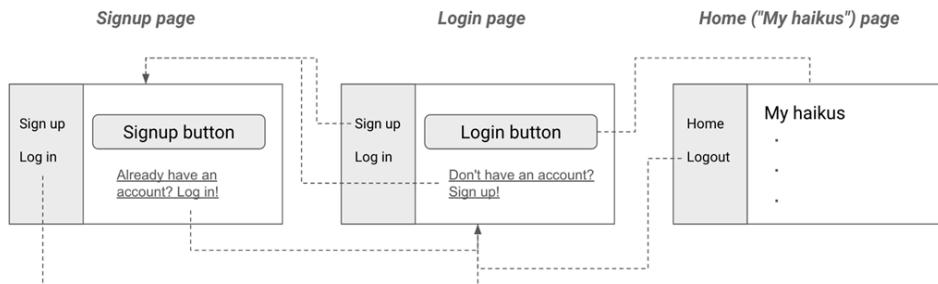


Figure 8.9 Diagram showing the connections between pages through redirection and page links

Besides being able to go back and forth between the signup and login pages, and redirect to a home page when logged in, we also want the navigation panel to display different options based on if we're logged in or not.

If the user is logged in, they should see the home page and have the ability to log out—which shows them the login page again. If they're not, they should instead see the options to sign up or log in.

MOVING THE PAGES DICTIONARY TO ITS OWN FILE

Since we're now going to be navigating between individual pages, it would be cleaner to put the `pages` dictionary (the one in `main.py` that defines the available pages) in its own separate file, so let's move that part of the code to `frontend/pages.py`, shown in listing 8.11.

Listing 8.11 `frontend/pages.py`

```
import streamlit as st

pages = {
    "login": st.Page("frontend/login.py", title="Log in",
                     icon=":material/login:"),
    "signup": st.Page("frontend/signup.py", title="Sign up",
                      icon=":material/person_add:"),
    "home": st.Page("frontend/home.py", title="Home",
                    icon=":material/home:"),
    "logout": st.Page("frontend/logout.py", title="Log out",
                      icon=":material/logout:")
}
```

You'll notice that we've added two new pages: `home`, which is supposed to represent the logged-in home page, and `logout`, which will log the user out.

LINKS BETWEEN PAGES

Streamlit allows you to create links between pages in a multi-page app through a widget that is, appropriately enough, named `st.page_link`.

Let's use this to link between the `login` and `signup` pages.

`login.py` should look like this, with the page link added to the very bottom:

```

import streamlit as st
from frontend.pages import pages

...
with st.container(border=True):
    ...
else:
    st.error("Invalid username or password")

st.page_link(pages["signup"], label="Don't have an account? Sign up!")

```

`st.page_link` is quite easy to understand; the first argument is the `st.Page` object (from `pages`, imported from `pages.py`) we want to link to, and the second is the label text.

You can also pass a regular URL as the first argument, in case you want to link to an external page. `signup.py` has very similar changes:

```

import streamlit as st
from frontend.pages import pages

...
with st.container(border=True):
    ...
else:
    st.error("Invalid username or password")

st.page_link(pages["signup"], label="Don't have an account? Sign up!")

```

DYNAMICALLY CHANGING ST.NAVIGATION

The next feature we'll implement is showing the user the right pages for their context in the navigation bar, i.e. `signup` and `login` when they're logged out, or `home` and `logout` when they're logged in.

Edit `main.py` so it now looks like this:

```

import streamlit as st

from backend.hub import Hub
from frontend.pages import pages

if 'hub' not in st.session_state:
    config = st.secrets['config']
    st.session_state.hub = Hub(config)

if 'logged_in' in st.session_state and st.session_state.logged_in:
    page = st.navigation([pages['home'], pages['logout']])
else:
    page = st.navigation([pages['login'], pages['signup']])

page.run()

```

Obviously, the `pages` dictionary is now defined in `pages.py` and imported into `main.py`.

As shown above, to dynamically change what's in the navigation panel, we use the `logged_in` session state variable that we save when the user logs in, and vary the `st.navigation` object that's assigned to `page` accordingly.

If the user is logged in, the navigation bar will now show the `home` and `logout` options. Since `pages['home']` is the first item in the list passed to `st.navigation`, that's the page that will be loaded by default when the user logs in.

Let's now actually set up a placeholder page in `frontend/home.py` (listing 8.12), so that there's something for a logged-in user to see.

Listing 8.12 frontend/home.py

```

import streamlit as st

user = st.session_state.user
st.title(f"Welcome, {user.username}!")

```

Nothing earth-shattering here for now; we just display a greeting that includes the logged-in user's username. Recall that we save the logged-in `User` object in `st.session_state.user` in `login.py`.

AUTOMATIC REDIRECTION FOR LOGIN AND LOGOUT

Our proposed ideal login flow requires the user to be redirected automatically on login and logout. How does this work exactly?

Remember that when a user clicks the "Log in" button, the `logged_in` session state variable is set to `True`.

This means that in the next re-run, `main.py` will pick up the changed value of `logged_in`, display the new navigation panel and load `home.py`.

To make this truly seamless, we have to trigger the re-run though. So add an `st.rerun()` to `login.py`:

```
user = hub.user_service.get_authenticated_user(username, password)
if user:
    st.session_state.logged_in = True
    st.session_state.user = user
    st.rerun()
```

And logging out? Well, that's going to reverse everything that happens at login. Create a `logout.py` with content shown in listing 8.13.

Listing 8.13 frontend/logout.py

```
import streamlit as st

st.session_state.user = None
st.session_state.logged_in = False
st.rerun()
```

That completes our signup/login/logout flow! Re-run the app and try it all out! When you log in, you should now see a different navigation panel and the loaded home page (see figure 8.10).



Figure 8.10 The logged-in page with different options in the navigation bar (see `chapter_8/in_progress_04` in the GitHub repo for the full code)

Clicking "Log out" in the navigation bar will load the login and signup pages again, which have page links to each other at the bottom.

8.5 Creating, reading, updating, and deleting haikus

Now that user authentication is taken care of, it's finally time to work on the crux of our app: the ability to create, read, update and delete haikus. We'll start with haiku creation, encapsulating this behavior in a `HaikuService` class, and then making the appropriate changes to the frontend.

8.5.1 Defining a HaikuService class

The code structure we'll follow in the haiku service is pretty analogous to what we already have in `UserService`.

Let's begin with a `Haiku` dataclass to represent a haiku. Create this as `haiku.py` in the `backend/` folder (listing 8.14).

Listing 8.14 backend/haiku.py

```
from dataclasses import dataclass

@dataclass
class Haiku:
    haiku_id: int
    created_at: str
    author: str
    text: str
```

As in the `User` class, the fields mirror those in our corresponding database table (`haikus`).

`HaikuService` itself is shown in listing 8.15.

Listing 8.15 backend/haiku_service.py

```
from backend.haiku import Haiku

class HaikuService:
    def __init__(self, database):
        self.database = database

    def create_haiku(self, author, haiku_text):
        query = '''
            INSERT INTO haikus (author, text)
            VALUES (%s, %s)
            RETURNING haiku_id, created_at, author, text
        '''
        params = (author, haiku_text)
        results = self.database.execute_query(query, params)
        return Haiku(*results[0]) if results else None
```

Again, the code is fairly analogous to that of `UserService`, so a detailed explanation isn't warranted.

As we've seen, we only need to supply the `author` and `text` fields in our SQL query; the database automatically provides `haiku_id` and `created_at`, and all of the fields are returned as per the `RETURNING` clause.

To wrap up the backend changes, let's add an instance of `HaikuService` to `hub.py`:

```
...
from backend.haiku_service import HaikuService

class Hub:
    def __init__(self, config):
        database = Database(config['connection_string'])
        self.user_service = UserService(database)
        self.haiku_service = HaikuService(database)
```

This will enable the haiku creation function to be accessed from the frontend, as we'll see presently.

8.5.2 Enabling users to create haikus

Our earlier `home.py` was, of course, just a placeholder. Our actual logged-in home page should ideally have a way to create haikus and display them.

For creating haikus, let's create a modal dialog similar to the one we created in Chapter 7.

Create a new file, `frontend/haiku_editor.py` as shown in listing 8.16.

Listing 8.16 `frontend/haiku_editor.py`

```
import streamlit as st

@st.dialog("Haiku editor", width="large")
def haiku_editor(hub, user):
    haiku_text = st.text_area('Enter a haiku')
    if st.button('Save haiku', type='primary'):
        haiku = hub.haiku_service.create_haiku(user.username, haiku_text)
        if haiku:
            st.success('Haiku saved successfully!')
        else:
            st.error('Failed to save haiku')
```

The `haiku_editor` function is decorated with `st.dialog`—which, as we saw in this previous chapter, executes its body in a modal screen.

The body of `haiku_editor` is uncomplicated. We first accept the haiku text entered by the user in an `st.text_area` widget:

```
haiku_text = st.text_area('Enter a haiku')
```

`st.text_area` is precisely what you'd expect it to be—an area for entering several lines of text.

On clicking "Save haiku", we call the `create_haiku` method under `HaikuService` to save it to the database, and show the appropriate success/failure message.

Close the loop by including—in `home.py`—an "Add Haiku" button that triggers the `haiku_editor` dialog we just defined:

```
import streamlit as st
from frontend.haiku_editor import haiku_editor

hub = st.session_state.hub
user = st.session_state.user
st.title(f"Welcome, {user.username}!")

if st.button(':material/add_circle: Haiku', type='primary'):
    haiku_editor(hub, user)
```

As you can see, we're using a Material icon again, this time in the place of the word "Add" within the button label.

Let's see everything working so far! Re-run your Streamlit app, login, and add a haiku (see figure 8.11).

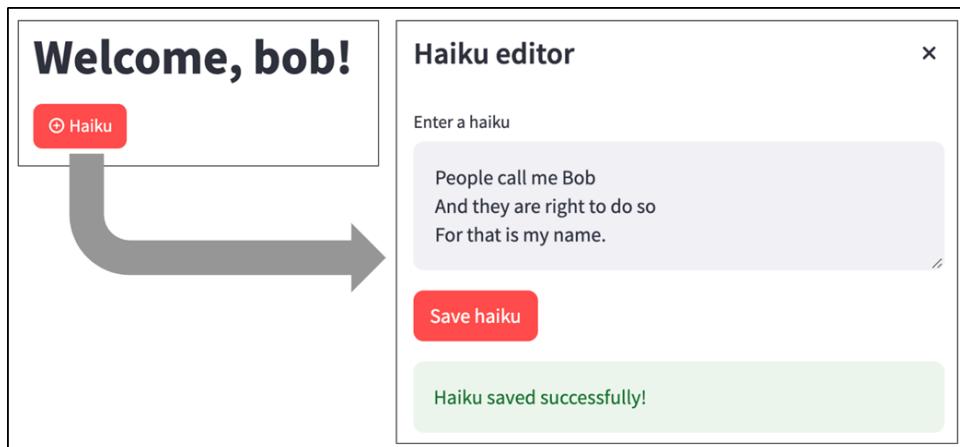


Figure 8.11 Creating a haiku (see `chapter_8/in_progress_05` in the GitHub repo for the full code)

To convince yourself that a haiku has indeed been created, you can query your `haikus` table.

8.5.3 The other CRUD operations: Read, Update, Delete

To make our CRUD app feature complete, there are three more operations we need to implement:

- *Reading* the current users' haikus from the database and list them in the app
- *Updating* a given haiku
- *Deleting* a haiku altogether

DEFINING THE OPERATIONS IN HAIKUSERVICE

As before, let's define these operations in the backend first, by editing the `HaikuService` class in `haiku_service.py`:

```
from backend.haiku import Haiku

class HaikuService:
    ...
    def create_haiku(self, author, haiku_text):
        ...

    def get_haikus_by_author(self, author):
        query = 'SELECT * FROM haikus WHERE author = %s'
        params = (author,)
        results = self.database.execute_query(query, params)
        return [Haiku(*row) for row in results]

    def update_haiku(self, haiku_id, haiku_text):
        query = 'UPDATE haikus SET text = %s WHERE haiku_id = %s RETURNING *'
        params = (haiku_text, haiku_id)
        results = self.database.execute_query(query, params)
        return Haiku(*results[0]) if results else None

    def delete_haiku(self, haiku_id):
        query = 'DELETE FROM haikus WHERE haiku_id = %s RETURNING *'
        params = (haiku_id,)
        results = self.database.execute_query(query, params)
        return Haiku(*results[0]) if results else None
```

We define one method for each operation: `get_haikus_by_author` for reading haikus, `update_haiku` for updating, and `delete_haiku` for deleting a haiku.

In each case, we use the same pattern we've seen before—we run a SQL command, convert the results into `Haiku` objects, and return them.

In the case of `get_haikus_by_author`, we return the results as a *list* of `Haiku` objects given that there may be more than one by a certain author.

In both `update_haiku` and `delete_haiku`, we accept `haiku_id`—the unique identifier of a haiku—as an argument, using `UPDATE..SET` and `DELETE FROM` SQL commands respectively to achieve the desired result.

CREATING THE UI

Our backend methods return instances of the `Haiku` class, but how do we display these in the frontend?

We should probably create a `display_haiku` function that accepts a `Haiku` object and displays it on the screen. The next natural question is: what do we display? What is the user likely to be interested in seeing?

The `Haiku` class has four attributes: `haiku_id`, `created_at`, `author`, and `text`.

Of these, `haiku_id` is an internal identifier that would be of no meaning to the end user, so we can exclude that. `created_at` might be useful to jog the user's memory of when a haiku was originally created. It would be redundant to show the `author` field since we're only going to be displaying haikus by the current logged-in user. And `text` is the content of the haiku, so we obviously want to display that.

That gives us: `created_at` and `text`. Anything else? Well, we also want to give the user the option to edit or delete a particular haiku, so let's pop in a couple of buttons as well.

Listing 8.17 `frontend/haiku_display.py`

```
import streamlit as st

hub = st.session_state.hub
user = st.session_state.user

def get_haiku_created_display(haiku):
    day = haiku.created_at.strftime('%Y-%m-%d')
    time = haiku.created_at.strftime('%H:%M')
    return f':gray[:material/calendar_month: {day} \n :material/schedule: {time}]'

def get_haiku_text_display(haiku):
    display_text = haiku.text.replace('\n', ' \n')
    return f':green[{display_text}]'

def edit_button(haiku):
    if st.button(':material/edit:', key=f"edit_{haiku.haiku_id}"):
        pass

def delete_button(haiku):
```

```

if st.button(':material/delete:', key=f"delete_{haiku.haiku_id}"):
    pass

def display_haiku(haiku):
    with st.container(border=True):
        cols = st.columns([2, 5, 1, 1])
        created_col, text_col, edit_col, delete_col = cols

        created_col.markdown(get_haiku_created_display(haiku))
        text_col.markdown(get_haiku_text_display(haiku))
        with edit_col:
            edit_button(haiku)
        with delete_col:
            delete_button(haiku)

```

The key function to focus on here is the last one: `display_haiku`. Given `haiku`, an instance of the `Haiku` class, it makes four columns for the four things we want to display: `created_at`, `text`, the edit button, and the delete button.

The actual rendering of each of these takes place in its own function.

`get_haiku_created_display` takes `haiku`'s `created_at` property—a timestamp—and breaks it down into a day and a time using a method called `strftime`, which is used to format a timestamp in any given format based on a *format string*. In this case, `%Y-%m-%d` formats it as just a date, whereas `%H:%M` extracts the time in hours and minutes.

```
f':gray[:material/calendar_month: {day} \n :material/schedule: {time}]'
```

There are a few things going on here. The `:<color>[<text>]` syntax is used in Streamlit to display text in various colors. For instance, the string `:red>Hello` would be interpreted by widgets like `st.write` or `st.markdown` as the word `Hello` in red text.

We also see the icon syntax we saw earlier. Here we're using a calendar icon for the date and a clock icon (`schedule`) for the time, creating a user-friendly display for the date and time.

`get_haiku_text_display` is meant to display the content of the haiku. Why do we have the following replace method? Why not just display the content directly?

```
haiku.text.replace('\n', ' \n')
```

This is something of a workaround. Streamlit handles line breaks in text rather strangely. To get text widgets like `st.markdown` to display the newline character `\n` properly, we have to precede it with two spaces, i.e. `' \n'` instead of just `'\n'`.

The `edit_button` and `delete_button` functions simply display `st.button` widgets. You'll notice that we use icons as their labels, and give them widget keys—which is required for Streamlit to distinguish between them when we have many haikus displayed on a page. We've given them placeholder bodies with `pass` for now; we'll come back to them later.

Since we're using icons so much throughout the app, our code would actually be a lot more readable if we gave them better names (e.g. `CLOCK` instead of `:material/schedule:`) and put them in a more central location.

Let's put all the icons in their own file, `frontend/icons.py` (listing 8.18) and *import* them instead.

Listing 8.18 `frontend/icons.py`

```
LOGIN = ":material/login:"
SIGNUP = ":material/person_add:"
HOME = ":material/home:"
LOGOUT = ":material/logout:"
ADD = ":material/add_circle:"
CALENDAR = ":material/calendar_month:"
CLOCK = ":material/schedule:"
EDIT = ":material/edit:"
DELETE = ":material/delete:"
```

We can now change `haiku_display.py`:

```
import streamlit as st
from frontend.icons import CALENDAR, CLOCK, EDIT, DELETE

...
def get_haiku_created_display(haiku):
    ...
    return f':gray[{CALENDAR} {day} \n {CLOCK} {time}]'

...
def edit_button(haiku):
    if st.button(f'{EDIT}', key=f"edit_{haiku.haiku_id}"):
        pass

def delete_button(haiku):
    if st.button(f'{DELETE}', key=f"delete_{haiku.haiku_id}"):
        pass
...
```

There—that's much more readable! Let's now call `display_haiku` to show the user their list of created haikus! We'll edit `home.py` to do this:

```
import streamlit as st
from frontend.haiku_editor import haiku_editor
from frontend.haiku_display import display_haiku
from frontend.icons import ADD

...
if st.button(f'{ADD} Haiku', type='primary'):
    haiku_editor(hub, user)

haikus = hub.haiku_service.get_haikus_by_author(user.username)
if len(haikus) == 0:
    st.info("You haven't written any haikus yet.")
else:
    for haiku in haikus:
        display_haiku(haiku)
```

The changes here are quite simple. After the add button we created earlier—which we've now changed to use an icon imported from `icons.py`—we display a header that says `<username>`'s haikus along with a divider (just a horizontal line underneath, controlled by `divider="gray"`).

We then call the `get_haikus_by_author` method we defined in `HaikuService`, and loop through the results, calling `display_haiku` on each. If there are no haikus, we display an `st.info` message that says so.

Re-run your app now to see the changes in figure 8.12 (after adding one more haiku)!

Welcome, bob!

+ Haiku

bob's haikus

📅 2024-12-20 People call me Bob
 ⏰ 12:28 And they are right to do so
 For that is my name.



📅 2024-12-20 Once it was Robert,
 ⏰ 12:41 But that is not the case now.
 As discussed, I'm Bob.



Figure 8.12 Listing created haikus (see chapter_8/in_progress_07 in the GitHub repo for the full code)

Note that there's one more place we've been using icons: `pages.py`, where we've defined the pages for the multi-page app. Go ahead and update that file too:

```
import streamlit as st
from frontend.icons import LOGIN, LOGOUT, SIGNUP, HOME

pages = {
    "login": st.Page("frontend/login.py", title="Log in", icon=LOGIN),
    "signup": st.Page("frontend/signup.py", title="Sign up", icon=SIGNUP),
    "home": st.Page("frontend/home.py", title="Home", icon=HOME),
    "logout": st.Page("frontend/logout.py", title="Log out", icon=LOGOUT)
}
```

ADDING THE UPDATE AND DELETE FUNCTIONALITY

Let's get back to the edit and delete buttons, which currently have placeholders under them. The edit button should let a user edit one of their existing haikus. We can repurpose the haiku editor dialog we've already created to work with the "Add Haiku" button for this:

```

import streamlit as st

@st.dialog("Haiku editor", width="large")
def haiku_editor(hub, user, haiku=None):
    default_text = haiku.text if haiku else ''
    haiku_text = st.text_area('Enter a haiku', value=default_text)
    if st.button('Save haiku', type='primary'):
        if haiku:
            new_haiku = hub.haiku_service.update_haiku(haiku.haiku_id, haiku_text)
        else:
            new_haiku = hub.haiku_service.create_haiku(user.username, haiku_text)
        if new_haiku:
            st.success('Haiku saved successfully!')
            st.rerun()
        else:
            st.error('Failed to save haiku')

```

The `haiku_editor` function now accepts a `haiku` argument which is `None` by default. If we're calling the editor to edit an existing haiku, we can pass the corresponding `Haiku` instance to `haiku`. Otherwise we're calling the dialog to add a haiku, so we pass `None`.

In the rest of the function, we'll use the condition `if haiku` to check if we're performing the edit action or the add action.

In the next two lines, we pre-populate the existing haiku's text in the text area as a default value using the `value` parameter of `st.text_area` if we're in the edit action.

```

default_text = haiku.text if haiku else ''
haiku_text = st.text_area('Enter a haiku', value=default_text)

```

Then, once the save button is clicked, we pick either the `update_haiku` or the `create_haiku` method from `HaikuService` to execute. In the former case, we pass the existing haiku's `haiku_id` to identify the haiku we want to edit.

If the operation is successful—which we determine by checking the return value, `new_haiku`—we issue an `st.rerun()`. This re-runs the entire app, closing the dialog in the process, because the button that triggered it in the first place is now in the "unclicked" state.

We can now replace the placeholder under the edit button in `haiku_display.py`:

```

def edit_button(haiku):
    if st.button(f'{EDIT}', key=f"edit_{haiku.haiku_id}"):
        haiku_editor(app, user, haiku)

```

In the same page, let's also make the delete button trigger a deletion:

```
def delete_button(haiku):
    if st.button(f'{DELETE}', key=f"delete_{haiku.haiku_id}"):
        deleted_haiku = app.haiku_service.delete_haiku(haiku.haiku_id)
        if deleted_haiku:
            st.rerun()
        else:
            st.error("Failed to delete haiku.")
```

Here we call the `delete_haiku` method in `HaikuService`. If the deletion is successful, we perform an `st.rerun()` so that the list of haikus can update and no longer show the deleted haiku. If the deletion fails for any reason, we show an error.

Re-run the app now and try editing or deleting a haiku (figure 8.13)!

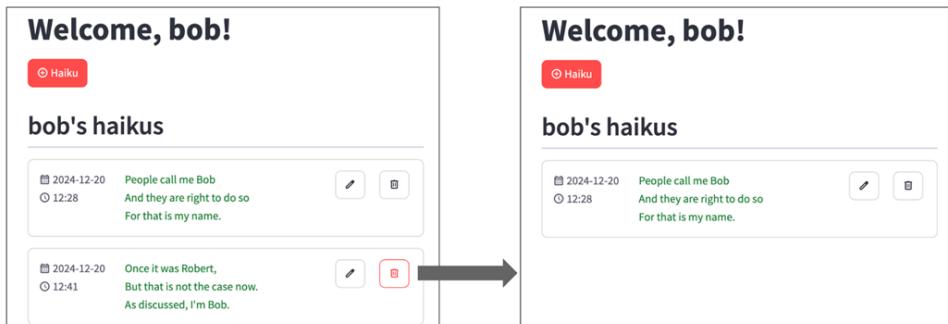


Figure 8.13 Deleting a haiku (see chapter_8/in_progress_08 in the GitHub repo for the full code)

Haiku Haven has now been fully built—or at least we have a version that's fit to deploy to production. Before we do that though, there are a couple of final issues we need to address.

8.6 Multi-user considerations

While Haiku Haven can be used by many users simultaneously, we need to be able to share resources efficiently between these users. The main resource that we're responsible for managing here is the database.

8.6.1 Using `st.cache_resource` to share the database connection pool

Let's consider how a Streamlit app works when there are multiple users accessing it simultaneously. While there's a single Streamlit server that serves the app, each time a user accesses it, a new instance of the app is created, with all of the objects required to run the app created anew for that user.

Most of the time, this is what we want; it makes sure different user sessions don't interfere with each other. However, there are some things we *don't* want to create anew each time someone loads the app in a new browser tab.

A key example is the database connection pool we create in the `Database` class. In fact, the whole point of having a database connection pool is that when multiple user sessions need to access the database, they can do so by requesting connections from a *shared* pool, returning them when they're done.

This means that there should only be *one* instance of the `Database` class ever created, to be shared across all users. However, we haven't set it up that way. Right now, whenever a new user loads the app—or a user loads the app in a new browser tab—a new session begins, and a new instance of the `Hub` class is created, which means a new `Database` instance is also created.

Fortunately, Streamlit offers a solution for this: `st.cache_resource`.

Like `st.cache_data`—which we used in Chapter 6 to make loading the data in our metric dashboard faster—`st.cache_resource` is a way to make sure that only one instance of something exists across all users of an app.

While `st.cache_data` is used for caching things like Pandas dataframes or the results of API calls, `st.cache_resource` is used for resources like database connections. In this case, we'll use it to store the single instance we want to create of the `Database` class.

This requires refactoring our app a little. Recall that currently (in `hub.py`) our `Hub` class accepts a `config` object and creates the `Database` instance itself:

```
...
class Hub:
    def __init__(self, config):
        database = Database(config['connection_string'])
        self.user_service = UserService(database)
        self.haiku_service = HaikuService(database)
```

But in our new scheme, the `Hub` class will continue to have a new instance created for every session, which means a new `Database` instance will *also* keep getting created. Instead, we'll have `Hub`'s `__init__` accept an *already-created* `Database` instance that it can simply pass to `UserService` and `HaikuService`.

That way, we can always pass the *same* instance of the `Database` class, avoiding creating it every time. `hub.py` will now look like this:

```
...
class Hub:
    def __init__(self, database):
        self.user_service = UserService(database)
        self.haiku_service = HaikuService(database)
```

Where do we create the `Database` instance then? In `main.py`, which is where we'll use `st.cache_resource`:

```
...
from backend.database import Database
from frontend.pages import pages

@st.cache_resource
def get_database():
    connection_string = st.secrets['config']['connection_string']
    database = Database(connection_string)
    return database

if 'hub' not in st.session_state:
    st.session_state.hub = Hub(get_database())
...

```

As is the case with `st.cache_data`, `st.cache_resource` is a decorator that's applied to a function. Here we define a new function to decorate called `get_database` which will return the cached `Database` object.

`get_database` will now run only once for a particular Streamlit server that's started up, i.e. a single `streamlit run` command—when `main.py` is loaded for the first time. For all subsequent runs, `get_database` will return the cached `Database` instance, thereby making sure only one such instance is ever created. This cached instance will persist until the server process is terminated.

8.6.2 Using `atexit` to clean up database connections

We've made sure that a `Database` instance is only created once per running instance of a Streamlit app server. That's one part of managing our database resource efficiently. The other is safely cleaning up any connections we create.

Our `Database` class has a `close_all` method:

```
def close_all(self):
    print("Closing all connections...")
    self.connection_pool.closeall()
```

If you inspect our code so far, however, we're not actually calling this method anywhere.

To clean up our connections correctly, we want this method to be called only once, when the Streamlit server is terminated. How do we accomplish this?

The solution is the `atexit` module, which comes built-in with Python. `atexit` lets you register functions to be executed automatically when the Python interpreter is about to exit.

Here's how we would modify `main.py` to register a function that cleans up our database connections using `atexit`:

```
import streamlit as st
import atexit

...
@st.cache_resource
def get_database():
    connection_string = st.secrets['config']['connection_string']
    database = Database(connection_string)
    atexit.register(lambda db: db.close_all(), database)
    return database
...
```

`atexit.register` accepts a function to register, along with the values of any parameters we want to pass to the function. In the above code, the function we're registering is a one-line lambda function: `lambda db: db.close_all()`.

It accepts one parameter—`db`, which is the database instance. All the function does is to invoke `db`'s `close_all` method. The second parameter we pass to `atexit.register` is `database`, which is the `Database` instance created in the line above.

Calling `atexit.register` "schedules" a function to be executed when Python itself exits, which, in the case of `streamlit run`, happens when the Streamlit server shuts down.

Why do we put the call to `atexit.register` within `get_database`? Why not put it somewhere else, perhaps at the end of the file? Well, like the creation of a `Database` object, we only want to perform the registration of the function with `atexit` *once* across all users—because there's only one Streamlit server. That means we have to call `atexit.register` within a function decorated with `st.cache_resource`, or this registration would happen multiple times.

To see this change in action, try restarting the app server with `streamlit run`, and then press `Ctrl+C` to exit it. You should see the message `Closing all connections...` which indicates that `close_all` has been called.

8.7 Deploying Haiku Haven

Since we have Haiku Haven working locally, it's time to productionize our app on Community Cloud. The process to do so is the same as what we've been following since Chapter 5, but there's an additional wrinkle here: our app requires a running PostgreSQL server to host our database.

8.7.1 Setting up a managed PostgreSQL server in production

While we were developing locally, it was a simple matter to install PostgreSQL on the same machine, but Streamlit Community Cloud doesn't provide an option to do that. Instead, we need to set up an external PostgreSQL server somewhere.

We'll use a cloud-based managed PostgreSQL service called Neon, which makes this process super-easy, and has a significant free quota. Create an account with Neon now at <https://neon.tech/>. The signup process is quite painless; you can choose to sign up with your GitHub or Google account if you like.

You'll be asked for a project name and a database name. The project name can be whatever you like (`Haiku Haven`, maybe?), while the database name should be whatever you named your database in your local Postgres—`haikudb` if you've been following along faithfully.

You might also be asked to choose a cloud provider and location—these can be whatever you like, though I chose AWS as the provider.

Once your account is set up, navigate to the Quick Start page to see a connection string that looks something like this:

```
postgresql://haikudb_owner:Dxg2HFXreSZ3@ep-flower-dust-a63e8evn.us-west-2.aws.neon.tech/haikudb?sslmode=require
```

This is the connection string we'll use in production. Neon has taken the liberty of setting up a username (`haikudb_owner`) and password for you. Store this string somewhere safe.

Next, you'll need to set up the `users` and `haikus` tables once again in Neon. To do this, go to the SQL Editor tab. This is where you can enter SQL commands as though you were in the `psql` prompt. To create the tables, refer back to section 8.2.4 and grab the `CREATE TABLE` commands we executed locally. You can execute these commands in Neon's SQL Editor without any changes.

8.7.2 Deploying to Community Cloud

The rest of the deployment process should be straightforward, and is pretty much identical to what we did in Chapter 5.

Make sure to create a `requirements.txt` so that Community Cloud knows to install the third-party modules needed—primarily `psycopg2` and `bcrypt`.

For reference, listing 8.19 shows the `requirements.txt` that I used.

Listing 8.19 requirements.txt

```
streamlit==1.40.2
psycopg2-binary==2.9.10
bcrypt==4.2.0
```

Notice that I used `psycopg2-binary` instead of `psycopg2`; that's because Community Cloud threw an error when I tried to use the latter but not with the former.

Once you've completed deployment, you'll need to copy the contents of `secrets.toml` and paste it in Community Cloud's Secrets setting (refer to Chapter 5 for a refresher), replacing the connection string with the one you copied from Neon.

That's all, folks! Haiku Haven is now live! Tell all your friends they can unleash their seventeen-syllable creativity on your brand new web app!

As for us, let's turn the page on CRUD and try our hand at building AI apps next.

8.8 Summary

- CRUD stands for Create-Read-Update-Delete, the four fundamental operations that most apps perform.
- Relational databases like PostgreSQL organize data in tables with rows and columns according to a schema.
- Designing the data model for an application involves identifying the entities, defining their relationships, listing their attributes, and converting these into a schema, often assisted by an Entity-Relationship (ER) diagram.
- SQL (Structured Query Language) supports commands for creating tables (`CREATE TABLE`), inserting rows (`INSERT INTO`), reading data (`SELECT..FROM`), updating rows (`UPDATE..SET`), deleting rows (`DELETE FROM`) and dropping tables (`DROP TABLE`).
- `psycopg2` is a Python module used to connect to PostgreSQL using a shared connection pool.
- Never store passwords in plain text; instead hash them using a library like `bcrypt` and store the hashed version. To authenticate, hash the given password and compare with the stored hash.
- `st.Page` objects correspond to individual pages in multi-page apps in Streamlit.
- `st.navigation` is used to create a navigation bar and specify the pages in an app.
- `st.page_link` creates links between pages in a multi-page app.
- `st.cache_resource` is used to cache resources like database connections and share them between users.
- Use `atexit.register` from the built-in `atexit` module to register a function to execute when a Streamlit server shuts down.
- When deploying to production, you need to set up your database server separately, potentially using a managed service like Neon for PostgreSQL.

9 An AI-powered trivia game

This chapter covers

- Connecting your app to a Large Language Model (LLM)
- Engineering LLM prompts to achieve your desired results
- Using Structured Outputs to obtain LLM responses in a custom parsable format
- Managing state in a sequential Streamlit app
- Using `st.data_editor` to create editable tables

Creating software is significantly different from what it used to be just a few short years ago. The difference stems from major developments in the field of AI (Artificial Intelligence), which—unless you've been living under a rock—you've probably heard of.

I'm talking, of course, about breakthroughs in LLMs or Large Language Models, and the tremendously exciting possibilities they open up. By processing and generating natural language, LLMs can understand context, answer complex questions, and even write software on their own—all with astonishing fluency. Tasks that once required domain-specific expertise or painstaking programming can now be achieved with just a few well-crafted "prompts".

In this chapter, we'll dive into how you can harness the power of LLMs in your own applications, relying on AI prompts and responses to implement product features that would have required highly advanced techniques half a decade ago. Along the way, we'll also discuss how to tune your LLM interactions to get the results you want, and how to do so without burning a hole in your pocket.

NOTE The GitHub repo for this book is <https://github.com/aneevdavis/streamlit-in-action>. The chapter_09 folder has this chapter's code and a requirements.txt file with exact versions of all the required Python libraries.

9.1 Fact Frenzy: An AI trivia game

If you watched the game show *Jeopardy!* as a kid—or as an adult for that matter—you're going to love this chapter. Having grown up outside the US, I was in my thirties before I watched my first episode of the show, but I suffered no dearth of trivia shows to obsess over when I was a boy—*Mastermind*, *Bournvita Quiz Contest*, and *Kaun Banega Crorepati?* (an Indian take on the original British show *Who Wants to Be a Millionaire?*) were all staples of my youth.

Fifteen-year-old me would never forgive adult me if I didn't include at least one trivia app in this book. Fortunately, trivia is a great fit for our first AI-powered Streamlit app, which will generate questions, evaluate answers, and even mix it up with a variety of quizmaster styles, all using artificial intelligence.

9.1.1 Stating the concept and requirements

Once again, the first step we'll take is to state the concept of the app we want to build.

CONCEPT Fact Frenzy, a trivia game that asks users a set of trivia questions and evaluates their answers using AI

As we've done several times before by this point in the book, the requirements will flesh out this simple idea further.

REQUIREMENTS

Fact Frenzy will:

- use an AI model to generate trivia questions
- ask these questions to a player
- allow the user to enter a free-text response to each question
- use AI to evaluate whether the answer is correct, and to provide the correct answer
- keep track of a player's score
- allow the player to set a difficulty level for questions
- offer a variety of quizmaster speaking styles

While we *generally* want our requirements to be free of "implementation" language (as discussed in Chapter 3), in this case the entire point of our app is to demonstrate the use of AI—so we definitely need it to use AI models to perform its functions.

We've also added a fun element in the form of quizmaster speaking styles—or in other words, mimicking the style of various people while asking questions, something we'd only be able to achieve using AI.

WHAT'S OUT OF SCOPE

What are we leaving out? While a professional trivia game could be arbitrarily complex, in building Fact Frenzy, we want to create a minimal app to get hands-on practice with topics we haven't encountered before. So we won't focus on any of the following:

- persistent storage and retrieval of questions, answers, and scores
- creating and managing users
- letting users choose particular categories of questions

Placing the above items out of scope will let us concentrate on things like interacting with large language models or LLMs, state management, and of course, new Streamlit elements.

9.1.2 Visualizing the user experience

To give ourselves a concrete idea of what Fact Frenzy is about, take a look at a sketch of our proposed UI, displayed in figure 9.1.

The UI sketch illustrates a user interface for a trivia game named 'Fact Frenzy'. On the left, a sidebar contains two dropdown menus: 'Quizmaster' set to 'Gollum' and 'Difficulty' set to 'Medium'. Below these is a large grey button labeled 'New game'. The main content area on the right is titled 'Question 2/10' and shows a score of 'Score: 1'. A question box contains the text: 'Ah, precious, listen close, we asks a tricky question, yes! Here it comes, my precious: In which year did the famous artist Vincent van Gogh create his painting "Starry Night," oh yes, we wonders, we do!' Below the question box is a text input field labeled 'Enter your answer:' and a dark grey 'Submit' button.

Figure 9.1 A UI sketch of Fact Frenzy

The game window shown in the sketch has a two-column layout. The left column has a "New game" button, as well as a couple of settings: Quizmaster and Difficulty.

Referring to the last requirement identified in the previous section, the Quizmaster setting is supposed to use AI to mimic speaking styles of various characters. In figure 9.1, the selected value is "Gollum", a character from *The Lord of the Rings* who speaks in a distinctive, hissing manner using phrases like "my precious."

The column on the right shows an AI-generated question "spoken" in Gollum's voice along with the player's score and a box to enter the answer.

Gimmicks aside, it's a pretty standard question-and-answer trivia game that lets the player enter free-form text to answer.

9.1.3 Brainstorming the implementation

While we'll "outsource" large parts of our logic to an LLM, we'll still need to own the overall game flow. Figure 9.2 shows the design we'll work to implement in the rest of the chapter.

Unlike some of the other apps we've written where users could take a variety of actions at any point, Fact Frenzy is quite linear. As the diagram shows, the basic logic runs in a loop—using an LLM to retrieve a trivia question, posing it to the player, getting the LLM to evaluate the answer, stating whether the provided answer was right or not, and doing it all over again for the next question, until the game is done.

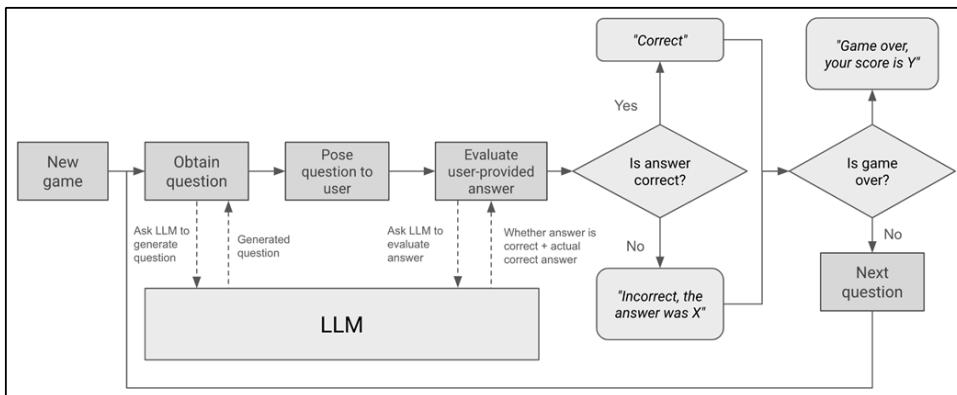


Figure 9.2 The flow of logic in our AI-based trivia game

Later on, we'll add a few bells and whistles such as allowing the player to set the difficulty level and quizmaster, but figure 9.2 is a pretty good representation of the core flow.

9.2 Using AI to generate trivia questions

At the heart of Fact Frenzy is an AI model that powers the trivia experience. This model generates questions, evaluates player responses, and even adds personality to the quizmaster. To achieve all this, we use a Large Language Model (LLM)—a powerful AI system designed to process and generate human-like text.

LLMs are trained on vast amounts of textual data, making them incredibly versatile. They can perform a wide range of tasks from answering factual questions to generating poetry, coding, or—importantly for our purposes—role-playing as a quizmaster. Popular examples of LLMs include OpenAI's GPT series, Anthropic's Claude, and Google's Gemini—all of which leverage cutting-edge machine learning techniques to generate coherent, contextually appropriate text.

9.2.1 Why use an LLM in Fact Frenzy?

What makes LLMs suitable for use in our trivia game? To answer this, let's take a minute to consider some possible components of such a game and how an LLM can support building each one:

A GIANT LIST OF QUESTIONS

Without an LLM, we would need to maintain a set of trivia questions large and diverse enough to keep our app engaging. Since the major LLMs of today are all trained on text and data pertaining to history, culture, geography, astronomy, and every other category of trivia you can think of, they're instead able to generate questions on the fly.

THE ABILITY TO EVALUATE ANSWERS

If we were going the traditional non-LLM route, we would realistically only be able to ask multiple-choice questions, as we'd need to match the answer a player gives to the actual one accurately. LLMs, on the other hand, can interpret and respond to free-text user inputs. This allows us to ask open-ended trivia questions and still evaluate player responses correctly.

ENTERTAINMENT VALUE

In addition to being able to handle factual information, LLMs can also be creative, and provide humor for engagement. Later in the chapter, we'll ask one to mimic the style of various characters as quizmasters, giving the game a personality, so to speak.

Thus, in Fact Frenzy, the LLM will play the triple role of question generator, answer evaluator, and comic relief provider. In the next section, we'll set up an account with a top-tier LLM provider—OpenAI—enabling us to start interacting with LLMs for the first time.

NOTE While LLMs are undeniably powerful, they are not omniscient. They rely on patterns in their training data to generate responses and may occasionally make mistakes, especially when evaluating highly nuanced or ambiguous answers. However, for our trivia app, these models strike a perfect balance between intelligence, versatility, and entertainment.

9.2.2 Setting up an OpenAI API key

For this chapter, we'll use an LLM provided by OpenAI, perhaps the best-known among the set of new-ish AI firms that have been dominating tech news lately.

As we've done many times in this book so far, we'll need to open an account with an external service, and wire our Python code to it.

If you don't already have an OpenAI account (your ChatGPT account counts), go to <https://platform.openai.com/> and sign up for one.

NOTE If you created your OpenAI account recently or just now, you *may* have some free credits applied to your account. However, the free tier currently comes with relatively severe usage limits, such as only being able to call some models three times per minute. While you could *technically* get away with using the free tier for this chapter (assuming your account has the free credits in the first place), if you plan to do any amount of serious AI development, I recommend upgrading to a paid tier. As you'll see, you can get a lot of LLM usage for fairly low prices. Though you'll likely need to *buy* \$5 in usage credits to get to the lowest paid tier, you won't need to *spend* much of that in this chapter. For reference, I spent less than 15 cents in credits for all the testing I did while developing this lesson. You can learn more about cost optimization in the sidebar named "Cost considerations".

Once you're in, go to the Settings page—at the time of writing, you can do this by clicking the gear icon at the top right corner of the page—and then click on "API keys" in the panel to the left.

Create a new secret key and note it down. You can leave all the defaults unchanged (see figure 9.3).

Create new secret key

Owned by

You Service account

This API key is tied to your user and can make requests against the selected project. If you are removed from the organization or project, this key will be disabled.

Name Optional

My Test Key

Project

Default project

Permissions

All Restricted Read only

Figure 9.3 Creating a secret key in your OpenAI account

You'll only be able to see your key once, so copy-and-paste it somewhere. This goes without saying, but keep it safe!

9.2.3 Calling the OpenAI API in Python

Before we start any development on the app, let's make sure we can call the OpenAI API in Python without any issues.

To begin, install the OpenAI's Python library with `pip install openai`. You could technically call the API via HTTP calls using the `requests` module (as we did in Chapter 5), but this library is more convenient to use.

Once that's done, open a Python shell and import the `OpenAI` class:

```
>>> from openai import OpenAI
```

The `OpenAI` class lets you instantiate a client using an API key to make calls to the OpenAI API:

```
>>> client = OpenAI(api_key='sk-proj-...')
```

Replace `sk-proj-...` with the actual API key you copied in the previous step. With a `client` object created, let's prepare an instruction to send the LLM:

```
>>> messages = [
...     {'role': 'system', 'content': 'You are a helpful programming assistant'},
...     {'role': 'user', 'content': 'Explain what Streamlit is in 10 words or fewer'}
... ]
```

Each request you send to the LLM is called a *prompt*. A prompt—or at least the type we'll use here—consists of *messages*. In the code above, we're assembling these into a list. Each message takes the form of a dictionary with two keys: `role` and `content`.

`role` can be one of `user`, `system`, and `assistant`. We'll explore more examples later, but the value of `role` signifies the perspective of the speaker in the conversation:

- `system` represents instructions or context-setting for the model, such as rules for how it should behave.
- `user` represents the person interacting with the model (you).
- `assistant` represents the model's responses—we'll discuss this in the next chapter.

The prompt we're creating here tells the LLM (in the system message) that it should behave like a helpful programming assistant. The actual instruction we want the LLM to respond to is in the user message: Explain what Streamlit is in 10 words or fewer.

We can now make an actual request to the API:

```
completion = client.chat.completions.create(model='gpt-4o-mini', messages=messages)
```

The OpenAI API has several different endpoints—one for turning audio to text, one for creating images, and so on. The one we'll be using is the *chat completions* endpoint, and it's used for text generation.

Given a list of messages in a conversation, this endpoint is supposed to return what comes next—hence the term *completion*. OpenAI has a plethora of models we could use, but here we've picked `gpt-4o-mini`, which provides a good balance between intelligence, speed, and cost.

NOTE While gpt-4o-mini is currently the most suitable model OpenAI offers for our use case, given the speed of developments in the AI space, by the time this book goes to print, we may have newer models that are smarter *and* cheaper. Keep an eye on OpenAI's pricing page at <https://openai.com/api/pricing/> to ensure you're using the model that fits best.

The above statement should take a few seconds to execute, and will return a `ChatCompletion` object. If you like, you can inspect this object by typing just `completion` into the shell, but you can access the actual text response we want like so:

```
>>> completion.choices[0].message.content  
'Streamlit is a framework for building interactive web applications easily.'
```

I couldn't have said it better myself! That concludes our first programmatic interaction with an LLM. Next, let's build this into our trivia game's code.

9.2.4 Writing an LLM class

In Chapter 8, we created a `Database` class that encapsulated the interaction that our app could have with an external database. We'll follow the same model in this chapter with an `LLM` class that handles all communication with the external LLM. This allows us to separate the logistics of talking to the LLM from the rest of our app, making it easier to maintain, test, or even swap it out entirely, without touching the remaining code.

We've covered the basics of calling the LLM in the prior section, so all that's left is to put the logic in a class.

Create a new file, `llm.py` with the content shown in listing 9.1.

Listing 9.1 llm.py

```
from openai import OpenAI

class Llm:
    def __init__(self, api_key):
        self.client = OpenAI(api_key=api_key)

    @staticmethod
    def construct_messages(user_msg, sys_msg=None):
        messages = []
        if sys_msg:
            messages.append({"role": "system", "content": sys_msg})
        messages.append({"role": "user", "content": user_msg})
        return messages

    def ask(self, user_msg, sys_msg=None):
        messages = self.construct_messages(user_msg, sys_msg)
        completion = self.client.chat.completions.create(
            model="gpt-4o-mini",
            messages=messages
        )
        return completion.choices[0].message.content
```

(chapter_09/in_progress_01/llm.py in the GitHub repo)

The `Llm` class' `__init__` simply creates a new OpenAI client object using an API key that's passed to it, assigning this to `self.client`.

The `ask` method is what logic outside the `Llm` class will interact with, and returns the LLM's response to our prompt. Its code is essentially the same as what we ran in the Python shell earlier, except that we take in `user_msg` and `sys_msg` as arguments and put the creation of the `messages` list in its own method, called `construct_messages`.

Since we don't have to pass a system role message—the LLM will try to be helpful anyway—we give `sys_msg` a default value of `None`. `construct_messages` takes this fact into account while generating the `messages` list. Since this is a utility function that doesn't depend on anything else in the object, we make it a static method by decorating it with `@staticmethod`.

We'll refine the `Llm` class further along in the chapter, but for now, let's move on to writing the code that calls it.

9.2.5 The Game class

As in Chapter 8, we'll have a single class—appropriately named `Game`—that contains all the backend logic that our app's frontend will call directly. This is somewhat analogous to the `Hub` class from Chapter 8, though we'll structure `Game` differently.

For the moment we'll keep it quite simple, as all it needs to do is to pass a prompt to our `Llm` class. The initial version of the `Game` class that we'll place in `game.py` is shown in listing 9.2.

Listing 9.2 game.py

```
from llm import Llm

class Game:
    def __init__(self, llm_api_key):
        self.llm = Llm(llm_api_key)

    def ask_llm_for_question(self):
        return self.llm.ask(
            'Ask a trivia question. Do not provide choices or the answer.',
            'You are a quizmaster.'
        )
```

(chapter_09/in_progress_01/game.py in the GitHub repo)

The initialization of a `Game` instance (through `__init__`) involves creating an `Llm` object by passing it the API key that we'll presumably get from the calling code.

The `ask_llm_for_question` method passes a simple prompt asking the LLM to generate a trivia question. Notice that the system message now tells the LLM to behave like a quizmaster.

The user message instructs the LLM to ask a question, warning it to not provide any choices or reveal the answer.

9.2.6 Calling the Game class in our app

We can now write a minimal version of our frontend code to test out everything we've done. As usual, our API key needs to be kept secret and safe, so we'll put it in a `secrets.toml` file in a new `.streamlit` directory, as seen in listing 9.3.

Listing 9.3 .streamlit/secrets.toml

```
llm_api_key = "sk-proj-..." #A  
#A Replace sk-proj-... with your actual API key.
```

We've kept `secrets.toml` quite simple this time around with a non-nested structure—notice the absence of a section like `[config]`. The O in TOML does stand for "obvious" after all.

Go ahead and create `main.py` (shown in listing 9.4) now.

Listing 9.4 main.py

```
import streamlit as st
from game import Game

game = Game(st.secrets['llm_api_key'])

question = game.ask_llm_for_question()
st.container(border=True).write(question)
```

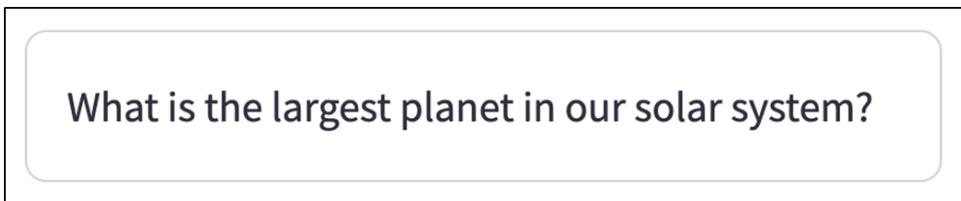
(`chapter_09/in_progress_01/main.py` in the GitHub repo)

There's nothing fancy here yet; we simply create a `Game` instance called `game`, call its `ask_llm_for_question` method to generate a trivia question, and write it to the screen.

Notice how we've combined `st.container` with a border and `st.write` into a single statement:

```
st.container(border=True).write(question)
```

Concise and pretty, much like Streamlit itself. Run your app using `streamlit run main.py` (as in prior chapters, make sure you first `cd` into the directory that contains `main.py` so Streamlit can find the `.streamlit` directory) to see something like figure 9.4.



What is the largest planet in our solar system?

Figure 9.4 The app can now successfully call the OpenAI API to obtain a trivia question (see `chapter_09/in_progress_01` in the GitHub repo for the full code)

Excellent—Our AI quizmaster can now ask the player a question! Next we'll have it evaluate the player's answer.

9.3 Using AI to evaluate answers

The prompt we fed GPT-4o mini is a simple one. Vitaly, we didn't have to do anything particularly complicated with the output—just display it on the screen as-is. As such, we didn't really care about the *format* in which the LLM responded.

However, evaluating answers presents a slightly different challenge. When a player enters an answer into the app, we need the LLM to tell us two things:

- Whether the answer is correct
- If not, what the correct answer actually is

If you've interacted with AI assistants before, this sounds well within their capabilities. But let's examine a practical challenge: how do you reliably parse the LLM's response?

For instance, let's say we tell the LLM "Hey, you asked this question: <question>, and the player said the answer was <answer>. Tell me if that's right, or if not, what the correct answer is".

The LLM does its thing and responds with:

"Incorrect, the answer is actually <correct answer>".

What do we do with this reply? Sure, we could display it on the screen, but we probably also need to perform additional actions, like deciding whether or not to increment the player's score. Which means that we need to *parse* the response to understand whether the answer was right.

How do we do that? A naive approach would be to look for the word "incorrect" in the response and mark the answer as being correct or not accordingly.

But what if the LLM's answer is actually " Nope, that's not right, the answer is <correct answer>" or even "Hah! It *would* have been correct if they'd said <correct answer>. But they didn't, so tough luck."

The point is that there are tons of creative ways the LLM could answer, and while we would be able to understand them as humans, we need a simple way to determine their meaning in a machine-friendly way.

We could request the LLM in our prompt to include the words "correct" or "incorrect" somewhere, but it might still occasionally mess it up. Luckily, there's a better approach.

9.3.1 Structured Outputs

Being able to parse LLM outputs reliably is a natural concern for developers, so OpenAI has a solution for this called *Structured Outputs*.

Structured Outputs is a feature that ensures that the model will generate a response that adheres to a schema that *you* provide, making it simple to parse programmatically.

For our use case, this means we can request the LLM to provide two structured fields in its response: a boolean field that says whether the provided answer is correct, and the actual correct answer.

Let's create this schema as a class named `AnswerEvaluation` (listing 9.5). We'll need the third-party `pydantic` module to get this working, so install that first with `pip install pydantic`.

Listing 9.5 answer_evaluation.py

```
from pydantic import BaseModel

class AnswerEvaluation(BaseModel):
    is_correct: bool
    correct_answer: str
```

(chapter_09/in_progress_02/answer_evaluation.py in the GitHub repo)

`pydantic` is a data validation library that uses type hints to ensure that data conforms to a specified type.

`BaseModel`, which we import from `pydantic`, is a class that allows you to define a schema and perform data validation. It works well with OpenAI's Structured Outputs.

The class we're defining, `AnswerEvaluation`, is a *subclass* of `BaseModel`. *Subclasses* and *superclasses* are related to the concept of *inheritance* in object-oriented programming. Explaining inheritance in detail is beyond the scope of this book, but just know that a subclass can *inherit* functionality and attributes from its superclass, allowing you to reuse code and build on existing functionality without starting from scratch.

In this case, `AnswerEvaluation` (subclass) inherits the features of `BaseModel` (superclass), such as data validation, serialization, and type checking, making it easy to define and work with structured data.

The body of `AnswerEvaluation` is identical to what you might expect if it were a dataclass instead. Indeed, dataclasses are similar to `pydantic`'s `BaseModel` except that dataclasses do not provide the complex validations and related functionality that `BaseModel` does.

Fortunately, we don't really need to worry about the internals of how this works—just note that `AnswerEvaluation` has the two fields we spoke of: a boolean `is_correct` and a string, `correct_answer`.

Next, let's modify our `ask` function in the `Llm` class (`llm.py`) to support Structured Outputs:

```

from openai import OpenAI

class Llm:
    ...
    def ask(self, user_message, sys_message=None, schema=None):
        messages = self.construct_messages(user_message, sys_message)

        if schema:
            completion = self.client.beta.chat.completions.parse(
                model="gpt-4o-mini",
                messages=messages,
                response_format=schema
            )
            return completion.choices[0].message.parsed
        else:
            completion = self.client.chat.completions.create(
                model="gpt-4o-mini",
                messages=messages
            )
            return completion.choices[0].message.content

```

(chapter_09/in_progress_02/llm.py in the GitHub repo)

Here we've added an argument called `schema` that's `None` by default. If a value is provided (if `schema`), we call a different method in our OpenAI client (`beta.chat.completions.parse` as opposed to `chat.completions.create` from earlier).

The first two parameters we pass to this new method are the same as before, but we've added a third one: `response_format`, to which we provide the value of `schema`.

The final value we return is also different: `completion.choices[0].message.parsed` rather than `completion.choices[0].message.content`.

If `schema` is not provided, we simply default to the earlier behavior, thereby ensuring that the `ask` method can handle both Structured Outputs and regular text.

The value we need to pass to `schema` is a *class*—not an instance of the class, but the class *itself*. The value *returned* will then be an *instance* of that class, and will therefore follow the schema. As you may already have guessed, for our use case, we'll pass the `AnswerEvaluation` class to `schema`.

We'll create this calling code in a bit, but first let's create a new LLM prompt asking the model to evaluate a player's answer.

During development, you should expect to have to tweak your prompt many times to get better results—in fact, there's a whole field called *prompt engineering* that has sprung up in recent years.

Since our prompts are meaningfully different from our code, let's put them in a different file where we can edit them without touching the rest of the code. We'll name this `prompts.py` and give it the contents in listing 9.6.

Listing 9.6 prompts.py

```
QUESTION_PROMPT = {
    'system': 'You are a quizmaster.',
    'user': 'Ask a trivia question. Do not provide choices or the answer.'
}

ANSWER_PROMPT = {
    'system': 'You are an expert quizmaster.',
    'user': '''

        You have asked the following question: {question}
        The player answered the following: {answer}

        Evaluate if the answer provided by the player is close enough
        to be correct.

        Also, provide the correct answer.
    '''
}
```

(`chapter_09/in_progress_02/prompts.py` in the GitHub repo)

Each prompt is structured as a dictionary with keys `system` and `user`, corresponding to the system and user messages.

`QUESTION_PROMPT` is our prompt from earlier, while `ANSWER_PROMPT` is new. Notice that its user message (a Python *multi-line string* bounded by ''s if you're wondering about the syntax) contains these lines:

```
You have asked the following question: {question}
The player answered the following: {answer}
```

We're treating `{question}` and `{answer}` here as variables that we can replace with real values when we send the prompt to the LLM later.

Also note the last two lines in this message where we're telling the LLM to evaluate the answer for correctness and *also* to provide the correct answer. The model is smart enough to interpret this instruction and provide the results in our `AnswerEvaluation` schema.

Speaking of which, let's actually write the code that passes this schema along with the prompt to fulfill our answer evaluation use case. We'll do this by modifying `game.py`:

```

from llm import Llm
from prompts import QUESTION_PROMPT, ANSWER_PROMPT
from answer_evaluation import AnswerEvaluation

class Game:
    def __init__(self, llm_api_key):
        self.llm = Llm(llm_api_key)

    def ask_llm_for_question(self):
        usr_msg, sys_msg = QUESTION_PROMPT['user'], QUESTION_PROMPT['system']
        return self.llm.ask(usr_msg, sys_msg)

    def ask_llm_to_evaluate_answer(self, question, answer):
        sys_msg = ANSWER_PROMPT['system']
        user_msg = (
            ANSWER_PROMPT['user']
            .replace('{question}', question)
            .replace('{answer}', answer)
        )
        return self.llm.ask(user_msg, sys_msg, AnswerEvaluation)

```

(chapter_09/in_progress_02/game.py in the GitHub repo)

We've refactored the `ask_llms_for_question` method to use our new `prompts.py` module.

But the main change here is the new function, `ask_llm_to_evaluate_answer`, which takes in the originally asked `question`, and the `answer` provided by the user, plugging these values into the `{question}` and `{answer}` slots in the user message we discussed a minute ago.

This time, we pass in `AnswerEvaluation` (imported from `answer_evaluation.py`) as a third argument to `self.llm's` `ask` method—schema, as I hope you recall. One interesting aspect to this is that we're passing in `AnswerEvaluation itself` here—not an *instance* of `AnswerEvaluation`, but the class. In Python, most of the constructs in your code are *objects* that you can pass around like this, including classes—something that enables powerful and flexible programming patterns.

But I digress. Let's get back to enabling our game to accept and evaluate answers from a player. The last step is to make the requisite changes to the frontend in `main.py`:

```
import streamlit as st
from game import Game

game = Game(st.secrets['llm_api_key'])

question = game.ask_llm_for_question()
st.container(border=True).write(question)

answer = st.text_input("Enter your answer")
if st.button("Submit"):
    evaluation = game.ask_llm_to_evaluate_answer(question, answer)
    if evaluation.is_correct:
        st.success("That's correct!")
    else:
        st.error("Sorry, that's incorrect.")
        st.info(f"The correct answer was: {evaluation.correct_answer}")
```

(chapter_09/in_progress_02/main.py in the GitHub repo)

You should be able to understand what we're doing here pretty easily. Once we've posed the trivia question to the player, we display a text input for their answer, as well as a "Submit" button that when clicked, triggers the `ask_llm_to_evaluate_answer` method in `game.py`.

The resulting value—stored in `evaluation`—is an instance of the `AnswerEvaluation` class. We use its `is_correct` attribute to display the appropriate correct/incorrect message, and `evaluation.correct_answer` for the real answer.

Try re-running your app now and supplying an answer to the trivia question. Figure 9.5 shows what I got.

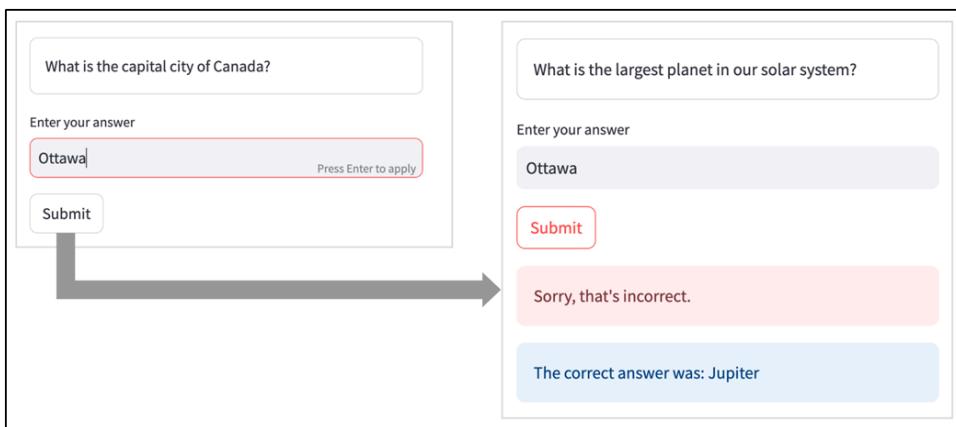


Figure 9.5 An issue with session state causes our answer to be matched with the wrong question (see chapter_09/in_progress_02 in the GitHub repo for the full code)

Oops! There are shenanigans afoot! The question I answered "Ottawa" to in the left part of figure 9.5 was "What is the capital of Canada?". But when I press the "Submit" button, the app displays a different question—"What is the largest planet in our solar system?"—and then has the nerve to tell me that "Ottawa" is, in fact, *not* the right answer to that.

You'll notice a similar bait-and-switch in your testing. What's going on? Are our AI overlords toying with us?

As it turns out, the LLM is entirely innocent of this alleged mischief. The issue—as we've seen several times before in this book—is related to session state and Streamlit app re-runs. In this particular instance, a click on the "Submit" button triggers a re-run from the *top*, meaning that before the code under `if st.button("Submit"):` can run, `game.ask_llm_for_question()` is called once again, resulting in a *new* question, which is what is passed as the first argument to `ask_llm_to_evaluate_answer`, along with "Ottawa" as the second argument pulled from the text input.

Well, at the very least, the Structured Outputs part of our code seems to be working, since Jupiter is indeed the largest planet in the solar system, and Ottawa is incorrect.

However, to get Fact Frenzy to exhibit the behavior we expect, we'll need to think through state management quite thoroughly in the next section.

9.4 Moving through game states

In chapters prior, Streamlit's rerun-the-whole-app-each-time model meant that we had to make extensive use of `st.session_state` to get the app to remember values. That holds true here as well.

Fact Frenzy, however, is *sequential* in a way that our previous apps weren't. You can think of the desired behavior of our game as moving between various *states*, taking a different set of actions and displaying different widgets in each. Figure 9.6 illustrates this.



Figure 9.6 Our game consists of four sequential states

The diagram identifies four states that the game can be in:

- GET_QUESTION, in which we retrieve a question from the LLM
- ASK_QUESTION, where we pose said question to the player
- EVALUATE_ANSWER, which involves calling the LLM to evaluate the answer
- STATE_RESULT, where we tell the player whether they were right or not

Each of these actions should only happen while the game is in the state that it corresponds to. Additionally, we need to ensure that each LLM request we're making only happens once per question—because the alternative messes up our logic *and* costs money.

A pattern that we'll find useful in sequential apps such as this one and others you may write in the future is to formally retain a state/status attribute in the app's central class (`Game` in our case) coupled with methods to modify it, and to use conditional logic based on this attribute in the frontend to display the right elements on the screen. Figure 9.7 should make what I'm talking about clearer.

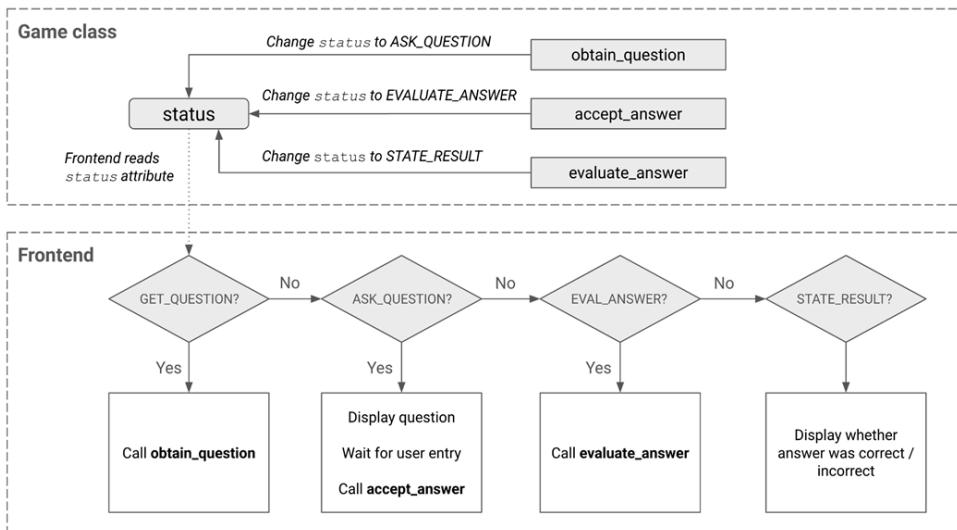


Figure 9.7 Conditional branching in the frontend based on the game's state

As you can see from our proposed logic in figure 9.7, the `Game` class will have a `status` attribute that indicates the state the game is in. We'll read this state in the frontend to branch between the various display elements—whether it's a text input for the answer in the `ASK_QUESTION` state, or simply a wait-during-the-LLM-request status element during the `GET_QUESTION` and `EVALUATE_ANSWER` states.

The Game class also has methods that change its status attribute, controlling all of this. obtain_question would presumably get the question from the LLM, but also change the status to ASK_QUESTION when it's done. accept_answer would take in the answer from the player and also switch the status to EVALUATE_ANSWER, and the evaluate_answer method, in addition to getting the LLM to give us the result, would shift the status to STATE_RESULT.

Let's put all this in action now, starting with the Game class:

Listing 9.7 game.py (modified)

```
from llm import Llm
from prompts import QUESTION_PROMPT, ANSWER_PROMPT
from answer_evaluation import AnswerEvaluation

class Game:
    def __init__(self, llm_api_key):
        self.llm = Llm(llm_api_key)
        self.status = 'GET_QUESTION'

        self.curr_question = None
        self.curr_answer = None
        self.curr_eval = None

    def ask_llm_for_question(self):
        usr_msg, sys_msg = QUESTION_PROMPT['user'], QUESTION_PROMPT['system']
        return self.llm.ask(usr_msg, sys_msg)

    def ask_llm_to_evaluate_answer(self):
        sys_msg = ANSWER_PROMPT['system']
        user_msg = (
            ANSWER_PROMPT['user']
            .replace('{question}', self.curr_question)
            .replace('{answer}', self.curr_answer)
        )
        reply = self.llm.ask(user_msg, sys_msg, AnswerEvaluation)
        return reply

    def obtain_question(self):
        self.curr_question = self.ask_llm_for_question()
        self.status = 'ASK_QUESTION'
        return self.curr_question

    def accept_answer(self, answer):
        self.curr_answer = answer
        self.status = 'EVALUATE_ANSWER'
```

```
def evaluate_answer(self):
    self.curr_eval = self.ask_llm_to_evaluate_answer()
    self.status = 'STATE_RESULT'
```

(chapter_09/in_progress_03/game.py in the GitHub repo)

As discussed, the first big change here is the inclusion of a `self.status` attribute that formally indicates the state of the game. We initialize this to `GET_QUESTION` as that's the first sequential state we want.

You'll notice we also have three other attributes—`curr_question`, `curr_answer`, and `curr_eval`—to hold the current question, answer, and evaluation within the `Game` instance. This is a departure from the earlier version of `game.py` where we were handling `question` and `answer` as variables outside the class. Keeping track of these within the class is better-suited to our new *stateful* approach.

You'll see this reflected in the `ask_llm_to_evaluate_answer` method, where we've dispensed with the `question` and `answer` parameters in favor of the `self.curr_question` and `self.curr_answer` attributes.

Additionally, we've introduced three new methods (also discussed earlier)—`obtain_question`, `accept_answer`, and `evaluate_answer`. `obtain_question` and `evaluate_answer` are wrappers around `ask_llm_for_question` and `ask_llm_to_evaluate_answer` respectively, each merely assigning the result to its associated attribute—`self.curr_question` or `self.curr_answer`—before adding a line to move `self.status` to its next value.

`accept_answer` is even simpler; it just sets `self.answer` to an answer presumably provided by the player.

The second half of the set of changes we need to implement the state management approach we've envisioned lies in `main.py`:

```

import streamlit as st
from game import Game

if 'game' not in st.session_state:
    st.session_state.game = Game(st.secrets['llm_api_key'])
game = st.session_state.game

if game.status == 'GET_QUESTION':
    with st.spinner('Obtaining question...') as status:
        question = game.obtain_question()
        st.rerun()

elif game.status == 'ASK_QUESTION':
    st.container(border=True).write(game.curr_question)
    answer = st.text_input("Enter your answer")
    if st.button("Submit", type='primary'):
        game.accept_answer(answer)
        st.rerun()

elif game.status == 'EVALUATE_ANSWER':
    with st.spinner('Evaluating answer...') as status:
        game.evaluate_answer()
        st.rerun()

elif game.status == 'STATE_RESULT':
    if game.curr_eval.is_correct:
        st.success("That's correct!")
    else:
        st.error("Sorry, that's incorrect.")
        st.info(f"The correct answer was: {game.curr_eval.correct_answer}")

```

(chapter_09/in_progress_03/main.py in the GitHub repo)

We start by placing our `Game` instance in `st.session_state`, ensuring that we'll be dealing with the same instance across re-runs:

```

if 'game' not in st.session_state:
    st.session_state.game = Game(st.secrets['llm_api_key'])
game = st.session_state.game

```

The last line here is for convenience, enabling us to refer to our `Game` instance simply as `game`, instead of spelling out `st.session_state.game` each time.

The remaining code builds out conditional branching based on `game`'s `status` attribute. Let's briefly consider each such condition:

```
if game.status == 'GET_QUESTION':
    with st.spinner('Obtaining question...') as status:
        question = game.obtain_question()
    st.rerun()
```

The first branch deals with the `GET_QUESTION` state. The app handles this state without any interaction from the user, as it merely obtains the question from the LLM. This can take a perceptible amount of time, however, so we display what's known as a *status element*.

A status element is simply a widget that gives some indication of what's going on to the user while a long-running operation happens in the background. Streamlit has several different status elements—`st.spinner`, `st.status`, `st.toast`, `st.progress`—each of which has slightly different characteristics.

`st.spinner`, which we've used here, simply displays a spinning circle animation (the same one we saw in Chapter 6 when we applied the `show_spinner` parameter to `@st.cache_data`) until the background operation has been completed.

Notice the `st.rerun()` after we've called `game.obtain_question()`. This exists because once `obtain_question` (from `game.py`) has changed the status to `ASK_QUESTION`, we need the code to run again, so as to enter the *next* conditional branch given by `elif game.status == 'ASK_QUESTION':`.

The remaining branches are quite similar. In each case, there is some kind of trigger causing the app to move to the next state, followed by a rerun. In the `ASK_QUESTION` state, a click on "Submit" calls `game.accept_answer(answer)`, which sets `game`'s `curr_answer` attribute and changes the state to `EVALUATE_ANSWER`.

In `EVALUATE_ANSWER`, we call `game.evaluate_answer()` and display another `st.spinner` while we wait for it to return, eventually changing the status to `STATE_RESULT`.

After the rerun, we simply display the appropriate messages based on `game.curr_eval`, our `AnswerEvaluation` object.

Check out the results now by rerunning the app (figure 9.8)

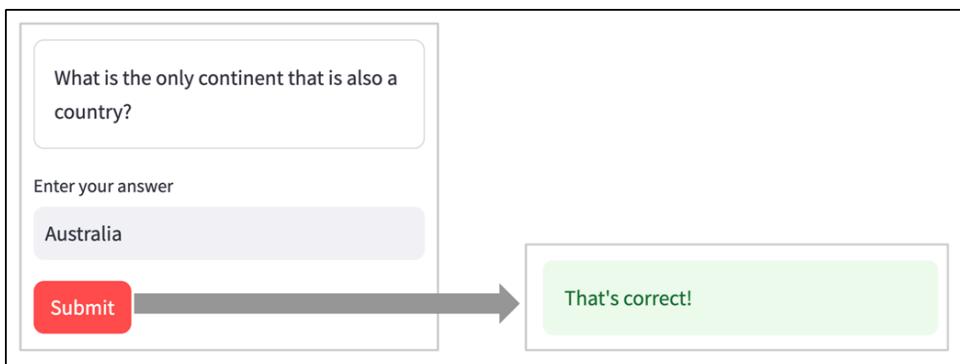


Figure 9.8 The question-answer mismatch issue has been resolved (see chapter_09/in_progress_03 in the GitHub repo for the full code)

This time you'll see that the app matches the question and answer correctly, fixing the issue we saw earlier.

9.5 Game mechanics: Keeping score, a New Game button, and Game Over

Fact Frenzy currently does the bare minimum we need it to do: it asks a player a question, and evaluates the answer, all using AI. It isn't much of a *game* though; there's no score, and no concept of when the game starts and ends. Let's tackle each of these problems in turn.

KEEPING SCORE

We want Fact Frenzy to be easy to pick up, so we'll keep our scoring mechanism basic; each correct answer fetches one point.

This should be fairly trivial to incorporate into the `Game` class:

```
...
class Game:
    def __init__(self, llm_api_key):
        self.llm = Llm(llm_api_key)
        ...
        self.score = 0

    ...
    def evaluate_answer(self):
        self.curr_eval = self.ask_llm_to_evaluate_answer()
        if self.curr_eval.is_correct:
            self.score += 1
        self.status = 'STATE_RESULT'
```

(chapter_09/in_progress_04/game.py in the GitHub repo)

We simply add a new `score` attribute to the `Game` instance in `__init__`, setting it to 0 to start with.

Later, in the `evaluate_answer` method, we increment this `score` by 1 if the LLM determines that the answer is correct.

GAME OVER

Our app doesn't yet have any concept of when the game is over, so let's define this. Again, we'll keep the logic simple: we'll ask a predefined number of questions, and the game is done when all of them have been asked and answered.

This involves modifying the `Game` class again:

```

...
class Game:
    def __init__(self, llm_api_key):
        ...
        self.score = 0
        self.num_questions_completed = 0
        self.max_questions = 1

    ...
    def evaluate_answer(self):
        self.curr_eval = self.ask_llm_to_evaluate_answer()
        self.num_questions_completed += 1
        if self.curr_eval.is_correct:
            self.score += 1
        self.status = 'STATE_RESULT'

    def is_over(self):
        return self.num_questions_completed >= self.max_questions

```

(chapter_09/in_progress_04/game.py in the GitHub repo)

Again, this should be pretty easy to follow. We add two more attributes to the instance in `__init__`: `num_question_completed` and `max_questions` (set to 1 for now since we don't actually support multiple questions yet—that's coming up in the next section).

We increment `num_questions_completed` by 1 in `evaluate_answer`, and add a new method called `is_over` that returns `True` if the number of questions completed matches or exceeds `self.max_questions`.

A NEW GAME BUTTON

At the moment, Fact Frenzy jumps straight into asking the LLM for a question as soon as the page loads. A "New Game" button would let users trigger the start of a game or take any other action we may want to add later before we engage the LLM.

This will primarily affect our frontend code, so let's update `main.py` thus:

```

import streamlit as st
from game import Game

def start_new_game():
    st.session_state.game = Game(st.secrets['llm_api_key'])
    st.rerun()

def new_game_button(game):
    if game and not game.is_over():

```

```

        button_text, button_type = "Restart game", "secondary"
    else:
        button_text, button_type = "Start new game", "primary"
    if st.button(button_text, use_container_width=True, type=button_type):
        start_new_game()

game = st.session_state.game if 'game' in st.session_state else None
side_col, main_col = st.columns([2, 3])
with side_col:
    st.header(" Fact Frenzy", divider='gray')
    new_game_button(game)

with main_col:
    if game:
        st.header("Question", divider='gray')
        if game.status == 'GET_QUESTION':
            ...
            ...
            ...
        elif game.status == 'STATE_RESULT':
            if game.curr_eval.is_correct:
                st.success("That's correct!")
            else:
                st.error("Sorry, that's incorrect.")
                st.info(f"The correct answer was: {game.curr_eval.correct_answer}")

        if game.is_over():
            with st.container(border=True):
                st.markdown(f"Game over! Your final score is: **{game.score}**")

```

(chapter_09/in_progress_04/main.py in the GitHub repo)

There are several changes to highlight here. Firstly, there are two new functions: `start_new_game` and `new_game_button`, which we'll look into in a second.

Since it's now possible for a game to not have started yet—before the "New Game" button is clicked, we allow for game to be `None` if it hasn't been added to `st.session_state` yet:

```
game = st.session_state.game if 'game' in st.session_state else None
```

We've also changed the layout of the app to have two columns: a side column (`side_col`) and a main one (`main_col`):

```
side_col, main_col = st.columns([2, 3])
```

This side column could have simply been an `st.sidebar`, but in a later section, it'll turn out we need this column to have a higher width than `st.sidebar` offers by default.

Anyway, `side_col` contains a header introducing Fact Frenzy, and a call to `new_game_button`:

```
with side_col:
    st.header(" Fact Frenzy", divider='gray')
    new_game_button(game)
```

In Chapter 8, we used the Material library to display icons. Here, we've given Fact Frenzy a lightning-bolt icon using a different approach: by pasting an emoji into our code. We're able to do this because emoji are part of the *Unicode* standard, which defines a consistent way to represent text and symbols across different systems and platforms. Whenever you want to add an emoji, you can search for it on a website like emojipedia.org and copy it.

The `new_game_button` function is defined thus:

```
def new_game_button(game):
    if game and not game.is_over():
        button_text, button_type = "Restart game", "secondary"
    else:
        button_text, button_type = "Start new game", "primary"
    if st.button(button_text, use_container_width=True, type=button_type):
        start_new_game()
```

In essence, we're checking if the game is already in progress—if `game` and not `game.is_over()` determines that `game` doesn't have the value `None` and that its `is_over` method returns `False`—and displaying a different button according to the result.

We vary two characteristics of the button—its text and its type. The text says "Restart game" if the game is in progress, and "Start new game" if it's not.

How about the button's `type` parameter? We may have given it a value in previous chapters, but let's examine it more thoroughly now. The three values `type` can take are `primary`, `secondary`, and `tertiary`—each indicating how prominent the button should be. A button with `type primary` has a solid color (usually orange) with white text, a `secondary` button—the default if you don't specify a `type`—is white with solid-colored text, while a `tertiary` button is more subtle and appears as regular text without a border.

In UI design, it's a good practice to guide the user towards the "correct" or most likely action they might want to take at any given point—it makes for a more intuitive design. If the game hasn't started yet, the choice that makes the most sense is to click the "Start new game" button, so we give it a type of primary. If the game is in progress, the default action should be to answer the question, not to restart the game. Therefore, while we make that possibility available, we don't overly emphasize it.

These differences in the button are mostly cosmetic, however. In either case, a click issues a call to `start_new_game`, which has the following code:

```
def start_new_game():
    st.session_state.game = Game(st.secrets['llm_api_key'])
    st.rerun()
```

As before, we create a `Game` instance and assign it to `st.session_state.game`. Since the presence of `game` in `st.session_state` changes what should be displayed on the screen, we also issue an `st.rerun()`.

By wrapping this logic in a function, we're preventing it from executing by default, instead requiring the New Game button to actually be clicked.

The main column of the game—`main_col`—is, of course, where the content is meant to be displayed. In this iteration of `main.py`, we've simply moved the widgets we had before into `main_col`. There are a few additions worth highlighting though.

If `game` is `None`—which means no game has started yet—we want the main column to be completely blank so the player's attention is focused on `side_col`. This explains why the code under `with main_col` starts with `if game`:

```
with main_col:
    if game:
        st.header("Question", divider='gray')
        if game.status == 'GET_QUESTION':
            ...
    ...
```

We've also added a header that just says "Question." We'll update this later to show the question number.

Finally, we've added some logic to handle the case of the game being over under the `STATE_RESULT` state:

```
elif game.status == 'STATE_RESULT':
    ...
    if game.is_over():
        with st.container(border=True):
            st.markdown(f"Game over! Your final score is: **{game.score}**")
```

This should be quite clear. We use the `is_over` method we defined earlier to check if the game is done, and show an appropriate message and the final score (`game.score`) if so.

That concludes another iteration of our code. Go ahead and re-run your app to get figure 9.9.

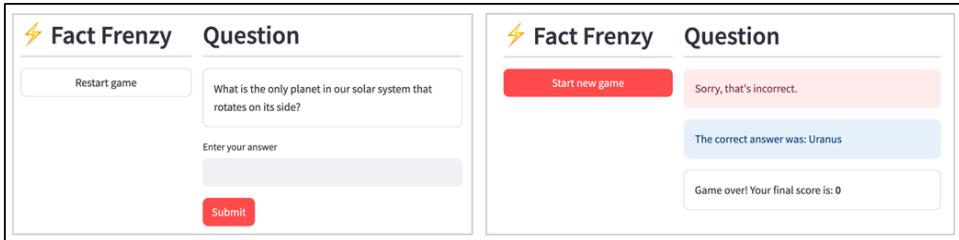


Figure 9.9 Keeping score, a New Game button, and Game Over (see `chapter_09/in_progress_04` in the GitHub repo for the full code)

Sweet! Fact Frenzy is starting to look pretty slick. Adding support for multiple questions is next!

9.6 Including multiple questions

In the previous section, we introduced key game mechanics to our app—adding a score system and defining the game's start and end—to make it feel more like a real game.

Fact Frenzy still only asks one question though, so it's not much of one yet. It's time to change that. But before we do, let's explore an LLM-related challenge we'll face.

9.6.1 Response variability, or lack thereof

In many ways, an LLM is like a black box. Unlike a piece of "regular" code that tends to be deterministic—meaning that the same input always produces the same output—LLMs are *probabilistic*, which means you may get different responses for the same input (or similar inputs), based on a set of probabilities.

Depending on what you're trying to achieve, this variability can be a good or a bad thing. For example, if you're trying to get an LLM to generate poetry, you may want a fairly high amount of creativity or variability in the response, whereas if you're evaluating a mathematical equation, you want less.

Vendors like OpenAI generally expose a few controls for this variability, making it easier to manage, but often we need to engineer the prompt to extract the behavior we want from the model.

For our use case of generating questions, we want relatively high variability. If you've played around with our current app for a while, you may have noticed that the questions we get from the LLM are often repeated. In my testing, for instance, the model had a particular fondness for asking about the only planet in the solar system that rotates on its side.

This won't work for us. For one thing, if a single game includes multiple questions, all of them *must* be unique. Secondly, even if a particular question isn't asked twice in the same game, we don't want it to repeat too often across *different* games either.

Let's take a look at a few ways in which we can control variability in the LLM's answer.

NOTE One consequence of the fact that LLMs generate text based on probabilistic patterns (rather than an understanding of facts) is what we call "hallucinations"—instances where the model produces outputs that are plausible-sounding but factually incorrect or entirely fabricated. These hallucinations arise because LLMs rely on the statistical relationships in their training data, which can sometimes lead to confident but misleading responses. Strategies exist to reduce the likelihood of hallucinations, such as enabling LLMs to connect to external sources of information, but there's no way to guarantee that they won't occur at all. Dealing with hallucinations is outside the scope of this chapter—~we'll tackle supplementing an LLM's knowledge base with our own sources in the next one. Just be aware that our app may occasionally produce a question or answer that isn't factual. Fortunately, based on my testing, these occurrences tend to be infrequent.

VARYING TEMPERATURE AND TOP_P

The prompts we send to a large language model and the response we get back from it both consist of *tokens*. A token may be a single word or it may be part of a word.

At its heart, an LLM constructs the response to a prompt step-by-step, or rather, token-by-token. In each step, it considers a wide range of possibilities for the next token to include in its response, assigning a *probability* of being picked (from high school math, a number between 0 and 1) to each token option.

These tokens form what's known as a *probability distribution*—think of it as a curve that represents the likelihood of each token being the next one, with the more likely tokens plotted at a higher place on the curve than the less likely ones.

OpenAI offers two parameters—`temperature` and `top_p`—that can adjust the composition of this curve. Figure 9.10 illustrates the effect of varying these parameters

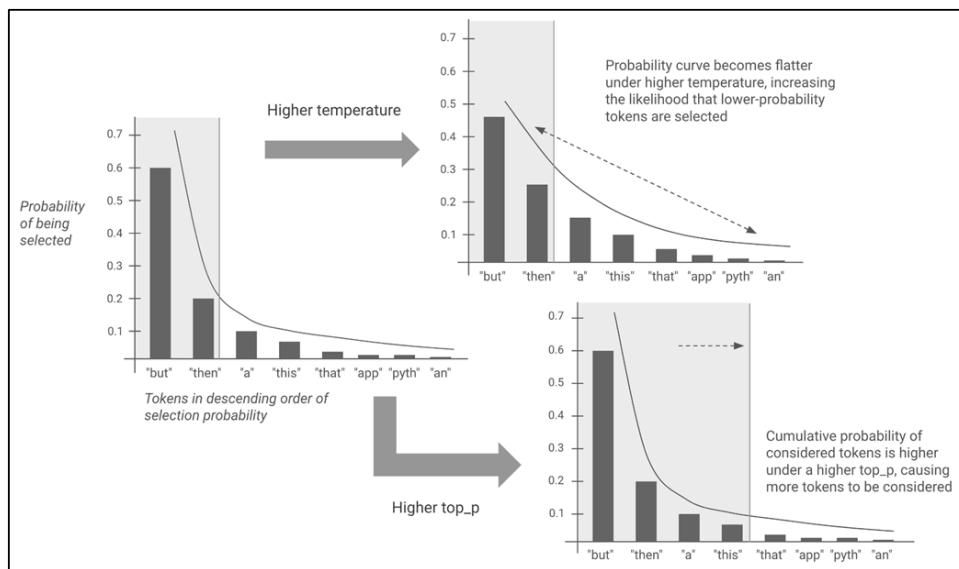


Figure 9.10 How temperature and top_p affect the creativity and predictability of an LLM's output

`temperature` can take values from 0 to 2, with a higher value serving to make the curve flatter, and a lower value making it more pronounced. A higher temperature thus tends to "even" out the curve, increasing the probability that some of the less likely token options may be picked, which makes the LLM take more "risks" and increases the overall creativity of its response.

`top_p` can go from 0 to 1. It represents a cutoff for the cumulative probability of the tokens that the model will choose from. To take an example, the LLM may determine that the five most likely next tokens in its response and their respective probabilities are: "but": 0.6, "then": 0.2, "a": 0.1, "this": 0.06, and "that": 0.02—with all other tokens having much lower probabilities.

A `top_p` of 0.8 would mean that the model should only choose between the most likely tokens with a combined probability of at least 0.8. In this case, since "but" and "then" together cover a probability of $0.6 + 0.2 >= 0.8$, the model discards everything else.

A `top_p` of 0.95, on the other hand, would require the model to also consider "a" and "this" to cover the required cumulative probability ($0.6 + 0.2 + 0.1 + 0.06 >= 0.95$).

A higher `top_p`, therefore, means that the LLM will consider more token options, potentially reducing the predictability and coherence of the response, but increasing its diversity.

Getting back to our original goal of generating a variety of questions, we probably want a moderately high `temperature`—say 0.7 to 0.8—and a relatively high `top_p` of, say, 0.9.

INCLUDING PREVIOUS QUESTIONS IN THE PROMPT

As discussed, while it would be ideal for questions to not repeat across *different* games, we must practically *guarantee* that the same question isn't asked twice in a *single* game.

Fortunately, this is easily achieved—by explicitly telling the LLM which questions have been asked so far in the game so it knows to steer clear of those.

For reinforcement, we could even tell the LLM to make sure to never ask the same question twice.

INJECTING RANDOMNESS

Another way to get a wider variety of questions back is to inject some structured randomness into the prompt. You may have heard of the word game "Mad Libs" where players are provided a story with various parts of speech replaced with blanks. Each player then fills in a blank with a word of their choice, with the completed story often being hilarious.

We could do something similar here. We could change our prompt to something like "Generate a unique trivia question in the category _____ and a topic _____ within that category. The question should reference a person or thing whose name starts with the letter _____".

Within our code, we could then maintain lists of categories, topics, and letters, randomly picking one from each list to fill in the blanks before sending the prompt to the LLM. If we have say, 10 categories, 10 topics within each category, we would have $26 \times 10 \times 10 = 2600$ unique combinations, in addition to the variability that the LLM itself provides.

Or to save us the trouble of maintaining these lists, why not ask the LLM to pick a category and topic first? Interestingly, doing this appears to increase the diversity of the responses generated.

Yet another way to inject randomness that seems to help is to explicitly provide a random seed in your prompt. In programming, a random seed is a value (generally an integer) that initializes a random number generator. While it's not very clear that adding one to the text of your prompt actually causes the LLM to generate a random number, in my testing, doing so did increase the variability of responses.

NOTE It's important to note here that modifying your prompt to get the results you want is not pure science; often you'll need to experiment with various techniques and prompts to identify the approach that works best. You may also see surprising results—for instance, AI researchers have found asking an LLM to think through its approach to solving a problem step-by-step often improves how well it performs the task.

9.6.2 Implementing multiple questions

Now that we've reviewed how variability works in LLMs and possible approaches to ensure we get different questions each time, let's modify Fact Frenzy so it asks the user multiple questions in a game.

MODIFYING THE LLM PROMPT

Let's first make the requisite changes to our prompts to put into practice what we've learned.

Before we do that—our `Llm` class doesn't currently offer a way to change the `temperature` and `top_p`, so we should modify its code (in `llm.py`) thus:

```
from openai import OpenAI

class Llm:
    ...

    def ask(self, user_message, sys_message=None, schema=None,
            temperature=None, top_p=None):
        messages = self.construct_messages(user_message, sys_message)

        llm_args = {'model': 'gpt-4o-mini', 'messages': messages}
        if temperature:
            llm_args['temperature'] = temperature
        if top_p:
            llm_args['top_p'] = top_p

        if schema:
            completion = self.client.beta.chat.completions.parse(
                response_format=schema,
                **llm_args
            )
            return completion.choices[0].message.parsed
        else:
            completion = self.client.chat.completions.create(**llm_args)
            return completion.choices[0].message.content
```

(`chapter_09/in_progress_05/llm.py` in the GitHub repo)

As you can see above, we've refactored the `ask` method in the `Llm` class a fair bit. First, it accepts `temperature` and `top_p` as new arguments, both defaulting to `None`.

Instead of repeating the `model`, `messages`, `temperature`, and `top_p` arguments to the OpenAI client's `beta.chat.completions.parse` or `chat.completions.create`, we construct an `llm_args` dictionary that holds the right arguments and their values depending on whether each is provided.

We then use the `**` dictionary unpacking operator (that we encountered in Chapter 7) to pass the arguments to the OpenAI methods. Note that we can combine this with the normal way of passing arguments:

```
completion = self.client.beta.chat.completions.parse(
    response_format=schema,
    **llm_args
)
```

Here we pass `reponse_format` in the regular way, but unpack `llm_args` for the remaining arguments.

Next, in `prompts.py`, edit our question-generation prompt so it now reads:

```
QUESTION_PROMPT = {
    'system': 'You are a quizmaster who never asks the same question twice.',
    'user': """
        First think of a unique category for a trivia question.
        Then think of a topic within that category.

        Finally, ask a unique trivia question, generated using the random seed
        {seed}, without revealing the category or topic.

        Do not provide choices, or reveal the answer.

        The following questions have already been asked:
        {already_asked}
        ...
    """
}

ANSWER_PROMPT = {
    ...
}
```

(`chapter_09/in_progress_05/prompts.py` in the GitHub repo)

You'll see that we've incorporated several of the techniques discussed in the last section:

- The system prompt requests the LLM to behave like a quizmaster who never asks the same question twice.
- We ask the LLM to think of a unique category and a topic within it.

- We add a "seed" variable and ask the LLM to generate the question using that seed.
- At the end of the prompt, for references, we provide the questions that have already been asked, so the LLM can avoid those.

We need to accompany these changes with additional ones in the `Game` class. Besides the LLM stuff, to enable asking multiple questions in a game, we need to be able to repeat the movement from the first game state to the last many times. In effect, our state diagram now becomes a *cycle* as opposed to a line, as shown in figure 9.11.

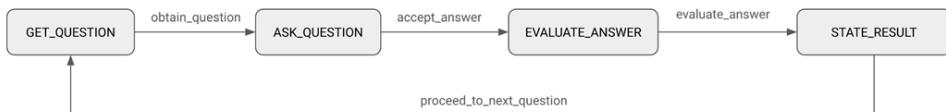


Figure 9.11 To implement multiple questions, we now cycle through the game states in a loop

This means we also need another method to move from the `STATE_RESULT` state back to `GET_QUESTION`. Let's add this method along with the rest of the changes to `game.py` now:

```

import time

from llm import Llm
...


class Game:
    def __init__(self, llm_api_key):
        ...
        self.max_questions = 5
        self.questions = []

    def ask_llm_for_question(self):
        seed = int(time.time())
        sys_msg = QUESTION_PROMPT['system']
        usr_msg = (
            QUESTION_PROMPT['user']
            .replace('{already_asked}', '\n'.join(self.questions))
            .replace('{seed}', str(seed))
        )
        return self.llm.ask(usr_msg, sys_msg, temperature=0.7, top_p=0.9)

    ...
    def obtain_question(self):
        self.curr_question = self.ask_llm_for_question()
  
```

```

    self.questions.append(self.curr_question)
    self.status = 'ASK_QUESTION'
    return self.curr_question

    ...
def proceed_to_next_question(self):
    self.status = 'GET_QUESTION'

    ...

```

(chapter_09/in_progress_05/game.py in the GitHub repo)

You'll see that we've added a new attribute, `self.questions`, within `__init__`, initializing it to an empty list. As you've likely guessed, this will hold all the questions we get from the LLM. We accomplish this through this addition in the `obtain_questions` method:

```
self.questions.append(self.curr_question)
```

Additionally, since we'll finally have more than one question to ask, we've changed the value of `self.max_questions` to 5. Feel free to change this to whatever number you like.

We've revamped the `ask_llm_for_question` method entirely, since our user message now has a couple of variables we need to provide values for. `{already_asked}` can simply be replaced by `self.questions` (with the individual list items separated by newlines).

For the random seed, we simply use the current timestamp converted to an integer:

```
seed = int(time.time())
```

Since timestamps always go up by definition, the current timestamp is guaranteed to be something the LLM has never gotten before from us.

We also now pass `temperature` and `top_p` values to `self.llm.ask` in line with our exploration of these parameters.

To enable multiple questions, a newly added `proceed_to_next_question` sets the game's status back to `GET_QUESTION`, completing the state cycle in figure 9.11.

The changes required to the frontend are relatively simple. Edit `main.py` thus:

```

import streamlit as st
from game import Game

...
with main_col:
    if game:
        st.header(
            f"Question {len(game.questions)} / {game.max_questions}",
            divider='gray'
        )
        st.subheader(f"Score: {game.score}")

    if game.status == 'GET_QUESTION':
        ...

        ...
        ...
        elif game.status == 'STATE_RESULT':
            ...
            if game.is_over():
                with st.container(border=True):
                    st.markdown(f"Game over! Your final score is: **{game.score}**")
            else:
                st.button(
                    "Next question",
                    type='primary',
                    on_click=lambda: game.proceed_to_next_question()
                )

```

(chapter_09/in_progress_05/main.py in the GitHub repo)

Firstly, we've modified the header of the main column to provide the question number (`len(game.questions)`) and the total number of questions (`game.max_questions`):

```

st.header(
    f"Question {len(game.questions)} / {game.max_questions}",
    divider='gray'
)

```

We've also added a subheader to display the score:

```
st.subheader(f"Score: {game.score}")
```

To facilitate the state change from `STATE_RESULT` to `GET_QUESTION`, we've added an `else` clause that will—if the game isn't over—display a "Next question" button that triggers the game object's `proceed_to_next_question()` when clicked.

Notice the unfamiliar way in which we've written the `st.button` widget:

```
st.button(
    "Next question",
    type='primary',
    on_click=lambda: game.proceed_to_next_question()
)
```

`st.button`'s `on_click` parameter lets you specify a function to execute when the button is clicked. We *could* also have written this in the way we've done so far in this book, i.e. as:

```
if st.button("Next question", type='primary'):
    game.proceed_to_next_question()
    st.rerun()
```

The difference lies in when the triggered function is actually executed. Specifically:

- When we use the `if st.button` notation, the button click first triggers a re-run of the page, causing everything above the button to be re-rendered again before the code under the `if` executes—triggered by the fact that `st.button` evaluates to `True` in the re-run. After this code executes, we may need to manually trigger *another* re-run as shown above—and as we've been doing throughout this book, really—to see any changes in the page caused by it.
- With the `on_click` notation, the button click causes the function listed under `on_click` (called a *callback* by the way) to execute *before* the page is re-run and everything above the button is re-rendered. We don't need a manual `st.rerun()` in this case, because the re-run triggered by the button-click already takes into account the changes made by the callback since it has already executed.

So why haven't we been using this method all along? Well, the `if st.button` structure tends to be a little easier to grasp for simple apps. Besides, callbacks have some restrictions—you can't trigger an app rerun within a callback, for instance.

In any case, you should be able to re-run your app at this point to try out a working multi-question game (figure 9.12).

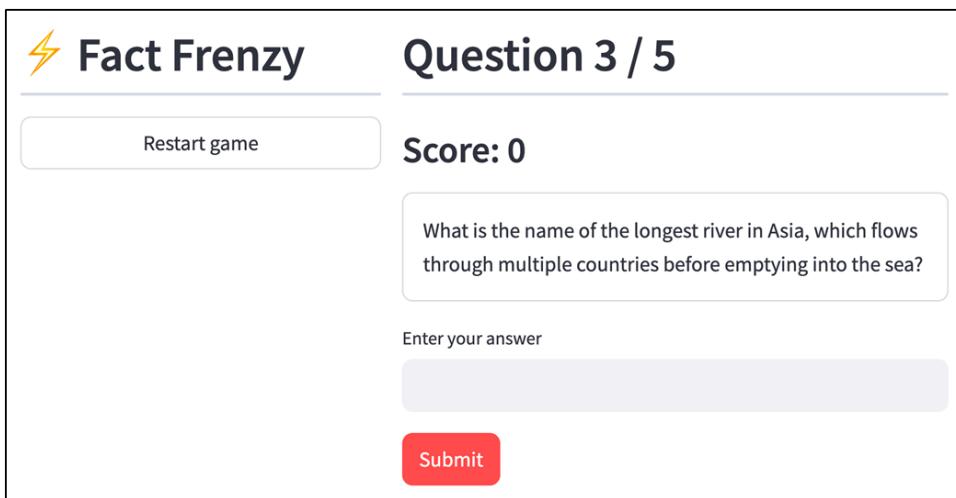


Figure 9.12 A working multi-question trivia game (see chapter_09/in_progress_05 in the GitHub repo for the full code)

Our trivia game is technically complete now, but let's see if we can make it more engaging in the next section.

COST CONSIDERATIONS

LLMs are an incredible general-purpose tool, but it is important to realize that using them in your app—especially in production where it may be accessed by hundreds of users—is not free. The last thing you want is to accidentally run up a huge bill.

How cost is calculated

When using an LLM, costs are typically calculated based on the number of *tokens* processed. As discussed in section 9.5.1, a "token" represents a chunk of text—a word or a part of a word—that the model reads (input) or generates (output).

OpenAI charges people based on the sum of the input and output tokens processed. This means that the size of the input prompt and the size of the output text *both* contribute to the cost. The pricing also differs by model. At the time of writing, the model we've been using in this chapter—gpt-4o-mini—costs 15 cents for every 1 million tokens processed.

You can use tools like <https://platform.openai.com/tokenizer> to count the tokens in a piece of text and determine costs.

Cost optimization strategies

There are several ways in which you could optimize cost while working with LLMs. Here are a few ideas:

- Keep your input prompts short and to-the-point to reduce costs associated with input tokens.
- Reduce the size of the output text, either by instructing the LLM to keep its response short, or by explicitly restricting the number of tokens processed to a maximum value (e.g. by passing a value to the `max_tokens` argument while calling the OpenAI endpoint).
- Batch together multiple requests to reduce the total number of prompts sent to the LLM. For example, instead of providing a list of previously asked questions each time we need a new question—as we’re doing here—we could simply ask the LLM to generate the total number of questions we want in one go.
- Use less capable but cheaper models for some of your prompts. In our case, we’re using `gpt-4o-mini`, which strikes a pretty good balance of cost and intelligence, but depending on your application, it may be possible to use even cheaper models for less complex tasks. Familiarize yourself with OpenAI’s pricing page.
- Avoid LLM costs altogether by asking the user to provide their *own* LLM API key. For our game, this involves a major hit to user experience as it requires players to create an OpenAI account before they can play, but you’re guaranteed to not have to pay a dime in LLM-related costs.

9.7 Adding AI personalities

As an informative trivia game, Fact Frenzy works perfectly fine now. The end-to-end flow—from starting a new game to cycling through the questions and keeping score until the game ends—has been established.

However, it still lacks a certain *je ne sais quoi*—it’s rather dry and mechanical. Wouldn’t it be cool if we could give our game a personality? Fortunately, this is exactly the kind of thing that LLMs are great at. We could, for instance, ask GPT-4o to mimic the style of various characters while asking questions.

In fact, we could let players choose what character they want their quizmaster to take. Sound fun? Let’s get to it!

9.7.1 Adding game settings

We don’t currently have a page or place in the app where players can view or change any settings, so we’ll tackle that first.

What options do we want the user to be able to set? We’ve already talked about the quizmaster speaking style, so that can be the first. We could also let the player pick a difficulty level that suits them.

There are several different designs we could use for a "settings editor", but I want to use this opportunity to introduce a handy Streamlit widget we haven't encountered before: `st.data_editor`.

ST.DATA_EDITOR

In Chapters 6 and 7, we learned about Pandas dataframes, which make working with tabular data in Python easy. We discovered `st.dataframe`, used to render dataframes as a table in a Streamlit app for viewing.

`st.data_editor` displays dataframes too, but also makes them *editable*, providing users with an experience that you might expect in a spreadsheet.

What does this have to do with adding settings to our app? Well, we could place the settings we want in a dataframe, and enable people to edit the dataframe to modify a setting.

With the two settings we discussed a few paragraphs above, the settings dataframe might look like the following:

Quizmaster	Difficulty
Alex Trebek	Medium

If this dataframe is stored in `default_settings`, we might write our `st.data_editor` widget like so:

```
settings = st.data_editor(default_settings, num_rows='fixed', key='settings_editor')
```

This would display the widget shown in figure 9.13.

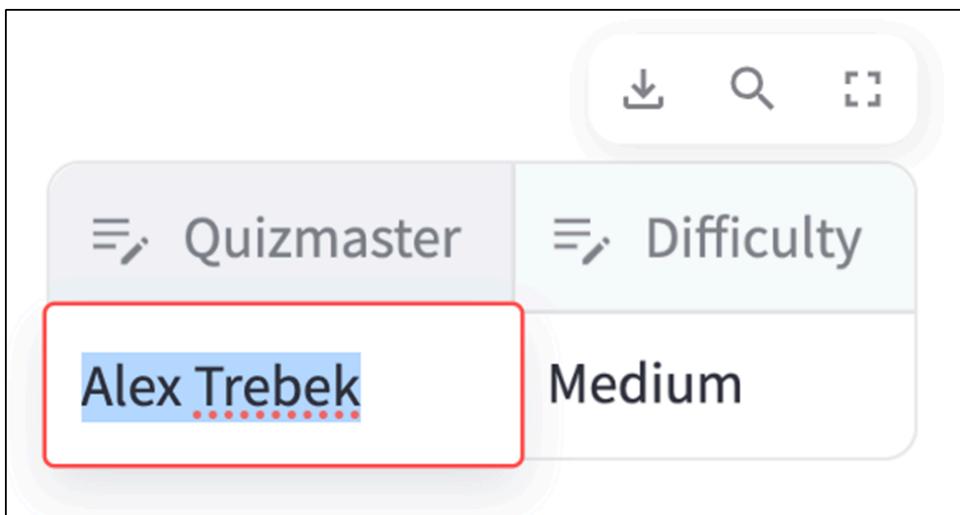


Figure 9.13 A simple output of `st.data_editor`

The first argument here is the initial state of the data we want to edit—in this case, the default settings.

The `num_rows='fixed'` means that the data editor widget shouldn't allow users to add any new rows. This makes sense because we don't want to have multiple rows in the dataframe shown above—a single setting can only have one value.

In any run of the app that happens before the user interacts with the data editor, `settings` will hold the same value as `default_settings`. Once the user changes a setting—for example, they might change `Difficulty` to `Easy`—`settings` will hold the edited dataframe across future re-runs until the user edits it again.

NOTE The `key='settings_editor'` parameter adds a `widget` key to the session state for the data editor. While this isn't strictly required for the app to function correctly, it protects us from a certain quirk of Streamlit where it forgets the values of a widget without an explicit key between re-runs if that widget isn't rendered for some reason in a particular run. Adding a widget key doesn't cost us anything, so it's safer to provide one to avoid unforeseen bugs.

Getting back to our example, we don't necessarily want users to have to type in the name of the quizmaster or a difficulty level; we'd rather have them select from a list of options. `st.data_editor` supports this in the form of *column configurations*:

```
st.data_editor(
    default_settings,
    column_config={
        'Quizmaster': st.column_config.SelectboxColumn(
            options=['Alex Trebek', 'Eminem', 'Gollum'],
            required=True
        ),
        'Difficulty': st.column_config.SelectboxColumn(
            options=['Easy', 'Medium', 'Difficult'],
            required=True
        )
    },
    num_rows='fixed',
    key='settings_editor'
)
```

Here we exert more granular control over the editable data, configuring each column in the data through `st.column_config`.

For each of our columns, we use a `SelectBoxColumn` which lets us specify a list of options to choose from, and whether a value must be specified (the `required` parameter, set to `True` above).

The results are shown in figure 9.14.

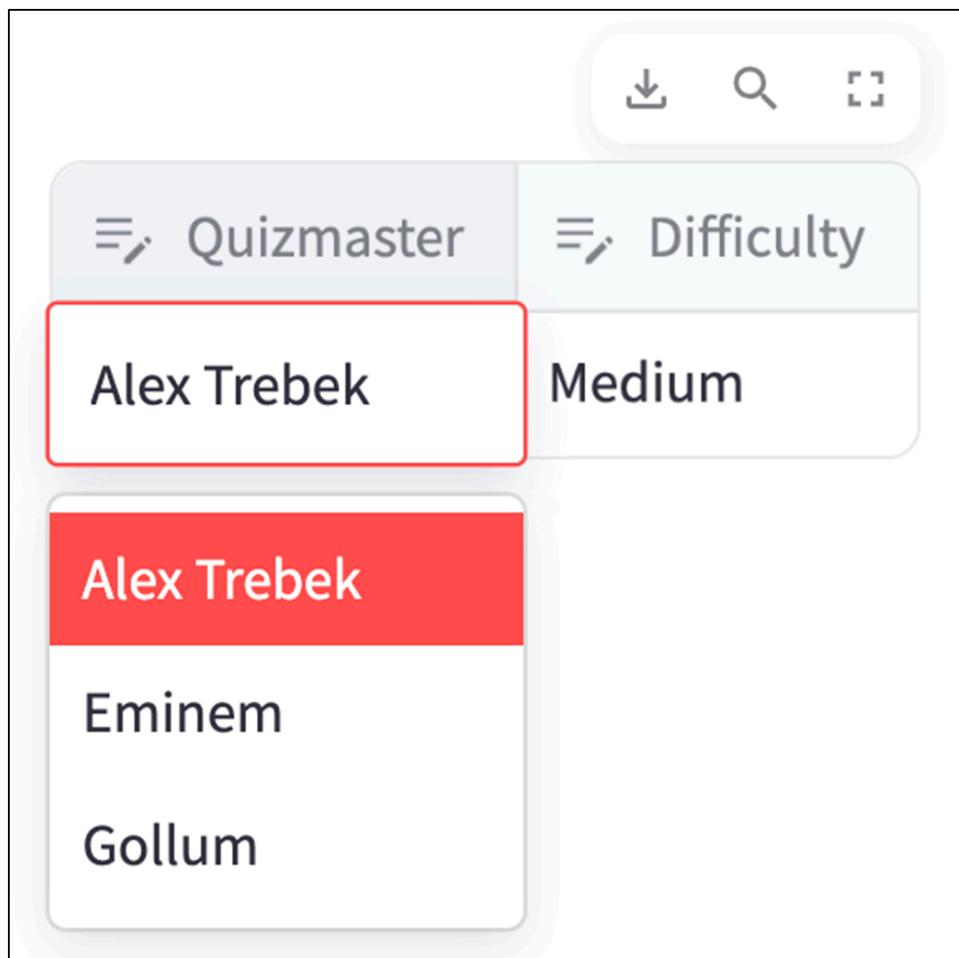


Figure 9.14 st.data_editor with one of the columns showing a `SelectBoxColumn`

`st.column_config` supports many different column types besides `SelectBoxColumn`, such as a `CheckboxColumn` for boolean values, a `DatetimeColumn` that displays a date/time picker, and a `LinkColumn` for clickable URLs.

It also supports non-editable types that can be used with `st.dataframe`, including `AreaChartColumn`, `BarChartColumn`, a `ListColumn` for lists, and even a `ProgressColumn` for numbers (displayed in a progress bar versus a target).

CREATING THE SETTINGS EDITOR

Now that we know how `st.data_editor` works, let's go ahead and create a settings editor UI. We'll put this in a new file called `settings.py` (shown in listing 9.8).

Listing 9.8 settings.py

```

import streamlit as st

QM_OPTIONS = ["Alex Trebek", "Eminem", "Gollum", "Gruk the Caveman"]
DIFFICULTY_OPTIONS = ["Easy", "Medium", "Hard"]

default_settings = {
    "Quizmaster": [QM_OPTIONS[0]],
    "Difficulty": [DIFFICULTY_OPTIONS[1]]
}

def settings_editor():
    with st.popover("Settings", use_container_width=True):
        return st.data_editor(
            default_settings,
            key='settings_editor',
            column_config={
                'Quizmaster': st.column_config.SelectboxColumn(
                    options=QM_OPTIONS, required=True),
                'Difficulty': st.column_config.SelectboxColumn(
                    options=DIFFICULTY_OPTIONS, required=True)
            },
            num_rows='fixed',
            use_container_width=True,
        )

```

(chapter_09/in_progress_06/settings.py in the GitHub repo)

We place the options for the Quizmaster and Difficulty settings right at the top for easy access, listing them under `QM_OPTIONS` and `DIFFICULTY_OPTIONS`.

The Quizmaster options range from an actual quizmaster, the late Alex Trebek of *Jeopardy!* fame, to a range of fictional characters like Gollum from *The Lord Of The Rings*, and Gruk the Caveman, an entirely made-up figure to let the LLM go wild.

We've initialized `default_settings` thus:

```

default_settings = {
    "Quizmaster": [QM_OPTIONS[0]],
    "Difficulty": [DIFFICULTY_OPTIONS[1]]
}

```

Note how this *isn't* a dataframe as we suggested initially—it's a dictionary instead, with the name of each setting as a key, and a one-element list containing the default option for that setting as the corresponding value.

Interestingly, `st.data_editor` can display things that are not Pandas dataframes. This includes native Python types such as dictionaries, lists, and sets. The quality of being able to display these types even applies to `st.dataframe`, despite the name. In this case, it means we don't actually have to maintain the settings as a dataframe; we can use the more readable dictionary form above.

The `settings_editor` function renders the actual settings editor UI. We place everything within yet another new Streamlit widget called `st.popover`:

```
with st.popover("Settings", use_container_width=True):
    ...
```

`st.popover` displays a popover widget, which is a small pop-up screen that you can trigger by clicking an associated button—in a similar manner to `st.expander`. The first argument is the label for the button that triggers the `st.popover`.

The contents of the popover are written within the `with st.popover(...)` context manager. In this case, we're displaying the `st.data_editor` widget and returning its value, i.e. the edited `settings` dictionary:

```
return st.data_editor(
    default_settings,
    key='settings_editor',
    column_config={
        'Quizmaster': st.column_config.SelectboxColumn(
            options=QM_OPTIONS, required=True),
        'Difficulty': st.column_config.SelectboxColumn(
            options=DIFFICULTY_OPTIONS, required=True)
    },
    num_rows='fixed',
    use_container_width=True,
)
```

This is pretty much the same code that we wrote in the previous section while we were discussing `st.data_editor`, though with the addition of a `use_container_width=True` argument, which adjusts the width of the popover.

APPLYING THE SETTINGS

How do we use these settings in Fact Frenzy? Both the Quizmaster and Difficulty settings relate to the question text generated by the LLM, so let's incorporate them in the question prompt in `prompts.py`, which becomes:

```

QUESTION_PROMPT = {
    'system': """
        You are a quizmaster who mimics the speaking style of {quizmaster} and
        never asks the same question twice.

    """,
    'user': """
        First think of a unique category for a trivia question.
        Then think of a topic within that category.

        Finally, ask a unique trivia question that has a difficulty rating of
        {difficulty} and is generated using the random seed {seed}, without
        revealing the category or topic.

        Do not provide choices, or reveal the answer.

        The following questions have already been asked:
        {already_asked}
    """
}

...

```

(chapter_09/in_progress_06/prompts.py in the GitHub repo)

Obviously, there are plenty of ways to work our two new variables into the prompt—the above is just one.

We'll modify game.py next:

```

...
class Game:
    def __init__(self, llm_api_key, settings):
        self.llm = Llm(llm_api_key)
        self.settings = settings
        self.status = 'GET_QUESTION'
        ...

    def get_setting(self, setting_name):
        return self.settings[setting_name][0]

    def modify_settings(self, new_settings):
        self.settings = new_settings

    def ask_llm_for_question(self):
        seed = int(time.time())
        sys_msg = (

```

```

    QUESTION_PROMPT['system']
    .replace('{quizmaster}', self.get_setting('Quizmaster'))
)
usr_msg = (
    QUESTION_PROMPT['user']
    .replace('{already_asked}', '\n'.join(self.questions))
    .replace('{seed}', str(seed))
    .replace('{difficulty}', self.get_setting('Difficulty'))
)
return self.llm.ask(usr_msg, sys_msg)

def ask_llm_to_evaluate_answer(self):
    ...
    ...

```

(chapter_09/in_progress_06/game.py in the GitHub repo)

Game's `__init__` now accepts a `settings` parameter, which—as you'd expect—is in the dictionary format we used in `settings.py`. This is assigned to `self.settings` so other methods can access it.

We've added two associated methods: `get_setting` and `modify_settings`. `get_setting` deals with getting the value of a given setting, which is slightly tricky because each dictionary value is a single-element list (designed that way so it works with `st.data_editor`). `get_setting` abstracts away this somewhat unsightly logic so we restrict it to one place in the code.

`modify_settings` replaces `self.settings` with a given `new_settings` dictionary. This will come into play when the user changes a setting.

Turning to the `ask_llm_for_question` method, we replace the `{quizmaster}` and `{difficulty}` variables we added to the prompt with their corresponding values from the `settings`, obtained through `get_setting`.

The changes to `main.py` are all that remain now, so let's make those:

```

import streamlit as st
from game import Game
from settings import default_settings, settings_editor

def start_new_game():
    st.session_state.game = Game(st.secrets['llm_api_key'], default_settings)
    st.rerun()

...
with side_col:
    st.header(" Fact Frenzy", divider='gray')
    settings = settings_editor()
    new_game_button(game)

with main_col:
    if game:
        game.modify_settings(settings)
        st.header(
            ...
            ...

```

(chapter_09/in_progress_06/main.py in the GitHub repo)

In `start_new_game`, to obtain the initial `Game` instance, we now pass in `default_settings`, directly imported from `settings.py`.

The actual settings editor is displayed within the side column (`side_col`), right above the New Game button. The return value—recall that this would be the `default_settings` dictionary before the user changes the value of any settings, and the modified dictionary afterward—is stored in the `settings` variable.

And, finally, in every re-run—provided that we're in a game—we run `game.modify_settings(settings)` to pick up any changes the user has made to the `settings`.

That should do it. Run your app again to see figure 9.15. Play around with the AI quizmaster options and difficulties; it's now more fun to read the questions!

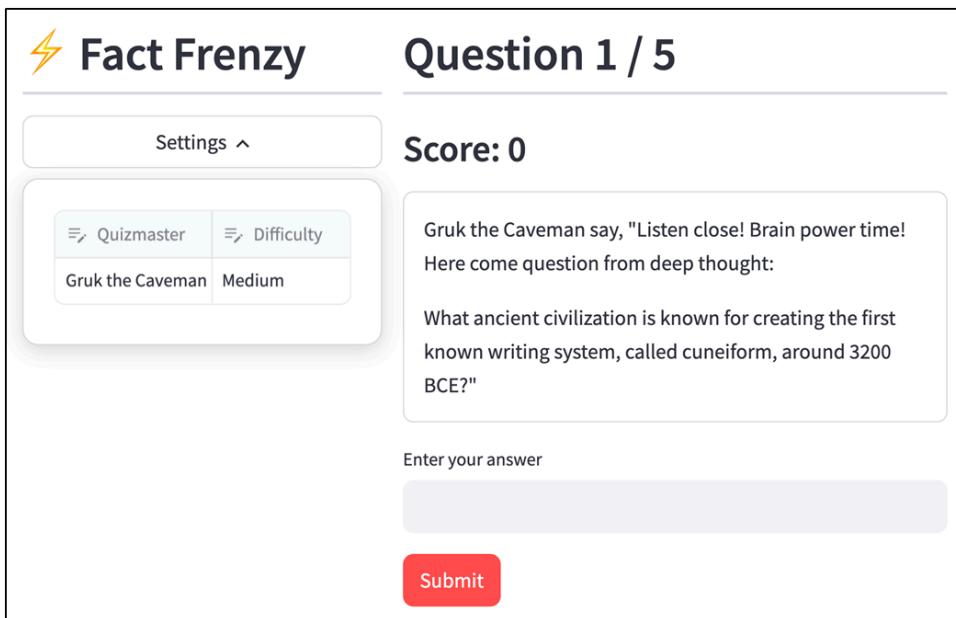


Figure 9.15 A final version of Fact Frenzy with editable settings (see chapter_09/in_progress_06 in the GitHub repo for the full code)

That concludes our development of Fact Frenzy, the first—and only—game we'll build in this book. In Chapter 10, we'll continue our exploration of AI apps with a more practical application: a customer support chatbot.

9.8 Summary

- A Large Language Model (LLM) is an AI system designed to process and generate human-like text.
- LLMs can perform both creative and analytical tasks.
- OpenAI, one of the most popular LLM providers, allows developers to access its GPT series through an API.
- The `openai` library provides a convenient way to call the OpenAI API in Python.
- You can pass a conversation to OpenAI API's chat completion endpoint with messages tagged as "system", "assistant" or "user", causing the model to complete the conversation in a logical way.
- Structured Outputs is a feature provided by OpenAI that ensures the model will generate a response that adheres to a given schema.
- A common pattern in linear Streamlit apps is to implement conditional branching logic based on a variable that's held in `st.session_state`.
- You can vary parameters such as `temperature` and `top_p` to affect the creativity and predictability of responses generated by an LLM.

- Injecting randomness into the prompt is a good way to ensure we get different responses to similar prompts.
- It's important to optimize cost in LLM-based applications. You can do this by having the LLM process fewer input and output tokens, reducing the number of prompts, using cheaper models, or even having users bear the cost by requiring them to supply their own API key.
- `st.data_editor` provides a way to create editable tables in Streamlit apps.
- `st.column_config` lets you configure columns to be of certain editable and non-editable types in `st.data_editor` and `st.dataframe`.
- `st.popover` displays a small pop-up screen triggered by a click on an associated button.

10 A customer support chatbot with LangGraph and Streamlit

This chapter covers

- Developing a chatbot frontend with Streamlit's chat elements
- Using LangGraph and LangChain to streamline an advanced AI app
- How embeddings and vector databases work
- Augmenting an LLM's pre-trained knowledge using Retrieval Augmented Generation (RAG)
- Enabling an LLM to access and execute real-world actions

Creating a fun and engaging experience—like the trivia game we built in chapter nine—is exciting, but the true power of AI lies in its ability to drive real business value. AI isn't just about answering questions or generating text; it's about transforming industries, streamlining operations, and enabling entirely new business models.

However, building AI applications that deliver economic value requires more than just calling a pre-trained model. For AI to be useful in real-world scenarios, it needs to be aware of the context in which it operates, connect to external data sources, and take meaningful actions. Companies need AI to understand and respond to domain-specific queries, interact with business systems, and provide personalized assistance.

In this chapter, we'll build such an application: a customer service chatbot that will retrieve real company data, help customers track and cancel orders, and intelligently decide when to escalate issues to a human agent. By the end of this chapter, you'll understand how to integrate LLMs with private knowledge bases, implement retrieval-augmented generation (RAG), and enable an AI agent to take action in the real world. Let's dive in.

NOTE The GitHub repo for this book is <https://github.com/aneevdavis/streamlit-in-action>. The chapter_10 folder has this chapter's code and a requirements.txt file with exact versions of all the required Python libraries.

10.1 Nibby: A customer service bot

Under the leadership of Note n' Nib's new CEO—renowned for his legendary decision-making prowess, aided by a certain dashboard revered across the company—the brand has flourished into a stationery powerhouse with rocketing sales.

But success brings its own challenges. The customer support department is swamped with calls from buyers who are impatient for their orders to arrive or seeking advice about fountain pen maintenance. After a month of complaints about long wait times, the CEO summons the one person known company-wide for reliable innovation.

And so it is that *you* are tasked with solving the support crisis. When you're not delivering seminars on Streamlit, you're reading up on the latest advances in AI; it is not long before an intriguing possibility hits you: might it be possible to *automate* customer support?

Over the course of a sleepless night, you sketch out plans for a Streamlit support bot named Nibby. Whispers of your project spread across the company. "We are saved!" some declare. "Nibby will not fail us!" Skeptics scoff: "'Tis folly! No *robot* can fix this."

Who will prove right? Let's find out.

10.1.1 Stating the concept and requirements

As always, we start with a distilled one-line description of what we intend to build.

CONCEPT Nibby, a customer support chatbot that can help Note n' Nib's customers with information and basic service requests

"Customer support" obviously spans a lot of territory, so let's define the exact requirements more clearly.

REQUIREMENTS

In our vision of automating customer support, Nibby will be able to:

- hold a human-like conversation with a customer
- answer relevant questions about Note n' Nib and its products based on a custom knowledge base
- handle the following requests from the customer:
 - tracking an order
 - canceling an order
- redirect to a human customer support agent if it cannot fulfill the request on its own

In essence, we want Nibby to take as much load off Note n' Nib's overworked human support agents as possible. Nibby should act as a "frontline" agent who can take care of most basic requests, such as providing product information or canceling orders, and redirect to a human only when necessary.

WHAT'S OUT OF SCOPE

To keep this project manageable and small enough to fit into this chapter, we'll decide *not* to implement the following:

- Storing or remembering prior conversations with a user
- Any "actions" other than the two provided above, i.e., tracking and canceling orders
- Actual working logic for the two actions discussed, e.g., building an order tracking or cancellation system

From a learning perspective, we truly want to focus on building a relatively complex AI system that can converse with users, understand a custom knowledge base, and take real-world actions.

The specific actions we enable the app to perform don't matter. For instance, the fact that our app can cancel an order—as opposed to replacing an item—is not of any particular significance. Indeed, as implied by the third point above, the order cancellation we will implement is dummy "toy" logic. What *is* significant is that our bot should be able to intelligently choose to run that logic based on the free-form conversation the user is having with it.

10.1.2 Visualizing the user experience

The user interface for Nibby might be the most straightforward among all the apps in this book. Figure 10.1 shows a sketch of what we'll build.

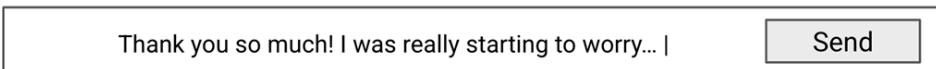
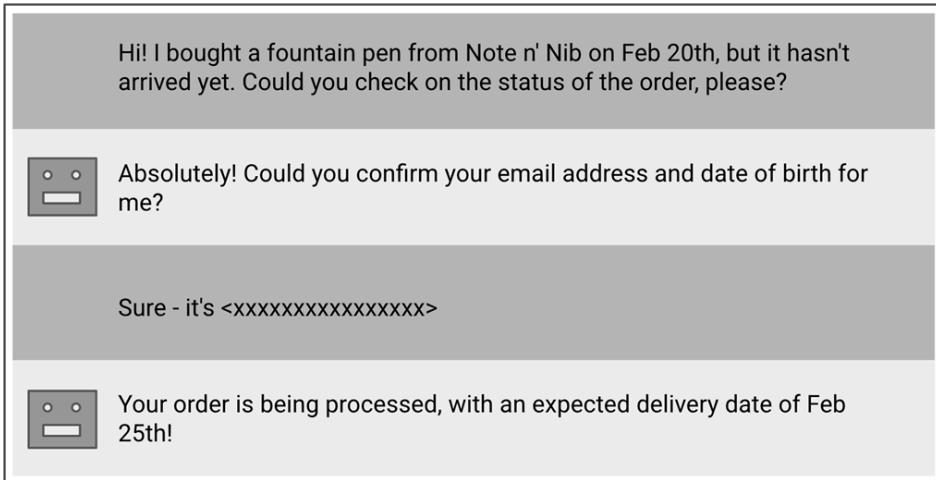


Figure 10.1 UI sketch for Nibby, our customer support chatbot.

Nibby's UI isn't significantly different from any chat or instant messaging app you may have used—from WhatsApp on your phone to Slack on your corporate laptop. You'll notice a familiar-looking text box at the bottom for users to type messages. Each user message triggers an AI response, which is appended to the conversation view above.

10.1.3 Brainstorming the implementation

The difficult part of building this app will be the backend—specifically, getting the bot to answer questions correctly and connect to outside tools. Figure 10.2 shows our overall design.

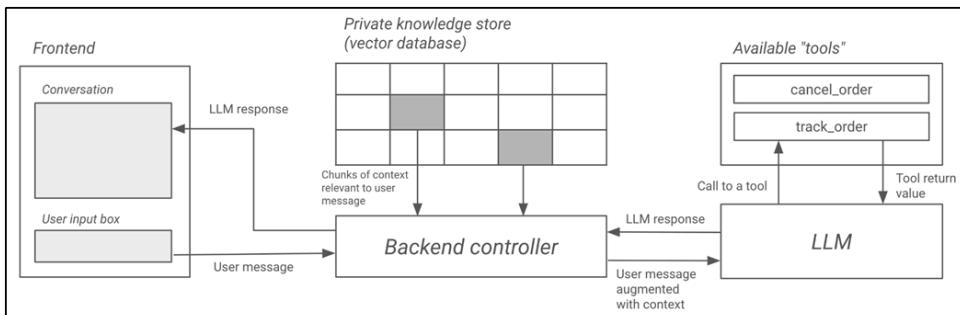


Figure 10.2 Overall design for Nibby

While our trivia app from chapter 9 had an interesting design when it came to state management, its "intelligent" aspect was fairly simple—feed a prompt to an LLM and have it respond.

On the other hand, a customer support app with the capabilities we're envisioning has a more involved design in at least two respects:

- It needs a way to augment a customer's query with private knowledge about the company that a human agent would possess.
- It needs to be able to execute code in the real world.

Figure 10.2 gives a basic overview of how we achieve these. When a user message comes in through our frontend, we retrieve context relevant to the message from a private knowledge store—a vector database, as we'll see later—and send that to the LLM.

We also organize the actions we want our bot to be able to take into so-called *tools* and make the LLM aware of their existence, what each tool does, and how to call them. For any given input, the LLM can either issue a call to a tool or respond to the user directly.

In the former case, we execute the tool as specified by the LLM and send the results for further processing, while in the latter case, we append the response to a conversation view on the frontend and await the user's next message.

10.1.4 Installing dependencies

We'll be using several Python libraries in this chapter. To get everything ready in advance, install them all at once by running the following command:

```
pip install langchain-community langchain-core langchain-openai langchain-pinecone
langgraph pinecone toml
```

10.2 Creating a basic chatbot

Chapter nine introduced LLMs and demonstrated how to use the OpenAI API for simple applications. While OpenAI's API is easy to integrate, developing more sophisticated AI-driven apps—such as those leveraging Retrieval-Augmented Generation (RAG) or agent-based workflows, which we'll encounter soon—adds complexity.

A new ecosystem of libraries and tools has emerged to make creating complex AI apps as easy as possible. In this chapter, we'll explore LangGraph and LangChain, two libraries that work together to smooth the application creation process.

10.2.1 Intro to LangGraph and LangChain

LLMs have undoubtedly been the most influential technological advance of the last decade. At their core, interacting with an LLM consists of providing a prompt that the LLM can "complete." That's what everything else is built around.

Contrast this with the complexities that modern AI applications have to deal with:

- Handling multi-step workflows (e.g., retrieving information before responding)
- Integrating with external tools
- Retaining conversation context across multiple turns

Managing this complexity manually is difficult, which is where LangChain and LangGraph—both Python libraries—come in. LangChain provides building blocks for working with LLMs, including prompt management, memory, and tool integration. LangGraph—developed by the same company—extends LangChain by structuring AI workflows as *graphs*, allowing for decision-making, branching logic, and multi-step processing. By combining these, we can design structured, intelligent AI applications that go beyond simple chat responses—enabling Nibby to retrieve knowledge, call APIs, and make decisions dynamically.

In the rest of this chapter, we'll use these libraries extensively to achieve the functionality we want.

NOTE Since we will model our chatbot as a *graph* in LangGraph, we'll primarily speak about and refer to LangGraph rather than LangChain. However, you'll notice that many of the underlying classes and functions we'll use are imported from LangChain.

10.2.2 Graphs, nodes, edges, and state

In LangGraph, you construct an AI application by building a *graph* of *nodes* that transform the application's *state*. If you don't have a background in computer science, that statement might trip you up, so let's break it down.

WHAT IS A GRAPH, EXACTLY?

In graph theory, a graph is a network of interconnected vertices (also called nodes) and edges, which connect the vertices with each other. Software developers often use graphs to create conceptual models of real-world objects and their relationships. For instance, figure 10.3 shows a graph of people you might expect to find on a social media website.

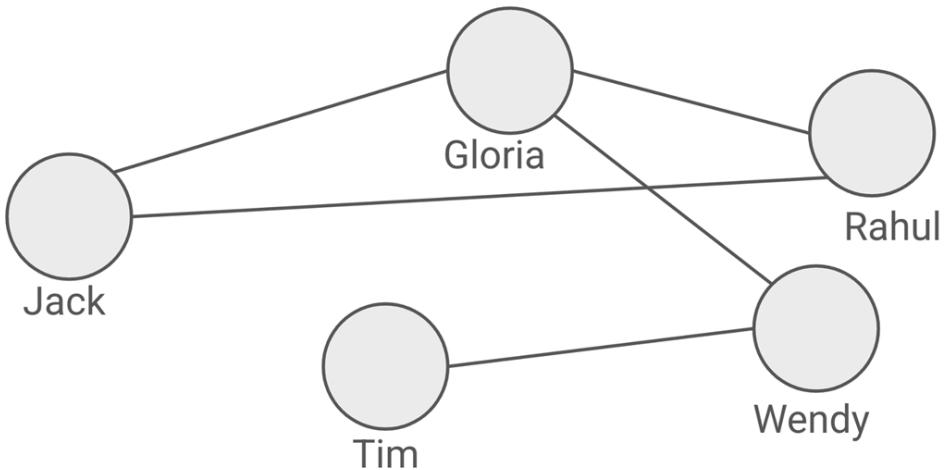


Figure 10.3 Using a graph to model friend relationships in a social network

Here, each person is a vertex or node (shown by a circle), and the "friend"-relationship between any two people is an edge (the lines between the circles).

By modeling relationships in this way, the social media website can apply various algorithms developed for graphs to do useful real-world things. For instance, there's an algorithm called breadth-first search (BFS) that finds the shortest path from one node to any other node. In this case, we could use it to find the fewest common friends required to connect two people.

In LangGraph, we model an application as a graph of *actions*, where a *node* signifies a single action that the application performs. A graph has a *state*, simply a collection of named attributes with values (similar to Streamit's concept of "session state"). Each node takes the current state of the graph as input, does something to modify the state, and returns the new state as its output.

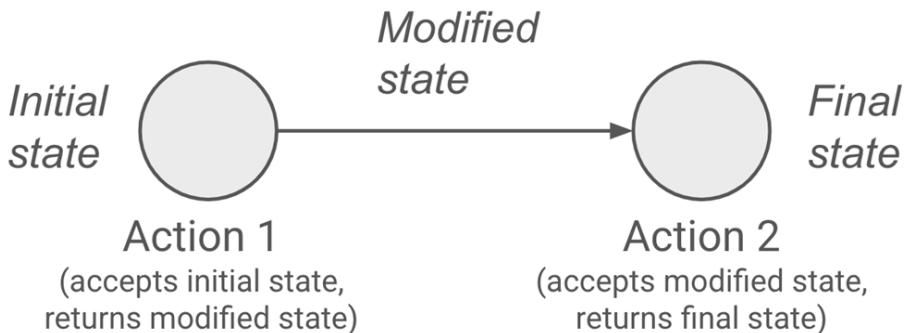


Figure 10.4 In LangGraph, nodes take in the graph state and modify it.

An edge in the graph represents a connection between two nodes, or in other words, the fact that the output of one node may be the input to another. Unlike in the case of a social media graph where the edges had no direction (i.e., if two people are friends, each is a friend of the other), edges in LangGraph are *directed* because one node in the edge is executed *before* the other. Visually, we represent the direction as an arrow on the edge (figure 10.4).

The input to the graph is its initial state that is passed to the *first* node that's executed, while the output is the final state returned by the *last* node that's executed.

That was a fair bit of theory; let's now consider a toy example (figure 10.5) to make this all real.

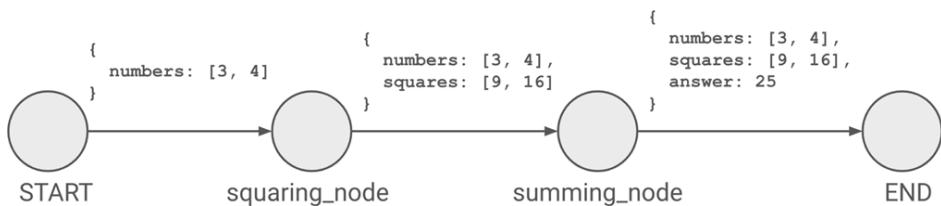


Figure 10.5 A graph in LangGraph that computes the sum of squares of numbers

The application shown in figure 10.5 is a very simple one. There's no AI involved; it's just a program that takes a list of numbers and returns the sum of their squares—e.g., for the input [3, 4], the graph would calculate the output 25 ($3^2 + 4^2 = 25$)

The graph's state contains three values: `numbers`, `squares`, and `answer`. `numbers` holds the list of input values (e.g. [3, 4]), while `squares` and `answer` don't have a value to start with.

Each LangGraph graph has dummy nodes called `START` and `END`, which represent the start and end of execution. There are two other "real" nodes: `squaring_node` and `summing_node`.

Here's how the graph is executed:

- The `START` node receives the initial state.
 - Since there's a directed edge from `START` to `squaring_node`, `squaring_node` is executed first.
 - `squaring_node` takes in the starting state, squares the numbers in the `numbers` list and saves the new list ([9, 16]) under the variable `squares` in the state.
 - As there's an edge from `squaring_node` to `summing_node`, `summing_node` takes as input this modified state returned by `squaring_node`.
 - `summing_node` adds up the numbers in `squares`, and saves the result as `answer`.
 - `summing_node` has an edge to `END`, which means the end of execution.
- The final state returned will contain 25 under `answer`.

Of course, this is a simple graph with only one path the execution can take. In a later part of this chapter, you'll encounter a graph with multiple paths—where a single node may branch into multiple nodes based on the state at that point.

I hope this helped crystallize the concept of graphs and how LangGraph uses them to perform a task. It's now time to use what we've learned to start building our app.

10.2.3 A one-node LLM graph

The basic graph we built in the previous section had nothing to do with AI or LLMs. Indeed, you can use LangGraph to build anything you like, whether or not AI is involved, but in practice, the point of LangGraph is to make building AI applications easier.

CREATING AND RUNNING YOUR FIRST GRAPH

In chapter nine, we encountered the OpenAI API's chat completions endpoint. In this endpoint, we pass a list of messages to the API, which predicts the next message in the conversation. In LangGraph, such an application could be represented by a simple one-node graph, as shown in figure 10.6.

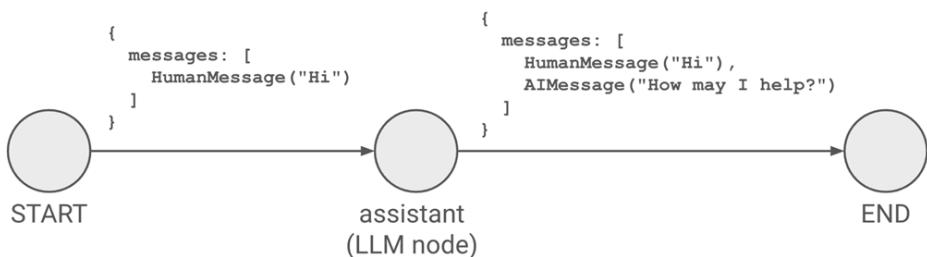


Figure 10.6 A basic single-node (apart from START and END) graph

The state of the graph consists of a single variable, `messages`, which is—as you might expect—a list of messages.

The only node in the graph, `assistant_node`, passes `messages` to an LLM, and returns the same list with an AI response message appended.

Listing 10.1 shows this graph translated to real (non-Streamlit) Python code.

Listing 10.1 graph_example.py

```

from langgraph.graph import START, END, StateGraph
from langchain_core.messages import AnyMessage, HumanMessage
from langchain_openai import ChatOpenAI
from typing import TypedDict

llm = ChatOpenAI(model_name="gpt-4o-mini", openai_api_key="sk-proj-...")

class MyGraphNode(TypedDict):
    messages: list[AnyMessage]

builder = StateGraph(MyGraphNode)

def assistant_node(state):
    messages = state["messages"]
    ai_response_message = llm.invoke(messages)
    return {"messages": messages + [ai_response_message]}

builder.add_node("assistant", assistant_node)
builder.add_edge(START, "assistant")
builder.add_edge("assistant", END)
graph = builder.compile()

input_message = input("Talk to the bot: ")
initial_state = {"messages": [HumanMessage(content=input_message)]}
final_state = graph.invoke(initial_state)

print("Bot:\n" + final_state["messages"][-1].content)

```

(chapter_10/graph_example.py in the GitHub repo)

First, we initialize the LLM in this line:

```
llm = ChatOpenAI(model_name="gpt-4o-mini", openai_api_key="sk-proj-...")
```

This is similar to what we did in chapter 9 when we created an OpenAI API client. LangChain—a library closely related to LangGraph—provides a class called `ChatOpenAI` that does essentially the same thing but is slightly easier to use. As before, don't forget to replace `sk-proj...` with your actual OpenAI API key.

Consider the next part:

```
class MyGraphState(TypedDict):
    messages: list[AnyMessage]
```

As we've discussed, a graph has a state. For each graph you define, you specify what fields exist in the state by creating a class that contains those fields.

In the above two lines, we're creating `MyGraphState` to represent the state of the graph we're about to define. In line with the example in figure 10.6, `MyGraphState` contains one field—`messages`—which is a list of objects of the type `AnyMessage`.

In chapter nine, we saw that each message in an (OpenAI) LLM conversation has a *role*—one of "user", "assistant", or "system". In `LangChain`, the same concept is represented by the `AnyMessage` superclass. `HumanMessage`, `AIMessage`, and `SystemMessage` are subclasses that inherit functionality from `AnyMessage`, and correspond to the "user", "assistant", and "system" roles, respectively.

`MyGraphState` itself is a subclass of `TypedDict`, a specialized dictionary type from Python's typing module that allows us to define a dictionary with a fixed set of keys and associated value types. Since it inherits all of `TypedDict`'s behaviors, we can treat instances of `TypedDict`—and therefore, `MyGraphState`—as regular dictionaries, using the same syntax for accessing its keys (i.e. the fields in the class) and values.

NOTE We don't technically *have to* use a `TypedDict` to represent the state of the graph. We could also have used a regular class, a dataclass, or a Pydantic `BaseModel`, which we used in chapter nine. We've introduced and used `TypedDict` here because it plays well with `MessagesState`, a built-in `LangGraph` class that we'll discuss shortly.

The next line, `builder = StateGraph(MyGraphState)`, initializes the construction of our graph. Here we're telling `LangGraph` that we're building a `StateGraph`—the type of graph we've been talking about where the nodes read from and write into a shared state—whose state is represented by a `MyGraphState` instance (which, as we've seen, will have a `messages` list).

We now define the only node in our graph thus:

```
def assistant_node(state):
    messages = state["messages"]
    ai_response_message = llm.invoke(messages)
    return {"messages": messages + [ai_response_message]}
```

Each node in a `LangGraph` graph is a regular Python function that accepts the current state of the graph—a `MyGraphState` instance—as input, and returns the parts of the state it wants to modify.

The `assistant_node` we've defined above is quite minimal; it simply passes the messages list—accessed using square brackets as `state["messages"]` just like in a regular dictionary—to the `invoke` method of `llm`, obtaining the AI's response message. It then modifies the "messages" key of the state, adding `ai_response_message` to the end, and returns the result.

NOTE In the above code, since `MyGraphState` only has a single key, `messages`, it seems that `assistant_node` is simply returning the entirety of the modified state. That's not strictly true—it's actually only returning the keys it wants to modify, leaving any other keys untouched. This will become clear in later sections.

Now that we've created our only node, it's time to put it in our graph:

```
builder.add_node("assistant", assistant_node)
builder.add_edge(START, "assistant")
builder.add_edge("assistant", END)
```

The first line adds a node called `assistant` to the graph, pointing to the `assistant_node` function we just developed as the logic for the node.

As mentioned earlier, each graph has dummy `START` and `END` nodes. The remaining two lines create directed edges from `START` to our `assistant` node, and from our `assistant` node to `END`, thus completing the graph.

The immediately following line, `graph = builder.compile()`, compiles the graph, readying it for execution.

The last few lines of code in the file show how a graph can be invoked:

```
input_message = input("Talk to the bot: ")
initial_state = {"messages": [HumanMessage(content=input_message)]}
final_state = graph.invoke(initial_state)

print("The LLM responded with:\n" + final_state["messages"][-1].content)
```

We first use the `input()` function—which prompts the user to enter something in the terminal—to collect the user's input message.

We then construct the starting state of the graph as a dictionary with the key "messages." The message itself is an instance of `HumanMessage` with its `content` attribute set to the `input_message` we just collected.

Passing `initial_state` to the graph's `invoke` method finally causes the graph to execute, effectively passing our user input to the LLM through `assistant_node`, returning the final state.

`final_state` contains all of the messages in the conversation so far (our user message and the LLM's response message), so we access the response message using `final_state["messages"][-1]` and print its content to the screen.

To see this in action, copy all the code to a new file called `graph_example.py`, and run your code in the terminal using the `python` command like this:

```
python graph_example.py
```

Enter a message when you see the "Talk to the bot" prompt. As an example output, here's what I got:

```
$ python graph_example.py
Talk to the bot: Howdy! Could you write a haiku about Note n' Nib for me?
Bot:
Ink and paper dance,
Whispers of thoughts intertwine—
Note n' Nib's embrace.
```

It looks like AI stole my dream haiku gig. Maybe I'll pivot to the performing arts—everyone loves a good mime.

TURNING OUR GRAPH INTO A CLASS

We've run our first graph in the terminal, but what we really want is to use it to power our customer support bot. We'll organize our code using object-oriented principles as we did in the last two chapters.

Let's start by converting the code we wrote in the prior section into a `SupportAgentGraph` class in `graph.py`, shown in listing 10.2.

Listing 10.2 graph.py

```

from langgraph.graph import START, END, StateGraph, MessagesState
from langchain_core.messages import HumanMessage

class SupportAgentGraph:
    def __init__(self, llm):
        self.llm = llm
        self.graph = self.build_graph()

    def get_assistant_node(self):
        def assistant_node(state):
            ai_response_message = self.llm.invoke(state["messages"])
            return {"messages": [ai_response_message]}
        return assistant_node

    def build_graph(self):
        builder = StateGraph(MessagesState)
        builder.add_node("assistant", self.get_assistant_node())
        builder.add_edge(START, "assistant")
        builder.add_edge("assistant", END)
        return builder.compile()

    def invoke(self, human_message_text):
        human_msg = HumanMessage(content=human_message_text)
        state = {"messages": [human_msg]}
        return self.graph.invoke(state)

```

(chapter_10/in_progress_01/graph.py in the GitHub repo)

The code here is very similar to that in `graph_example.py`, but I want to highlight a few differences.

Most obviously, we're encapsulating our graph in a class—`SupportAgentGraph`—which has a method for building the actual graph (`build_graph`) and another (`invoke`) for invoking it by passing a human (user) message.

Rather than creating the `LLM` object within the class, we accept it as a parameter in `SupportAgentGraph`'s `__init__`, which builds the graph by calling `self.build_graph()` and saves it under `self.graph` for future invocations.

You'll notice that our `MyGraphState` class, which we defined earlier, is nowhere to be found. We've swapped it out for `MessagesState`, a built-in `LangGraph` class that does more or less the same thing. `MessagesState`, like `MyGraphState`, has a `messages` field, which is a list of `AnyMessage` objects. The big difference between `MyGraphState` and `MessagesState` is how the `messages` field can be modified in a node—more on that in a second.

Next, consider the `get_assistant_node` method:

```
def get_assistant_node(self):
    def assistant_node(state):
        ai_response_message = self.llm.invoke(state["messages"])
        return {"messages": [ai_response_message]}
    return assistant_node
```

This method has a function definition—for `assistant_node`, which we encountered in the previous section—nested under it. It seems to do nothing other than return the function. What's that about?

Well, since `assistant_node` needs to access the LLM object (`self.llm`), its code must live inside a method of the `SupportAgentGraph` class. But `assistant_node` can't *itself* be a method of the class, because the first argument passed to a method is `self`—the current instance of the class—while the first (and only) argument passed to a valid `LangGraph` node must be the graph state.

So instead, we define `assistant_node` as an inner function within an outer method called `get_assistant_node`—taking advantage of the outer method's scope to access `self.llm` within the inner function—and have the outer method *return* the inner function so we can plug it into the graph. This programming pattern is called a *closure* since the inner function retains access to variables from its *enclosing* scope, even after the outer function has returned.

The aforementioned plugging-in of the node happens in the `build_graph` method in this line:

```
builder.add_node("assistant", self.get_assistant_node())
```

Since `get_assistant_node()` *returns* the `assistant_node` function (as opposed to calling it), we can use the call to `get_assistant_node` to refer to the inner function.

The `assistant_node` function differs from the one of the same name we defined in the prior section in one important way. Consider the return statement, which has changed from:

```
return {"messages": messages + [ai_response_message]}
```

to:

```
return {"messages": [ai_response_message]}
```

Why do we not return the other items in the `messages` list anymore? The answer has to do with our having replaced `MyGraphState` with `MessagesState`. You see, each node in `LangGraph`'s `StateGraph` receives the complete state as input, but the value it returns is treated as a set of *updates* to each key in the state. How exactly these updates are merged with the existing values depends on how we've specified it in our state type.

In `MyGraphState`, we didn't mention any particular way of handling this, so the value associated with the key `messages` is simply replaced by whatever the node returns for that key. This is why we needed to return the entire list—because we would have lost the earlier messages otherwise.

On the other hand, `MessagesState` internally specifies that the value returned by a node should be appended to the existing list. So, `ai_response_message` is simply tacked on to the existing messages, and we don't have to return the older messages separately.

NOTE

`MessagesState` implements this append feature through a function called `add_messages`. In fact, the only difference between `MyGraphState` and `MessagesState` is that the `messages` field in `MessagesState` is defined (internally) like this:

```
messages: Annotated[list[AnyMessage], add_messages]
```

I won't go into this in detail, but this is essentially saying that when updates occur, they should be handled by the `add_messages` function rather than a simple replacement.

Whew! That was a lot of explanation, but hopefully, you now understand how graphs are modeled in `LangGraph`.

THE BOT CLASS

Let's set aside `SupportAgentGraph` now and pivot to our main backend class, which we'll call `Bot`. `Bot` will be the single point of entry to the backend for our Streamlit frontend, similar to the `Game` and `Hub` classes in earlier chapters.

Importantly, `Bot` will supply the `LLM` object that `SupportAgentGraph` needs and provide a user-friendly method that our frontend can call to chat with the bot.

To create it, copy the code in listing 10.3 to a new file, `bot.py`.

Listing 10.3 bot.py

```
from langchain_openai import ChatOpenAI
from graph import SupportAgentGraph

class Bot:
    def __init__(self, api_keys):
        self.api_keys = api_keys
        self.llm = self.get_llm()
        self.graph = SupportAgentGraph(llm=self.llm)

    def get_llm(self):
        return ChatOpenAI(
            model_name="gpt-4o-mini",
            openai_api_key=self.api_keys["OPENAI_API_KEY"],
            max_tokens=2000
        )

    def chat(self, human_message_text):
        final_state = self.graph.invoke(human_message_text)
        return final_state["messages"][-1].content
```

(chapter_10/in_progress_01/bot.py in the GitHub repo)

Luckily, the `Bot` class is a lot more straightforward than `SupportAgentGraph`. `__init__` accepts a dictionary of API keys—which, spoiler alert, we'll supply through `st.secrets` again—before setting up the LLM object with a call to the `get_llm` method, and passing it to the `SupportAgentGraph` instance, saved to `self.graph`.

`get_llm` simply uses LangChain's `ChatOpenAI` class to create the LLM object as discussed earlier. Notice that we've added a new parameter called `max_tokens`. As you may remember from the previous chapter, tokens are the basic units of text that language models process. By setting `max_tokens=2000`, we're telling OpenAI's API to limit responses to a maximum of 2000 tokens (about 1,500 words), which helps in both cost reduction and keeping responses (relatively) concise.

The `chat` method abstracts away the complexity of dealing with graphs and states. It has a simple contract—put a human message string in, and get an AI response string out. It fulfills this promise by calling the `invoke` method of our `SupportAgentGraph`'s instance, and returning the content of the last message—which happens to be the AI message, as we saw earlier.

A CHATBOT FRONTEND IN STREAMLIT

Our app's backend is now ready, so let's focus on the frontend next. Streamlit really shines when it comes to chatbot interfaces because of its native support for them.

This is evident in the fact that our first iteration of `frontend.py`—shown in listing 10.4—is only 12 lines long.

Listing 10.4 `frontend.py`

```
import streamlit as st
from bot import Bot

if "bot" not in st.session_state:
    api_keys = st.secrets["api_keys"]
    st.session_state.bot = Bot(api_keys)
bot = st.session_state.bot

if human_message_text := st.chat_input("Chat with me!"):
    st.chat_message("human").markdown(human_message_text)
    ai_message_text = bot.chat(human_message_text)
    st.chat_message("ai").markdown(ai_message_text)
```

(`chapter_10/in_progress_01/frontend.py` in the GitHub repo)

We start by putting a reference to our `Bot` instance—`bot`—in `st.session_state`, which is essentially the same pattern we've used in the last two chapters for the `Hub` and `Game` classes. To do so, we pass in the `api_keys` object from `st.secrets` to do so. We'll create `secrets.toml` in a bit.

The interesting part is the last four lines. The first of these introduces a new Streamlit widget called `st.chat_input`:

```
if human_message_text := st.chat_input("Chat with me!"):
    ...
```

`st.chat_input` creates a text input box with a "Send" icon, similar to what you're probably used to in various messaging apps. Besides the "Send" icon, it's different from `st.text_input` in a few noticeable ways:

- It's pinned to the *bottom* of the screen or the containing widget you put it in
- Unlike `st.text_input`, which returns a value once a user clicks out of the textbox, `st.chat_input` only returns a value once the user has clicked "Send" or pressed Enter.

Apart from `st.chat_input`, the code above may look unfamiliar for another reason; we're using the character sequence `:=`, which is called a *walrus operator* in Python (because if you tilt your head to the side, it kind of looks like a walrus).

The walrus operator is just a trick to make your code slightly more concise. It allows you to assign values to variables as part of a larger expression rather than requiring a separate line for assignment. In other words, instead of the line we're discussing, we could have written the following to obtain the same effect:

```
human_message_text = st.chat_input("Chat with Nibby!")
if human_message_text:
    ...
    ...
```

NOTE Python developers are divided on whether the walrus operator increases or decreases the readability of your code. Regardless of whether you choose to use it, it's a good idea to know what it is.

Once we have an input message from the user, we can display the conversation:

```
st.chat_message("human").markdown(human_message_text)
ai_message_text = bot.chat(human_message_text)
st.chat_message("ai").markdown(ai_message_text)
```

`st.chat_message` is a Streamlit display widget that accepts either of two strings—"human" or "ai"—and styles the container accordingly. This includes showing an avatar corresponding to a user or a robot.

In this case, we display `human_message_text` using `st.chat_message("human")`, call the `chat` method of our Bot instance, and display the response AI text with `st.chat_message("ai")`.

Just to be clear, `st.chat_message` is similar to other Streamlit elements like `st.column` in that we could also have written:

```
with st.chat_message("human"):
    st.markdown(human_message_text)
```

To complete the first version of Nibby, we need to create a `secrets.toml` file for our OpenAI API key, in a new `.streamlit` folder. The contents of this file are in listing 10.5.

Listing 10.5 .streamlit/secrets.toml

```
[api_keys]
OPENAI_API_KEY = 'sk-proj-...'
#A

#A Replace sk-proj... with your actual OpenAI API key.
```

Go ahead and run your app with `streamlit run frontend.py` to test it out. Figure 10.7 shows our chatbot in action.

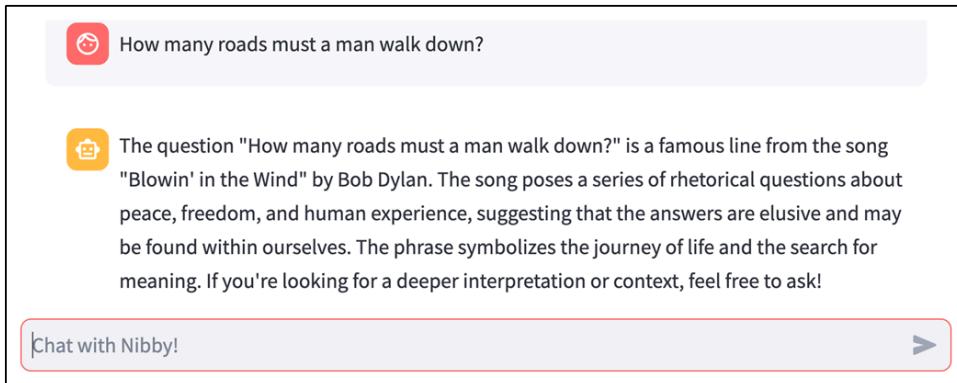


Figure 10.7 A single-prompt-single-response chatbot in Streamlit (see `chapter_10/in_progress_01` in the GitHub repo for the full code).

Sweet! Notice the human and bot avatars, as well as the subtle background shading to distinguish between the two kinds of displayed messages.

If you play around with the app, you'll realize that Nibby can't hold a conversation at this point, only respond to single messages. Next up, let's fix that!

10.3 Multi-turn conversations

It took some effort to get there, but we've built an initial version of Nibby. Unfortunately, at the moment, Nibby's idea of a conversation is a single response to a single message.

For instance, consider the exchange in figure 10.8

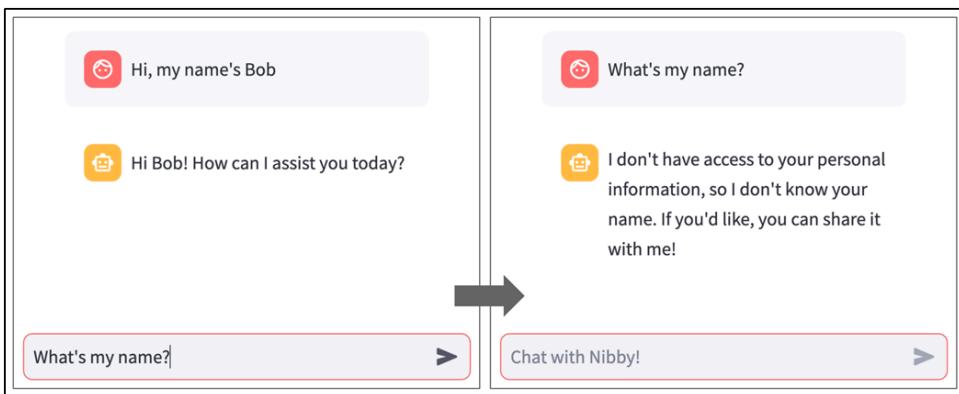


Figure 10.8 Our chatbot doesn't remember the information we gave it.

There are two things wrong here:

- The bot didn't remember the information I gave it in the prior message.
- Our frontend treats the second message-response pair as a completely new conversation, removing all traces of the first.

In this section, we'll iterate on Nibby, solving both issues.

10.3.1 Adding memory to our graph

Recall the simple one-node graph we created: it starts with a state containing a human message, passes this message to an LLM, appends the AI's response to the state, and then returns the updated state.

What happens if you invoke the graph again with a follow-up message? Well, the process repeats—a *new* state containing *only* the follow-up message is created and passed to the graph, which treats this as a brand-new independent execution.

This is a clear problem since conversations very rarely consist exclusively of a single message and response. The user *will* want to follow up, and the chatbot needs to remember what came before.

To enable our graph to remember prior executions, we need to *persist* the state, rather than starting from scratch every time. Luckily, LangGraph makes this a snap through the concept of *checkpointers*, which can save the state of the graph at each step.

Specifically, we're going to use a checkpoint to allow our graph state to be stored in memory. We can then assign a *thread ID* to each invocation of the graph. Whenever we pass the same thread ID while invoking the graph, the graph will recall the state previously stored in memory for that thread ID and start from *there* rather than from a clean slate.

To implement this, make the changes shown below to `graph.py`:

```

from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import START, END, StateGraph, MessagesState
...
class SupportAgentGraph:
    def __init__(self, llm):
        self.llm = llm

        self.config = {"configurable": {"thread_id": "1"}}
    ...
    ...
    def build_graph(self):
        memory = MemorySaver()
        builder = StateGraph(MessagesState)
        ...
        return builder.compile(checkpointer=memory)

    def invoke(self, human_message_text):
        ...
        return self.graph.invoke(state, self.config)

```

(chapter_10/in_progress_02/graph.py in the GitHub repo)

Let's start our discussion of the code above with `build_graph`. We've added a line at the top of this method:

```
memory = MemorySaver()
```

`MemorySaver` is a checkpointer built into LangGraph that can store graph states in memory. Various other kinds of checkpointers are available, depending on where you want to save your graph state. For instance, you could use different checkpointers to store conversations in a database like PostgreSQL or SQLite.

We pass this to our graph when we compile it at the end of the method:

```
return builder.compile(checkpointer=memory)
```

This allows our graph to save its state, but that's not enough. If we don't make any more changes, each graph invocation would still be a new, independent one. We need a way to tell the graph that a particular invocation belongs to a *thread* it has seen before.

Direct your attention to `__init__`, where we've assigned a strange-looking value to a field called `self.config`:

```
self.config = {"configurable": {"thread_id": "1"}}
```

The important part to notice here is `{"thread_id": "1"}`. Further down, in the `invoke` method, we pass this to the graph while invoking it:

```
return self.graph.invoke(state, self.config)
```

We're essentially passing the thread ID 1 to the graph here so it knows that every time we invoke it, we're always in the same conversation thread which has an ID of 1.

As a result of this change, the first invocation of the graph ("Hi, my name's Bob" in the example that prompted these changes) will be saved under the thread ID 1. At this point, the state will have two messages: the original human message and the AI response.

When the follow-up message ("What's my name") arrives, since we already have an existing thread with ID 1, it will be *appended* to the existing state. The state that's passed to `assistant_node` (and therefore to the LLM) will have *three* messages, enabling it to respond correctly.

ADDITIONAL QUESTIONS YOU MAY HAVE

Two natural questions may arise at this juncture:

- Why is the thread ID always 1?

Recall that our Streamlit app session doesn't persist beyond a single browser refresh. So each time the user accesses the app by opening it in a new tab or refreshing the browser, the `SupportAgentGraph` instance is rebuilt, and the graph is re-compiled with a new `MemorySaver` object. Since `MemorySaver` stores graph states in memory instead of persisting it to an external data store like PostgreSQL, any thread from a different browser session—whatever the thread ID—is inaccessible, so we can safely use the same thread ID 1 for the new session.

Long story short, how we've set things up guarantees that a single graph instance will see at most one conversation in its lifetime, so we only need to specify one thread ID.

- Why is the value of `self.config` so convoluted?

Looking at our explanation of check pointers and memory, it seems that all we need to pass the graph when we invoke it is the value 1. So why do we have this monstrosity: `{"configurable": {"thread_id": "1"}}`?

Though they are beyond the scope of this book, LangGraph offers many options when you invoke a graph, such as the ability to specify metadata or the number of parallel calls it can make. The thread ID is the only configuration we're using here, but it's far from the only one available. The convoluted-seeming structure of `self.config` reflects this.

Try running Nibby again and entering the same messages as before. This time you should see something similar to figure 10.9.

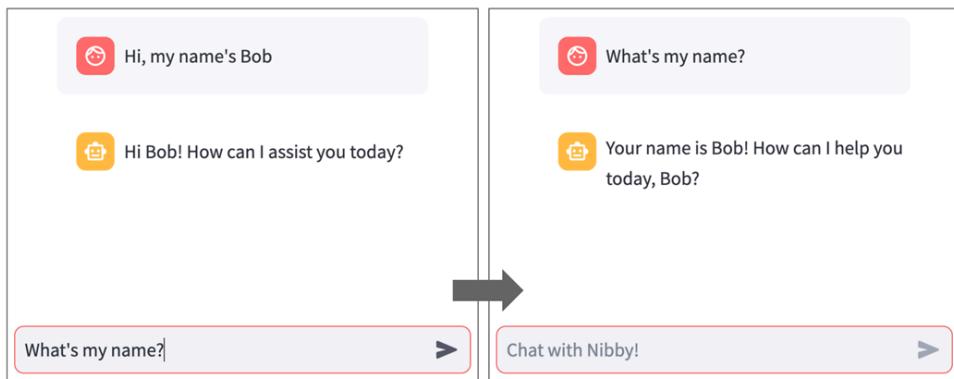


Figure 10.9 Nibby now remembers information we told it earlier in the conversation (see [chapter_10/in_progress_02](#) in the GitHub repo for the full code).

As you can see, the app does remember the previous information we gave it this time, but we still need to update the frontend to show the entire conversation.

10.3.2 Displaying the conversation history

Our Streamlit frontend is currently only set up to show the latest user-entered message and the AI's response.

To display the full history, we need to first expose it in the backend. Let's start by adding a method to `graph.py` to get the entire conversation so far at any point:

```

...
class SupportAgentGraph:

    ...
    def get_conversation(self):
        state = self.graph.get_state(self.config)
        if "messages" not in state.values:
            return []
        return state.values["messages"]

```

(chapter_10/in_progress_03/graph.py in the GitHub repo)

The `get_conversation` method in `SupportAgentGraph` simply returns the `messages` list in the graph's current state.

To do this, it first gets a reference to the state (`self.graph.get_state(self.config)`), and then accesses the `"messages"` key using `state.values["messages"]`. Passing `self.config` to `get_state` is required to get us the correct conversation thread, though—as the sidebar in the previous section discusses—there's only one.

Next, let's expose the full `messages` list in `bot.py`:

```

...
class Bot:

    ...
    def get_history(self):
        return self.graph.get_conversation()

```

(chapter_10/in_progress_03/bot.py in the GitHub repo)

All the `get_history` method does is to pass the result of the `get_conversation` method we just defined faithfully through to its caller.

```

We can now make the changes required in frontend.py:
...
bot = st.session_state.bot

for message in bot.get_history():
    st.chat_message(message.type).markdown(message.content)

if human_message_text := st.chat_input("Chat with Nibby!"):
    ...

```

(chapter_10/in_progress_03/frontend.py in the GitHub repo)

We call `bot.get_history()` to get the list of messages and iterate through it, displaying each in its own `st.chat_message` container.

Recall that each message in the `messages` list is an instance of either `HumanMessage` or `AIMessage`. Either way, it also has a `type` field with a value of "human" in the case of `HumanMessage` and "ai" for an `AIMessage`. This therefore works perfectly as the type indicator argument in `st.chat_message`.

`message.content` has the message's text, so we display that using `st.markdown`.

Rerun the app to see figure 10.10.

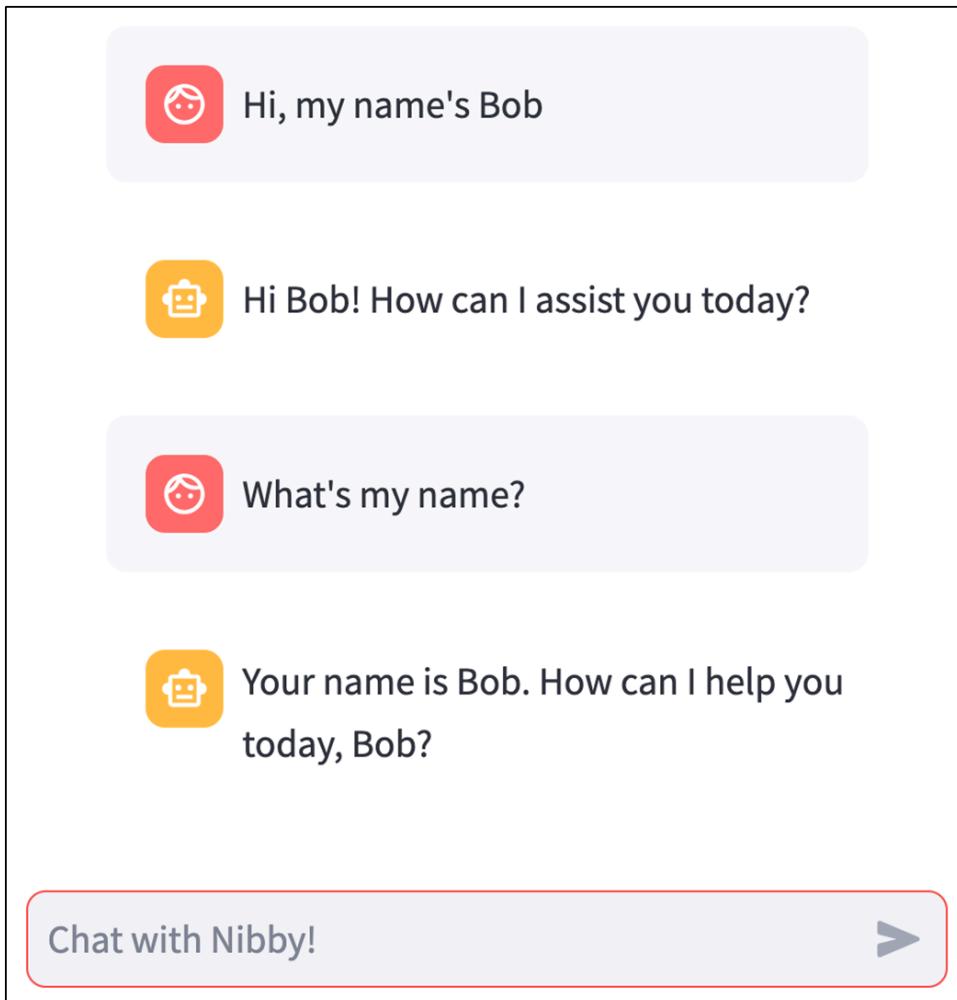


Figure 10.10 Our frontend now displays the full conversation history (see `chapter_10/in_progress_03` in the GitHub repo for the full code)

As expected, Nibby now shows the full conversation so we can keep track of what's going on.

10.4 Restricting our bot to customer support

Thus far, we've focused on getting Nibby's basic functionality right—including calling an LLM and handling a full conversation. The result is a *general* chatbot you can ask for pretty much anything.

For example, consider what happens if we ask Nibby to sing a song (figure 10.11):

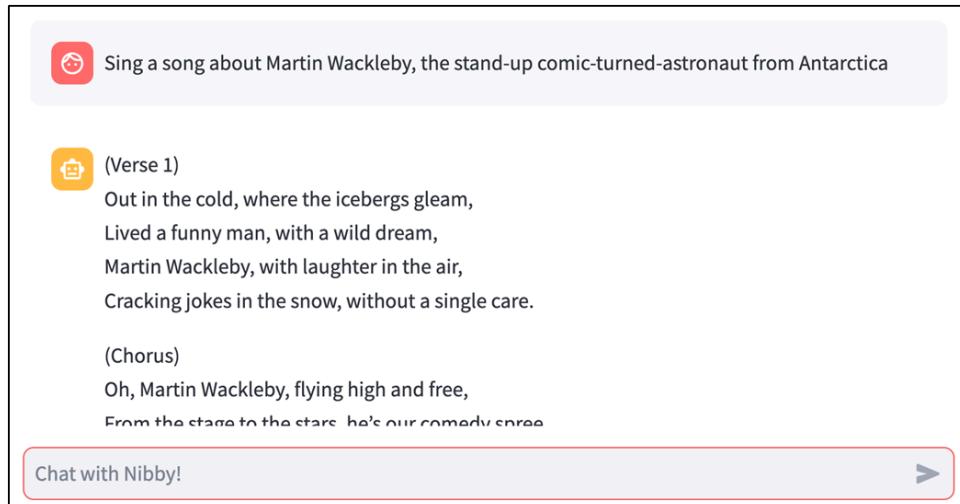


Figure 10.11 Nibby entertains frivolous requests, potentially costing us money.

Nibby can sing a song all right. It can also help you solve math problems or write an essay about the fall of the Roman Empire. Unfortunately, it does all of that on the company's dime. Remember, interacting with a cloud-based LLM costs *money*.

Each time someone makes a frivolous request to your customer support bot, and the bot indulges the request with a long-winded response, it spends precious LLM tokens and costs you something. Sure, each message is only a fraction of a cent but add up all the pleas for coding assistance or role-playing as a character from Battlestar Galactica, and suddenly your boss wants to know why there's a Nibby-shaped hole in the company's quarterly earnings report.

Of course, I'm being hyperbolic, but the point stands: we want Nibby to be strictly business.

The best thing about an LLM is that you can simply tell it what you want it to do or not do, so this turns out to be an easy fix; we'll simply add an appropriate *prompt*.

10.4.1 Creating a base prompt

As we did in chapter nine, we want to give the LLM some context about the use case we want it to serve. In that chapter, we did this by creating a message with the role "system." That's essentially what we're going to do here, too, though the abstractions we'll use are slightly different.

Create a file called `prompts.py` with the content shown in listing 10.6.

Listing 10.6 prompts.py

```
BASE_SYS_MSG = """
You are a customer support agent for Note n' Nib, an online stationery
retailer. You are tasked with providing customer support to customers who
have questions or concerns about the products or services offered by the
company.

You must refuse to answer any questions or entertain any requests that
are not related to Note n' Nib or its products and services.

"""
```

([chapter_10/in_progress_04/prompts.py in the GitHub repo](#))

The prompt gives Nibby its first indication that Note n' Nib exists and that it's supposed to be providing customer support for the company.

Importantly, `BASE_SYS_MSG` also has an instruction to refuse any requests that are unrelated to Note n' Nib. Next, let's incorporate this into our graph.

10.4.2 Inserting a base context node in our graph

As we learned in chapter nine, using OpenAI's chat completions endpoint involves passing a sequence of messages to the LLM.

In our current graph, the list starts with the user's first instruction and contains only user messages and AI responses. To prevent Nibby from responding to frivolous requests, we just need to insert the system prompt we just created as the first message in the list we send to the LLM.

We'll do this by inserting a new node in the graph to add the system message to the graph state and modifying the existing `assistant_node` to pass this message to the LLM before anything else.

Figure 10.12 shows a visual representation of the new graph.

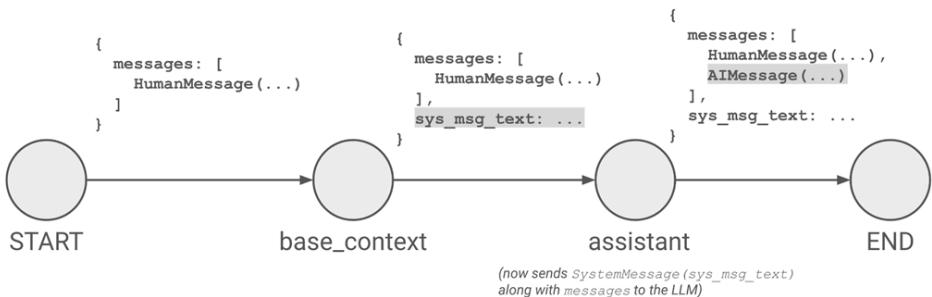


Figure 10.12 Adding a base context node in our graph

The changes required to `graph.py` are shown in listing 10.7:

Listing 10.7 graph.py (modified)

```

...
from langchain_core.messages import HumanMessage, SystemMessage
from prompts import *

class AgentState(MessagesState):
    sys_msg_text: str

class SupportAgentGraph:
    def __init__(self, llm):
        ...

    @staticmethod
    def base_context_node(state):
        return {"sys_msg_text": BASE_SYS_MSG}

    def get_assistant_node(self):
        def assistant_node(state):
            sys_msg = SystemMessage(content=state["sys_msg_text"])
            messages_to_send = [sys_msg] + state["messages"]
            ai_response_message = self.llm.invoke(messages_to_send)
            return {"messages": [ai_response_message]}
        return assistant_node

    def build_graph(self):
        memory = MemorySaver()
        builder = StateGraph(AgentState)

        builder.add_node("base_context", self.base_context_node)
        builder.add_node("assistant", self.get_assistant_node())

```

```

builder.add_edge(START, "base_context")
builder.add_edge("base_context", "assistant")
builder.add_edge("assistant", END)

return builder.compile(checkpointer=memory)

def invoke(self, human_message_text):
    ...
    ...

```

(chapter_10/in_progress_04/graph.py in the GitHub repo)

Starting from the top, we've added a couple of imports; we need the `SystemMessage` class in addition to `HumanMessage`, so that's one.

The statement `from prompts import *` allows us to access any prompt we may add to `prompts.py` using only its variable name—without a prefix like `prompt..`. Since we're using the `*` wildcard here rather than importing specific objects, every object in `prompt.py`'s global scope becomes part of `graph.py`'s scope. In this case, it means we can refer to `BASE_SYS_MSG` directly, as we do later in the code.

We've defined a new `AgentState` class:

```

class AgentState(MessagesState):
    sys_msg_text: str

```

`AgentState` inherits from `MessagesState`, so it also contains the `messages` field we've been using thus far. What we're effectively doing here is adding a new field to the state—called `sys_msg_text`—meant to hold the text of the system message.

Next, within the class itself, we've added a new static method:

```

@staticmethod
def base_context_node(state):
    return {"sys_msg_text": BASE_SYS_MSG}

```

This function represents the new node we're adding to the graph, called `base_context`. All this node does is to populate the `sys_msg_text` field we've added to the state. By returning `{"sys_msg_text": BASE_SYS_MSG}`, this node sets `sys_msg_text` to `BASE_SYS_MSG`—the context prompt we created a few minutes ago—in the graph's current state.

To understand how this works, it's helpful to remember that a graph node does not return the entirety of the state; rather it only returns the keys in the state that need to be modified. Therefore, even though there's no mention of the `messages` field here, once this node has been executed, the state will continue to have that field—unmodified—in addition to `sys_msg_text`.

NOTE Unlike in the case of `messages`, when we return a dictionary with a `sys_msg_text` key it replaces the value of `sys_msg_text` in the state. This is because `sys_msg_text` uses the default update behavior, as opposed to the `append` behavior (enabled internally by the `add_messages` function) that `messages` uses.

Why have we made `base_context_node` a static method? Well, recall once again that each node in the graph needs to accept the graph state as its first argument. We would like to put `base_context_node` inside `SupportAgentGraph` for logical code organization purposes, but if we make it a regular method, it'll need to accept the class instance (`self`) as its first argument. Making it a static method removes that requirement, and frees us to add a state argument.

Some of you might be asking, "Wait a minute, didn't we structure `assistant_node` as a nested function for the same reason? Why didn't we do *that* here?"

We could indeed have used a closure-based solution for `base_context_node` too, but we don't need to; unlike `assistant_node` which references `self.llm`, `base_context_node` doesn't need to access `self` at all. We therefore employ the more straightforward technique of applying the `@staticmethod` decorator to `base_context_node`.

Speaking of `assistant_node`, consider the changes we've made to its code above:

```
sys_msg = SystemMessage(content=state["sys_msg_text"])
messages_to_send = [sys_msg] + state["messages"]
ai_response_message = self.llm.invoke(messages_to_send)
```

Rather than invoking the LLM directly with `state["messages"]`, we now create a `SystemMessage` object with the `sys_msg_text` field we populated in `base_context_node` as the content and prepend it to `state["messages"]` to form the list we pass the LLM.

Finally, note our updates to `build_graph`. Since we've extended `MessagesState` to include a `sys_msg_text` field, we use that to initialize the `StateGraph`:

```
builder = StateGraph(AgentState)
```

We add the base context node like this:

```
builder.add_node("base_context", self.base_context_node)
```

Notice how we're passing a reference to the `self.base_context_node` method itself here, as opposed to calling it with the double parentheses.

We also reorder the edges in the graph to insert the `base_context` node between `START` and `assistant`:

```
builder.add_edge(START, "base_context")
builder.add_edge("base_context", "assistant")
```

That should be all we need. Go ahead and re-run your app. Try requesting the bot to sing a song again to get a response similar to figure 10.13.

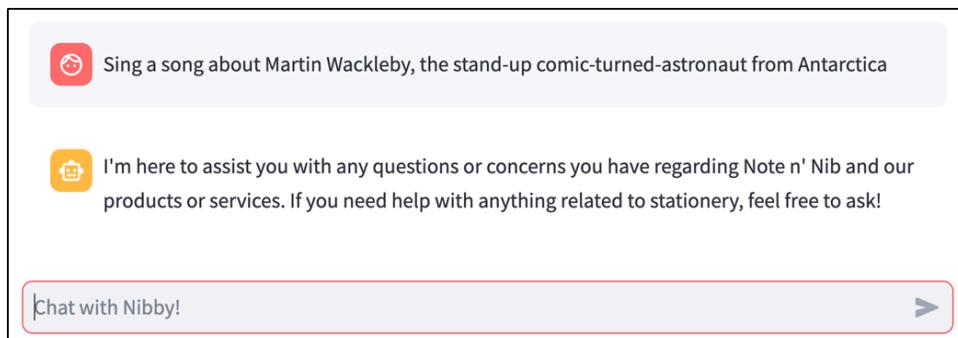


Figure 10.13 Nibby now refuses to entertain frivolous requests (see [chapter_10/in_progress_04](#) in the GitHub repo for the full code).

It looks like Nibby got the memo! It won't help the user with irrelevant requests anymore. In the next section, we'll solve the opposite problem: getting it to help with *relevant* questions.

10.5 Retrieval Augmented Generation

Models like GPT-4o are so effective because they have been pre-trained on a huge corpus of publicly available information, such as books, magazines, and websites. It's why Fact Frenzy, our trivia app from chapter nine, was able to ask and answer questions on such a wide range of topics.

However, many of the more economically valuable use cases of generative AI require more than information in the public domain. Truly molding AI into something that fits your specific use case usually requires providing it with private information that only you possess.

Take Nibby, for instance, who is ultimately meant to assist customers of Note n' Nib with their queries. What happens if we pose a valid question about a stationery product to Nibby? Figure 10.14 shows such a conversation.

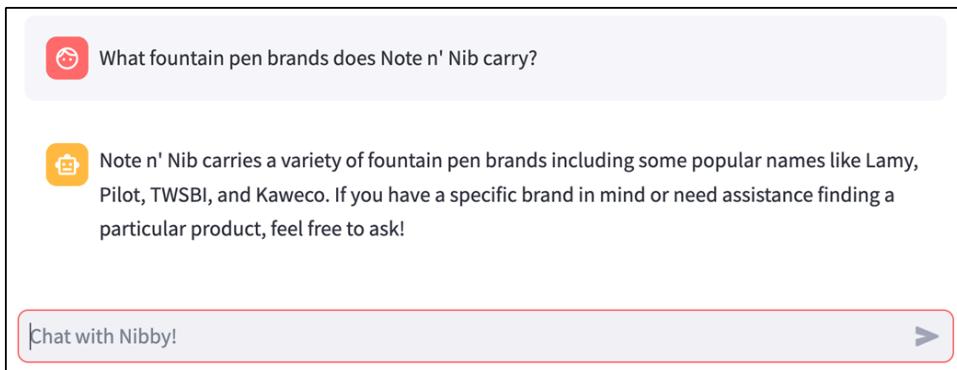


Figure 10.14 Nibby makes up information when it doesn't know the answer.

Seems like Nibby knocked that one out of the park, right? Not quite. We never told our bot what kinds of pens Note n' Nib carries, so where is it getting its information from? Additionally, where are the fictional brands—InkStream and RoyalQuill—we encountered in chapter six? As it turns out, Nibby had no information available about fountain pens and, therefore, simply hallucinated this response!

In this section, we'll discover a way to augment Nibby's existing store of worldly knowledge with custom information that we provide.

10.5.1 What is Retrieval Augmented Generation?

So, how do we supplement all of the information an LLM has been trained on with our own? For relatively small pieces of information, it's actually trivially easy—in fact, we already know how! All we need to do is provide the info to the LLM as part of our prompt!

We could get Nibby to get the question we posed in figure 10.14 right simply by listing the products Note n' Nib sells directly in the system message we send to the LLM.

What about other questions, though? Technically, we could give the model all of the contextual information it might realistically need to answer any question directly in the prompt. The maximum amount of such information we can provide is measured in tokens, called the model's *context window length*.

Relatively recent models have a huge context window. For instance, gpt-4o-mini can take up to 128,000 tokens (about 96,000 words, since—on average—a token is roughly three-quarters of a word), while o3-mini—a newer reasoning model from OpenAI—has a context window of 200,000 tokens. Models from other providers can take even more tokens in a single prompt. Google's Gemini 2.0 Pro has a context window that is a whopping 2 million tokens long—enough to fit the entire Harry Potter series of books, with space left over for almost all of the Lord of the Rings trilogy.

Surely our problem is solved then? We can simply assemble all the information we possess about Note n' Nib and feed it to the LLM in each prompt, correct?

Certainly, we could, but we probably don't want to for a couple of reasons:

- LLMs are prone to information overload; we generally see degraded performance with extremely large prompts.
- Even if there were no such degradation, LLM providers usually charge by the token, so if we had to pass our entire custom knowledge base in every LLM call, the costs would go through the roof.

No, we need a different solution. If only we could **read in a user's question and feed the LLM just the relevant parts of our knowledge base required to answer it**.

And that—in case you've somehow failed to realize where this spiel is going—is exactly what Retrieval Augmented Generation (RAG) is.

RAG has the following essential steps:

- Read the user's question
- **Retrieve** the context *relevant* to the question from the knowledge base
- **Augment** the question with the context required to answer it
- **Generate** the answer to the question by feeding the question and context to the LLM

The hard part of RAG is the *retrieve* step. Specifically, given a user question and a large custom knowledge base, how do you identify the parts of the knowledge base that are relevant to the question and extract only those parts from the base?

The answer lies in the concept of *embeddings* and a piece of software known as a *vector database*.

EMBEDDINGS AND VECTOR DATABASES

While we don't—strictly speaking—need to learn how embeddings—or even vector databases—work under the hood to implement RAG, it would be a good idea to gain a basic understanding of these concepts.

Let's start with a simplified example to achieve this. Say you're known as something of a movie buff in your friend circle. Your buddy approaches you and says, "Hey, I watched *The Dark Knight* yesterday and loved it! Could you recommend another movie like it?"

You're in a fix because—though you have an encyclopedic knowledge of movies—you're not sure how exactly to measure the *similarity* between two movies, so you can recommend the one that's the *most* similar to *The Dark Knight*. Refusing to accept defeat, you flee to your underground lair and try to work it out in solitude.

Eventually, you come up with a system. You reckon that when people express their preference for various movies, they're subconsciously talking about two attributes: *comedic value* and *explosions per hour*. Therefore, you rate your entire catalog of movies against those two scales and plot the results in a chart (partially reproduced in figure 10.15).

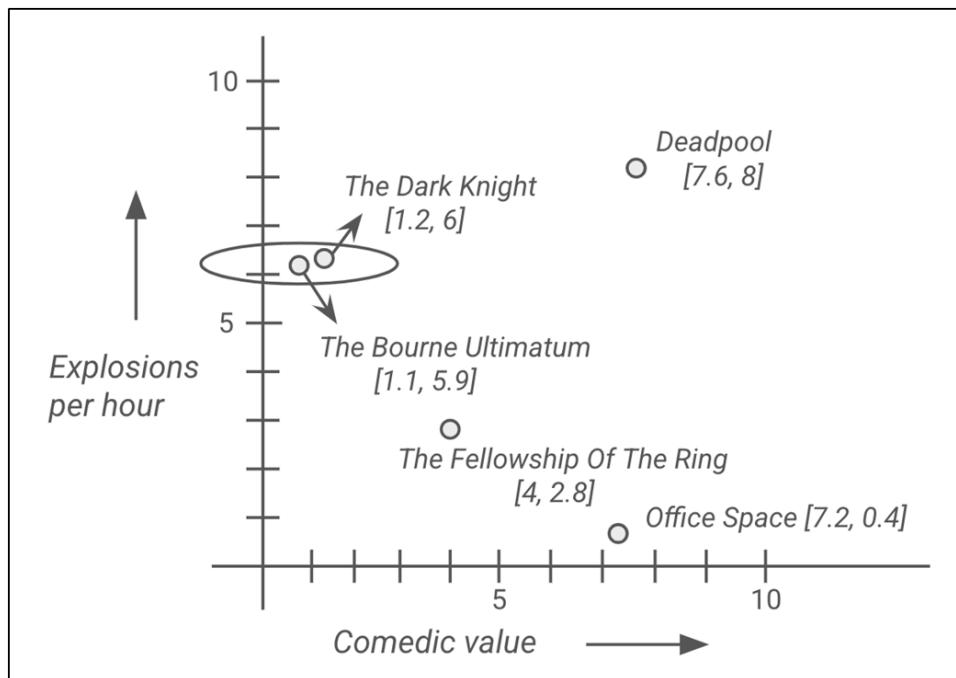


Figure 10.15 Converting movies into vectors and plotting them on a chart

As you can see, *The Dark Knight* has a comedic value of 1.2 but a relatively high explosions-per-hour of 6. We can represent it as a list of numbers: [1.2, 6], called a *vector*. We can call the vector [1.2, 6] the *embedding* of the movie *The Dark Knight* in the two-dimensional comedic-value/explosions-per-hour space.

Converting movies into numbers in this way makes it possible to measure their similarity. For instance, *Office Space* is represented as [7.2, 0.4] in the same space. The similarity (or, rather, lack thereof) between *Office Space* and *The Dark Knight* can be calculated mathematically by considering their *geometric distance*. The closer the embeddings of two movies are geometrically—as measured by the length of a straight line drawn between them—the more similar the underlying movies are.

After several such calculations, you find that *The Bourne Ultimatum*, which has the vector [1.1, 5.9] is the closest to *The Dark Knight*. Having concluded your research, you get back to your friend and let them know (to which your friend responds, "Thank goodness you're alive! It's been two years, where have you been?").

The question we're facing with Nibby is analogous to the movie recommendation problem above. Given a user's message (the movie your friend liked), and a knowledge base (your catalog of movies), we have to find the paragraphs/chunks (movies) that are most relevant (most "similar") to the user's message.

To answer this efficiently, we need two things:

- A way to convert a given piece of text into embeddings that capture its meaning (or *semantics*)

- A way to store these embeddings and quickly calculate distances between them

Obviously, the movie example above is overly simplistic. Our "space" only had two dimensions: comedic value and explosions per hour. Encoding the meaning of a piece of text requires a lot more dimensions (like hundreds or thousands), and the dimensions themselves would not be human-understandable concepts like "comedic value." We'll use a text embedding model provided by OpenAI for our use case.

To store the embeddings, we'll use a program called a *vector database*. Vector databases make it easy to calculate the distance between embeddings or find the entries closest to a specific one. Rather than the "straight line" (or *Euclidean*) distance between vectors, we'll use a score called *cosine similarity*, which measures the *angle* between two vectors to determine their similarity. The vector database we'll use is Pinecone, a cloud-hosted service.

10.5.2 Implementing RAG in our app

Armed with a conceptual understanding of RAG, let's now focus on implementing it to assist Nibby in answering customer questions.

PREPARING THE KNOWLEDGE BASE

The GitHub folder for this chapter (https://github.com/aneevdavis/streamlit-in-action/tree/main/chapter_10) has a subdirectory called `articles` with a series of customer support articles for Note n' Nib. Each article is a text file containing information about Note n' Nib's products or how it conducts business. For instance, `our_products.txt` includes a list of product descriptions, while `fountain_pen_maintenance.txt` is about maintaining RoyalQuill and InkStream pens.

Here's an excerpt from the article that we'll reference later:

Proper care ensures your InkStream and RoyalQuill fountain pens write smoothly for years.

- Cleaning: Flush the nib with warm water every few weeks.
- Refilling: Use high-quality ink to prevent clogging.
- Storage: Store pens upright to avoid leaks and ensure ink flow.

Copy the `articles` folder into your working directory now. This is the knowledge base that Nibby will have access to.

SETTING UP A VECTOR DATABASE

As mentioned briefly in the previous section, we will be using Pinecone, a managed vector database optimized for fast and scalable similarity search that's popular for AI applications. Pinecone's free "Starter" plan is more than enough for this chapter.

Go to <https://www.pinecone.io/> and sign up for an account now. As soon as you're done with the setup, you'll be presented with an API key, which you should save immediately. Once that's done, create a new *index*. An index is analogous to a table in a regular non-vector database like PostgreSQL. The index will store our support articles broken into parts, along with their embeddings.

You'll be asked to supply values for various options during the creation process:

- **Index name:** You can use whatever you like, but remember to save it since we'll use it in our code.
- **Model configuration:** Choose `text-embedding-ada-002`, the OpenAI embedding model we'll use; the value of dimensions should automatically be set to 1,536.
- **Metric:** Pick `cosine` to use the cosine similarity score we briefly discussed earlier.
- **Capacity mode** should be `Serverless`
- **Cloud provider:** `AWS` is fine for this.
- **Region:** At the time of writing, only `us-east-1` is available in the free plan, so pick that.

INGESTING THE KNOWLEDGE BASE

To incorporate the vector store into our chatbot app, we'll create a `VectorStore` class. Before doing so, add your Pinecone API key and the index name you just created to `secrets.toml` so it now looks like this:

```
[api_keys]
OPENAI_API_KEY = 'sk-proj-...'      #A
VECTOR_STORE_API_KEY = 'pcsk...'      #B

[config]
VECTOR_STORE_INDEX_NAME = 'index_name_you_chose'      #C
```

#A Replace with your actual OpenAI API key.

#B Replace with your Pinecone API key.

#C Replace `sk-proj...` with the actual index name.

We've added a new key to `api_keys` and a new section called `config` to hold the index name.

Now on to the `VectorStore` class. Create a file named `vector_store.py` with the contents shown in listing 10.8.

Listing 10.8 vector_store.py

```
from pinecone import Pinecone
from langchain_openai import OpenAIEMBEDDINGS
from langchain_pinecone import PineconeVectorStore
from langchain_community.document_loaders import DirectoryLoader, TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

class VectorStore:
    def __init__(self, api_keys, index_name):
        pc = Pinecone(api_key=api_keys["VECTOR_STORE_API_KEY"])
        embeddings = OpenAIEMBEDDINGS(api_key=api_keys["OPENAI_API_KEY"])
        index = pc.Index(index_name)
        self.store = PineconeVectorStore(index=index, embedding=embeddings)

    def ingest_folder(self, folder_path):
        loader = DirectoryLoader(
            folder_path,
            glob="**/*.txt",
            loader_cls=TextLoader
        )
        documents = loader.load()
        splitter = RecursiveCharacterTextSplitter(
            chunk_size=1000,
            chunk_overlap=200
        )
        texts = splitter.split_documents(documents)
        self.store.add_documents(texts)

    def retrieve(self, query):
        return self.store.similarity_search(query)
```

(`chapter_10/in_progress_05/vector_store.py` in the GitHub repo)

There's a fair bit going on here, so let's go through it step-by-step.

The `__init__` has the boilerplate code required to set up the vector store connection. It accepts two arguments: `api_keys` (the dictionary of API keys) and `index_name` (the name of the Pinecone index).

`__init__` first creates the `Pinecone` object—`pc`—using the Pinecone API key we noted a minute ago, and then an `OpenAIEmbeddings` object by passing it the OpenAI key. `pc.Index(index_name)` refers to the index we created earlier. Finally, we obtain a vector store object from the index and embeddings and assign it to `self.store` so we can use it in other methods.

The `ingest_folder` method accepts the path to a folder and saves its contents to the Pinecone index. Consider the first part of this method:

```
loader = DirectoryLoader(
    folder_path,
    glob="**/*.txt",
    loader_cls=TextLoader
)
```

LangChain provides various *document loaders* to help ingest and parse various types of data (text files, PDFs, web pages, databases, etc.) into a structured format suitable for processing. `DirectoryLoader` makes it easy to load files from a specified directory.

The `glob="**/*.txt"` ensures that all text files (`.txt`) in the folder (including subfolders) are included.

`loader_cls=TextLoader` tells `DirectoryLoader` to use another loader class called `TextLoader`—also provided by LangChain—for loading individual text files.

Once the `DirectoryLoader` is created, the next step is to load the documents:

```
documents = loader.load()
```

This reads all `.txt` files in the directory and loads them into a list of `Document` objects, which LangChain uses to store the raw text and metadata.

At the moment, each `Document` consists of an entire article. While the articles in our folder are relatively small, one can easily imagine a support article being thousands of words long. The point of fetching just the relevant text from our knowledge base is to reduce the overall size of the prompt; simply fetching entire articles defeats this purpose.

Therefore, we want to divvy up the articles into manageable *chunks* of text that are roughly of equal size. That's what the next part of our code does:

```
splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=200
)
texts = splitter.split_documents(documents)
```

`RecursiveCharacterTextSplitter` is a text-splitting utility from LangChain that can break documents into chunks while preserving meaningful context.

`chunk_size=500` sets the length of each chunk to 500 characters.

`chunk_overlap=200` means that chunks will have a 200-character overlap to maintain context.

After splitting the documents, the chunks—available in `texts`—are ready to be stored:

```
self.store.add_documents(texts)
```

`add_documents` takes the split text chunks and adds them to the Pinecone index, storing both the text and embeddings generated using the `text-embedding-ada-002` model, making them searchable.

The `retrieve` method allows callers to query the vector store:

```
def retrieve(self, query):
    return self.store.similarity_search(query)
```

The `similarity_search` method of our `PineconeVectorStore` instance (`self.store`) searches the vector store for documents similar to `query`—the user's message—using the generated embeddings, returning a list of relevant `Document` objects based on the query.

At this point, we've coded up the vector store functionality into a handy little class; next, let's use the class to ingest our `articles/` folder.

This is an *offline* step that you only need to perform one time; once you've stored your articles in Pinecone, they'll remain there until you remove them.

Go ahead and create a file called `ingest_to_vector_store.py`, copying the contents from listing 10.9.

Listing 10.9 ingest_to_vector_store.py

```
import toml
from vector_store import VectorStore

secrets = toml.load(".streamlit/secrets.toml")
api_keys = secrets["api_keys"]
index_name = secrets["config"]["VECTOR_STORE_INDEX_NAME"]
vector_store = VectorStore(api_keys, index_name)
vector_store.ingest_folder("articles/")
```

(`chapter_10/in_progress_05/ingest_to_vector_store.py` in the GitHub repo)

Since this isn't meant to be run using Streamlit, we use the `toml` module directly to read our `secrets.toml`:

```
secrets = toml.load(".streamlit/secrets.toml")
```

At the end of this, `secrets` should be a dictionary that contains the `api_keys` and `config` keys we organized `secrets.toml` into earlier.

Next we grab the values we need:

```
api_keys = secrets["api_keys"]
index_name = secrets["config"]["VECTOR_STORE_INDEX_NAME"]
```

We can now instantiate our `VectorStore` class:

```
vector_store = VectorStore(api_keys, index_name)
```

Finally, we trigger the `ingest_folder` method:

```
vector_store.ingest_folder("articles/")
```

To perform the actual ingestion, run this file in your terminal with the `python` command:

```
python ingest_to_vector_store.py
```

Once it completes, you can go to the page corresponding to your index on the Pinecone website to see the newly ingested chunks, as seen in figure 10.16.

Chunks

1 ID a199ffbo-3543-4433-a613-69e5786a6811
SCORE 1.0000 FIELDS source: "articles/pen_ink_guide.txt"
text: "Pen & Ink Guide\nOur InkStream and RoyalQuill fountain pens, as well as our GlidePro ballpoint pe..."

2 ID 29016f52-5cd2-46a6-b567-f95fe14706ce
SCORE 0.9232 FIELDS source: "articles/pen_ink_guide.txt"
text: "- Use bottled ink with a converter or pre-filled ink cartridges.\n- Store upright to prevent leaks.\n-..."

Text of a single chunk

Where the chunk came from

Figure 10.16 You can see the chunks in your index on the Pinecone website.

Notice the source field which holds the file name each chunk came from. You can also click the edit button on a record to add more metadata or to see its numeric vector values. If you were to count them, you'd find that there are 1536 of them, corresponding to the number of dimensions in the embedding model.

ADDING RAG TO THE GRAPH

The Pinecone index we need for RAG is ready to go, but we still need to incorporate the functionality in our chatbot. For this, let's first lay out the additional instructions Nibby needs to use the knowledge base. Append the following to `prompts.py`:

```
SYS_MSG_AUGMENTATION = """
You have the following excerpts from Note n' Nib's
customer service manual:
```
{docs_content}
```

If you're unable to answer the customer's question confidently with the
given information, please redirect the user to call a human customer
service representative at 1-800-NOTENIB.
"""
"""

(chapter_10/in_progress_05/prompts.py in the GitHub repo)
```

In a moment, we'll write the logic to replace `{docs_content}` with the document chunks we retrieve from Pinecone. The idea here is to give Nibby the context it needs and to get it to stop fabricating an answer out of thin air if it's not confident.

Next, let's modify `graph.py` as shown in listing 10.10 so that it implements RAG.

Listing 10.10 graph.py (with RAG nodes)

```
...
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_core.documents import Document
from prompts import *

class AgentState(MessagesState):
    sys_msg_text: str
    retrieved_docs: list[Document]

class SupportAgentGraph:
    def __init__(self, llm, vector_store):
        self.llm = llm
        self.vector_store = vector_store

        self.config = {"configurable": {"thread_id": "1"}}
        self.graph = self.build_graph()

    ...
    def get_retrieve_node(self):
        def retrieve_node(state: AgentState):
            messages = state["messages"]
            message_contents = [message.content for message in messages]
            retrieval_query = "\n".join(message_contents)
            docs = self.vector_store.retrieve(retrieval_query)
            return {"retrieved_docs": docs}
        return retrieve_node

    @staticmethod
    def augment_node(state: AgentState):
        docs = state["retrieved_docs"]
        docs_content_list = [doc.page_content for doc in docs]
        content = "\n".join(docs_content_list)
        new_text = SYS_MSG_AUGMENTATION.replace("{docs_content}", content)
        return {"sys_msg_text": BASE_SYS_MSG + "\n\n" + new_text}

    ...
    def build_graph(self):
```

```

...
builder.add_node("base_context", self.base_context_node)
builder.add_node("retrieve", self.get_retrieve_node())
builder.add_node("augment", self.augment_node)
builder.add_node("assistant", self.get_assistant_node())

builder.add_edge(START, "base_context")
builder.add_edge("base_context", "retrieve")
builder.add_edge("retrieve", "augment")
builder.add_edge("augment", "assistant")
builder.add_edge("assistant", END)

return builder.compile(checkpointer=memory)
...

```

(chapter_10/in_progress_05/graph.py in the GitHub repo)

The first change is to `AgentState`, which now looks like this:

```

class AgentState(MessagesState):
    sys_msg_text: str
    retrieved_docs: list[Document]

```

We'll now store the list of chunks retrieved from Pinecone in the graph state in a variable called `retrieved_docs`.

`__init__` now accepts `vector_store`—an instance of our `VectorStore` class—as an argument and saves it to `self.vector_store`.

Note that rather than create the `VectorStore` instance *within* `graph.py`, we've chosen to have it be created elsewhere (`bot.py`, as we'll soon find out) and simply pass it to the `SupportAgentGraph` class. This is because we want `graph.py` to only contain the core logic of the graph. Objects that the graph *depends* on such as `llm` and `vector_store` should be passed to it. This coding pattern is called *dependency injection*, and is helpful while writing automated tests.

Next, we need to introduce the process of retrieval-augmented generation to our graph. Figure 10.17 shows what the graph should look like by the end of this.

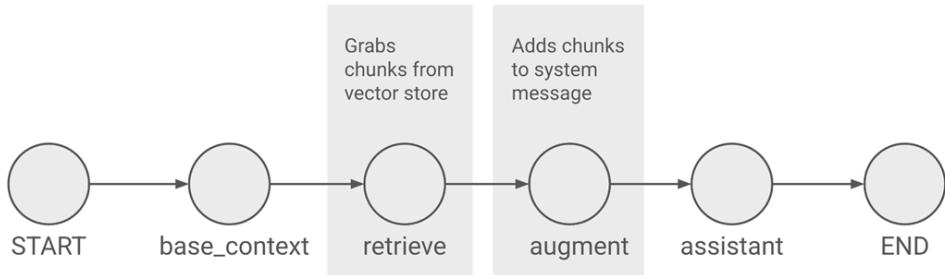


Figure 10.17 Our graph now has retrieve and augment nodes for RAG.

We've inserted two nodes in between the `base_context` and `assistant` nodes: a `retrieve` node that grabs context from the knowledge base that's relevant to the user's query and an `augment` node that adds this info to the prompt.

Here's the code for the `retrieve` node:

```

def get_retrieve_node(self):
    def retrieve_node(state: AgentState):
        messages = state["messages"]
        message_contents = [message.content for message in messages]
        retrieval_query = "\n".join(message_contents)
        docs = self.vector_store.retrieve(retrieval_query)
        return {"retrieved_docs": docs}
    return retrieve_node

```

As in the case of `assistant_node`, `retrieve_node` is structured as a nested function within a class method. It simply extracts the content of all the messages in the conversation and puts them in a single string to form the "query" that we'll pass to the vector store.

The idea here is to find the chunks most relevant to the conversation. Since we're measuring relevance against the query, it makes sense that this is simply the text of the conversation.

Once we have the query, we can call the `retrieve` method we defined earlier and return the list of retrieved documents in a dictionary, thereby updating the `retrieved_docs` key of the graph state.

NOTE While we've kept things simple here by including the text of the entire conversation in the retrieval query, you'll likely run into challenges as the conversation gets longer and longer. For any particular AI response, the most *recent* messages in the conversation are probably more contextually relevant—so it might be a good idea to form the retrieval query from just the last, say, five or six messages in the conversation.

The augmentation node is shown below:

```
@staticmethod
def augment_node(state: AgentState):
    docs = state["retrieved_docs"]
    docs_content_list = [doc.page_content for doc in docs]
    content = "\n".join(docs_content_list)
    new_text = SYS_MSG_AUGMENTATION.replace("{docs_content}", content)
    return {"sys_msg_text": BASE_SYS_MSG + "\n\n" + new_text}
```

This one doesn't need to access anything from `SupportAgentGraph`, so we structure it as a static method as we did for `base_context_node`.

`augment_node` mostly does the tedious work of wrangling and inserting the retrieved chunks into the system message. Once it has formed a string by concatenating the contents of the retrieved `Document` chunks, it simply plugs it into the text of the `SYS_MSG_AUGMENTATION` value we added to `prompts.py`, replacing `{docs_content}`.

At the end of this node, `sys_msg_text` has the full system message—the earlier base message warning Nibby not to entertain frivolous questions, as well as the retrieved context.

We already have a node for the "generate" step in RAG—`assistant_node`—so there's no need to add another.

The edits to `build_graph` should be fairly obvious given figure 10.17; we officially add the `retrieve` and `augment` nodes, and attach them to the right edges.

The next file to edit is `bot.py`. Make the changes below:

```

...
from vector_store import VectorStore

class Bot:
    def __init__(self, api_keys, config):
        self.api_keys = api_keys
        self.config = config

        self.llm = self.get_llm()
        self.vector_store = self.get_vector_store()

        self.graph = SupportAgentGraph(
            llm=self.llm, vector_store=self.vector_store)

    def get_vector_store(self):
        index_name = self.config["VECTOR_STORE_INDEX_NAME"]
        return VectorStore(api_keys=self.api_keys, index_name=index_name)
...

```

(chapter_10/in_progress_05/bot.py in the GitHub repo)

`__init__` now accepts a `config` parameter, saved to `self.config`. It also creates the `vector_store` object by calling the `get_vector_store` method we'll discuss below, and passes it to the `SupportAgentGraph` constructor.

`get_vector_store` has the code required to complete the loop. It obtains the Pinecone index name from `self.config` and passes both `self.api_keys` and `index_name` to create an instance of `VectorStore` before returning it.

The last change we need to make is a pretty small one in `frontend.py`:

```

...
if "bot" not in st.session_state:
    api_keys = st.secrets["api_keys"]
    config = st.secrets["config"]
    st.session_state.bot = Bot(api_keys, config)
bot = st.session_state.bot
...

```

(chapter_10/in_progress_05/frontend.py in the GitHub repo)

Since the `Bot` class now accepts a `config` parameter, we get its value from `st.secrets` and pass it in while instantiating the class.

Let's try asking Nibby what products Note n' Nib sells again. Re-run the app and talk to it. Figure 10.18 shows an example of the new interaction.

The screenshot shows a chat interface with three messages from a bot named Nibby. Nibby uses a red rounded square icon with a white smiley face and a small question mark. The first message is about fountain pen brands, the second about cleaning, and the third about ink sales. Each message includes a small orange icon of a notepad and pen.

What fountain pen brands do you sell?

At Note n' Nib, we offer two brands of fountain pens: InkStream and RoyalQuill. Both are elegant, refillable pens designed for smooth writing. If you have any more questions about these products, feel free to ask!

How do I clean my RoyalQuill?

To clean your RoyalQuill fountain pen, you should flush the nib with warm water every few weeks. This will help maintain optimal ink flow and performance. Regular maintenance, including cleaning, is essential for keeping your pen writing smoothly. If you have any other questions about care or maintenance, let me know!

Why do you not sell ink?

I'm sorry, but I am unable to provide information regarding our product offerings such as ink sales. For specific details about our products, I recommend reaching out to a human customer service representative at 1-800-NOTENIB. Thank you for your understanding!

Chat with Nibby! >

Figure 10.18 Nibby can now access and use information from our knowledge base (see chapter_10/in_progress_05 in the GitHub repo for the full code).

Notice how Nibby answers our questions using information from our knowledge base and redirects to a 1-800 number when it encounters a question it doesn't know how to answer.

10.6 Turning our bot into an agent

Giving Nibby access to Note n' Nib's customer support knowledge base has turned Nibby into a capable assistant, but it's still a purely informational bot. When customers call a support number or chat with a service representative, they're more often than not looking for help with their specific situation or order.

For instance, a customer might wonder why it's taking so long for their placed order to arrive and want to check on the status, or they might want to simply cancel it. To resolve such issues, Nibby can't just rely on static textual articles; it needs to connect to Note n' Nib's systems and retrieve the right information or take the appropriate action.

AI applications that can interact with the real world in this way have a special name: agents.

10.6.1 What are agents?

Traditional AI chatbots follow a question-answer pattern, providing helpful but static responses. However, when users need personalized assistance—such as checking an order status, updating an address, or canceling an order—an informational bot falls short.

This is where *agents* come in. Unlike passive chatbots, AI agents—also called *agentic apps*—can reason, plan, and interact with external systems to complete tasks. They don't just retrieve knowledge; they take actions based on it. These agents often rely on *tool use*, meaning they can call APIs, run database queries, and even trigger workflows in real-world applications.

For example, instead of telling a customer to reach out to a customer care number for an order tracking number, an agent can fetch the tracking details and provide an update directly. Instead of directing a user to a cancellation policy page, it can process a cancellation request on their behalf.

In short, agents make AI practical by allowing it to interface with the systems people already use. The key to making an agent work effectively is a framework that enables it to reason and decide the next steps dynamically. One such popular framework is called *ReAct*, short for *Reasoning + Acting*.

10.6.2 The ReAct framework

To function as a true agent, an AI system must do more than just retrieve facts—it needs to reason about a situation, determine the right action, execute that action, and then incorporate the result into its next steps. The ReAct framework is designed to facilitate this process.

NOTE The ReAct AI framework is not to be confused with React, a Javascript toolkit for building web apps. It just so happens that we'll encounter the latter in chapter 13.

Figure 10.19 is a visual representation of the ReAct framework.

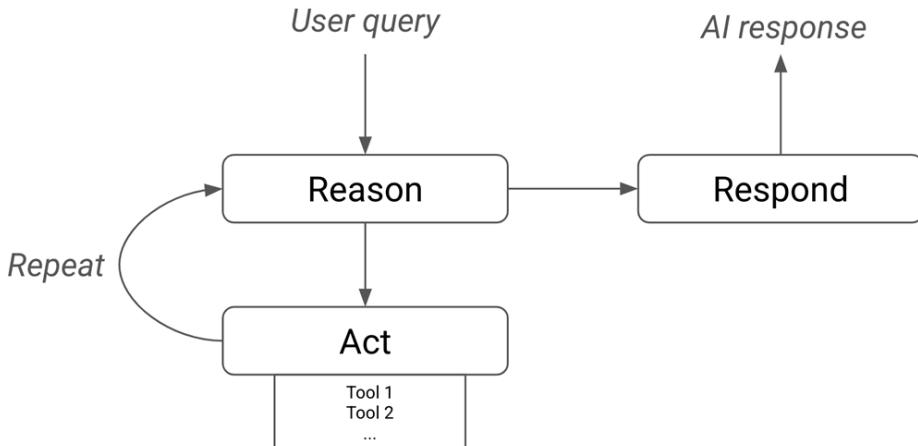


Figure 10.19 The ReAct framework

ReAct structures an AI agent's behavior as an interleaving of reasoning steps and actions:

1. **Reason:** The agent analyzes the user's query, breaks it down into logical steps, and determines what needs to be done.
2. **Act:** The agent takes a concrete action, such as calling an API or querying a database, to retrieve relevant data or execute a task.
3. **Reason:** The agent incorporates the action's results into its reasoning and decides whether further steps are necessary.
4. **Repeat:** If further steps are needed, the cycle repeats.
5. **Respond:** If no further steps are needed, respond to the user.

For example, consider a customer asking Nibby: "What's the status of my order?" Using the ReAct framework, the bot might follow these steps:

1. **Reason:** "The customer wants to check their order status. I need to fetch order details from the order database."
2. **Act:** Call Note & Nib's order management system to retrieve the order status.
3. **Reason:** "The system shows the order has shipped and is in transit. I should provide the expected delivery date."
4. **Respond:** Respond to the customer with the latest tracking details and estimated delivery date.

Creating tools our agent can use

The concept of *tools* is central to the process of developing a ReAct AI agent. In AI parlance, a tool is a function or API that an AI agent can call to perform real-world actions.

Referring back to the requirements we drafted, we want our bot to be able to track and cancel orders on behalf of Note n' Nib's customers. Of course, Note n' Nib is a fictional company with no real orders or customers.

This means we need some example data to work with. You can find this in the database.py file in the GitHub repository (https://github.com/aneevdavis/streamlit-in-action/tree/main/chapter_10/in_progress_06/). Copy the file to your working directory. Listing 10.11 shows a couple of excerpts from it.

Listing 10.11 database.py

```
users = {
    1: {
        "first_name": "Alice",
        "last_name": "Johnson",
        "date_of_birth": "1990-05-14",
        "email_address": "alice.johnson@example.com"
    },
    ...
}

orders = {
    101: {
        "user_id": 1,
        "order_placed_date": "2025-02-10",
        "order_status": "Shipped",
        "tracking_number": "TRK123456789",
        "items_purchased": ["RoyalQuill", "RedPinner"],
        "quantity": [1, 1],
        "shipping_address": "123 Main St, Springfield, IL, 62701",
        "expected_delivery_date": "2025-02-18"
    },
    ...
}
```

(chapter_10/in_progress_06/database.py in the GitHub repo)

The file has two dictionaries—`users` and `orders`—keyed on user ID and order ID, respectively. Highlighted in the listing is user ID 1 (Alice Johnson) and a corresponding order the user placed on 2025-02-10.

In a real application, this information would be stored in a database like PostgreSQL. Still, since our focus is on the mechanics of building an AI agent, the static data in `database.py` will suffice for our purposes.

There's another file in the repo called `tools.py`. Copy that one over too.

`tools.py` contains all the functions or tools that Nibby will be able to call. The file defines four such functions:

- `retrieve_user_id` looks up a user's user ID, given their email address and date of birth.
- `get_order_id` returns the ID of a particular order, given the user ID of the user that placed the order, and the date that they placed it. For simplicity, we assume that a user can only place one order on a particular day.
- `get_order_status` accepts an order ID and returns its order status, tracking number, and expected delivery date.
- `cancel_order` cancels an order, given an order ID.

We're not interested in the actual implementations of these functions, though you can read through them in `tools.py`. What matters is that Nibby needs to be able to call and use them correctly.

For this, we'll rely on type hints and docstrings. For instance, consider the definition of one of these functions in `tools.py`:

```
def retrieve_user_id(email: str, dob: str) -> str:
    """
    Look up a user's user ID, given their email address and date of birth.

    If the user is not found, return None.

    Args:
        email (str): The email address of the user.
        dob (str): The date of birth of the user in the format "YYYY-MM-DD".

    Returns:
        int: The user ID of the user, or None if the user is not found.
    """
    for user_id, user_info in users.items():
        if (user_info["email_address"] == email and
            user_info["date_of_birth"] == dob):
            return user_id
```

(chapter_10/in_progress_06/tools.py in the GitHub repo)

Notice how we're using type hints in the function signature (`(email: str, dob: str) -> str`), and explaining in detail what the function does in a docstring (the multi-line string after the signature), along with details about the arguments and return value.

These auxiliary items are what the AI model will use to figure out what tool to call when.

At the bottom of `tools.py`, there's this line that exports all the functions in the file as a list named `tools`:

```
tools = [retrieve_user_id, get_order_id, get_order_status, cancel_order]
```

We'll use the `tools` variable in our graph in a bit.

10.6.3 Making our graph agentic

Consider an example how Nibby might use the four tools we've made available to help a customer. Let's say the customer in the example data, Alice Johnson, wants to know the status of their order (order 101 shown in the excerpt from `database.py`). Here's how the interaction might proceed:

- **User:** "Hi, my name's Alice! I placed an order on Feb 10, 2025. Could you check on its status, please?"
- **Nibby (reasoning to itself):** "I have a tool called `get_order_status` that will give me the status of an order. To call it, I need an `order_id`. I don't have one, but I can get one by calling the `get_order_id` tool, which needs a `user_id` and an `order_placed_date`. The user says the `order_placed_date` is Feb 10, 2025, so I have that. I still need a `user_id`. To obtain *that*, I can call the `retrieve_user_id` tool, which requires an email and a date of birth. Therefore, I should ask the customer for this info."
- **Nibby:** "Could you give me your email address and date of birth?"
- **User:** "Sure, my email is alice.johnson@example.com and my date of birth is May 10, 1990."
- **Nibby (tool call):** Call the tool `retrieve_user_id` with the parameters `email=alice.johnson@example.com` and `dob=1990-05-10`
- `retrieve_user_id('alice.johnson@example.com', '1990-05-10'):` <returns the value 1, which is Alice's `user_id`>
- **Nibby (tool call):** Call the tool `get_order_id` with the parameters `user_id=1` and `order_placed_date=2025-02-10`
- `get_order_id(1, 1990-05-10):` <returns the value 101, the correct order ID>
- **Nibby (tool call):** Call the tool `get_order_status` with the parameter `order_id=101`
- **get_order_status(101):** <returns a dictionary: {'order_status': 'Shipped', 'tracking_number': 'TRK123456789', 'expected_delivery_date': '2025-02-18'}>
- **Nibby:** "Your order has been shipped and should arrive on Feb 18, 2025. You can use the tracking number TRK123456789 to track it."

Note how the bot needs to alternate between conversing with the user, reasoning about what needs to be done next, and issuing a call to a tool. How do we code all of this up? LangGraph actually makes this surprisingly easy.

First, let's modify `bot.py` to make our LLM aware of the tools available to it:

```
...
from vector_store import VectorStore
from tools import tools

class Bot:
    def __init__(self, api_keys, config):
        ...
        self.llm = self.get_llm().bind_tools(tools)
        ...
    ...
```

Here we're *binding* the tools we defined earlier—imported from `tools.py`—to our LLM object so that it knows they exist. As a result, if the LLM thinks it's appropriate, it can respond with a *tool call*.

In the example above, the tool calls are marked with "Nibby (tool call)". In practical terms, these are `AIMessages` produced by the LLM that have a property called `tool_calls` which contains information about any tools that the LLM wants us to call on its behalf and the parameters to call them with.

The binding logic uses the docstrings and type hints we specified in `tools.py` to explain to the LLM what each tool does and how to use it.

What about the graph itself? Listing 10.12 shows the changes required to `graph.py` to turn our bot into an agent.

Listing 10.12 graph.py (for an agentic app)

```

from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import START, StateGraph, MessagesState
from langgraph.prebuilt import tools_condition, ToolNode
from langchain_core.documents import Document
from langchain_core.messages import HumanMessage, SystemMessage
from prompts import *
from tools import tools

...
class SupportAgentGraph:
    ...
    def build_graph(self):
        ...
        builder.add_node("tools", ToolNode(tools))
        ...
        builder.add_edge("augment", "assistant")
        builder.add_conditional_edges("assistant", tools_condition)
        builder.add_edge("tools", "assistant")

        return builder.compile(checkpointer=memory)
    ...

```

(chapter_10/in_progress_06/graph.py in the GitHub repo)

Incredibly, all we needed to do was to add three lines to the `build_graph` method, and import a few extra items!

Let's go through the line additions, starting with the first one:

```
builder.add_node("tools", ToolNode(tools))
```

This adds a node named `tools` to our graph. `ToolNode` is a node already built into LangGraph, so we don't have to define it ourselves. It essentially does the following:

- It takes the last message in the messages list (from the graph state) and executes any tool calls in it based on the list of tools we pass it.
- Appends the value returned by the tool(s) as a `ToolMessage`—like `HumanMessage` and `AIMessage`—to the messages list.

At the end of a `ToolNode`, the last message in the messages variable is a `ToolMessage` representing the output of calling a tool.

We now have all the *pieces* we need, but how do we orchestrate the kind of thinking process outlined in the example interaction at the beginning of this section?

Before we get into that, turn your attention to figure 10.20, which is what our graph will look like at the end of these changes:

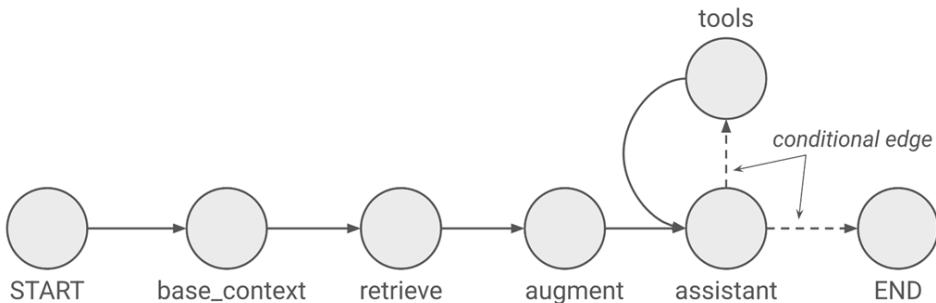


Figure 10.20 Our graph now has a tools node and a conditional edge

Notice that there are now *two* lines flowing out from the `assistant` node—one goes to `END` as before, while another goes to our newly added `tools` node.

In LangGraph, this is known as a *conditional edge*. We can choose the next node to execute from multiple options depending on a specified condition. A conditional edge is implemented as a function that takes the following form:

```

def some_condition(state):
    # Branching
    if <something is true>:
        return "name_of_node_1"
    elif <something else is true>:
        return "name_of_node_2"
    ...
  
```

In our case, we don't actually need to build our own conditional edge because LangGraph already has exactly what we want:

```
builder.add_conditional_edges("assistant", tools_condition)
```

`tools_condition`—imported from `langgraph.prebuilt`—simply routes to our `ToolNode` (named `tools`) if the last message in the conversation—i.e. the LLM's response—contains any tool calls, or to `END` otherwise.

Since `tools_condition` already has the logic to route to `END`, we can remove the earlier line that created a direct (non-conditional) edge between `assistant` and `END`.

Once the `ToolNode` executes, we need the LLM to read the return value and decide what to do with it—whether it's calling *another* tool, or responding to the user.

Therefore, we create a *loop* in the graph by connecting the tools node *back* to assistant:

```
builder.add_edge("tools", "assistant")
```

And that's all it takes! Now when a request comes in, the LLM will reason about what to do. If it decides to call a tool, it'll put a tool call in its response, causing `tools_condition` to route to the `ToolNode` which executes the call. Since `ToolNode` has a direct edge to `assistant`, the LLM will get the updated list of messages with the appended `ToolMessage` and can again reason about what to do with the response.

If the LLM decides it doesn't need to call any more tools—or that the next step in the process is to get some information from the user—it won't include any tool calls in its response, which means that `tools_condition` will route to `END`, and the final message will be displayed to the user.

While that's all that's required to get the bot to work correctly, there's a last change we need to make in `graph.py` that's related to what gets shown to the customer in the frontend.

As the above paragraphs hopefully make clear, the communication between the `assistant` and `tools` nodes happens through the `messages` variable in the graph state, and consists of internal messages of two types: `AIMessages` containing tool calls, and `ToolMessages` containing tool return values.

Since we don't want to expose these internal messages to the user of our Streamlit app, we need to hide them when we pass the conversation history back. Recall that this history is relayed through the `get_history` method in `bot.py`, which calls the `get_conversation` method in `graph.py`

Let's make the appropriate changes in `graph.py` to remove these messages:

```
...
@staticmethod
def is_internal_message(msg):
    return msg.type == "tool" or "tool_calls" in msg.additional_kwargs

def get_conversation(self):
    state = self.graph.get_state(self.config)
    if "messages" not in state.values:
        return []
    messages = state.values["messages"]
    return [msg for msg in messages if not self.is_internal_message(msg)]
```

(chapter_10/in_progress_06/graph.py in the GitHub repo)

First, we define `is_internal_message`, a static method that determines whether a message that we pass it is an "internal" one, i.e., one that's not fit to show the user. Above, we're defining an internal message as one that either has the type "`tool`"—which is true of `ToolMessages`—or has a "`tool_calls`" property (within `additional_kwargs`, a property that the LLM will use to set metadata within a message).

Then, rather than returning all the messages from the state in `get_conversation`, we now filter for the non-internal messages and only return those.

With that out of the way, re-run the app and test out its new capabilities! Figure 10.21 shows an example.

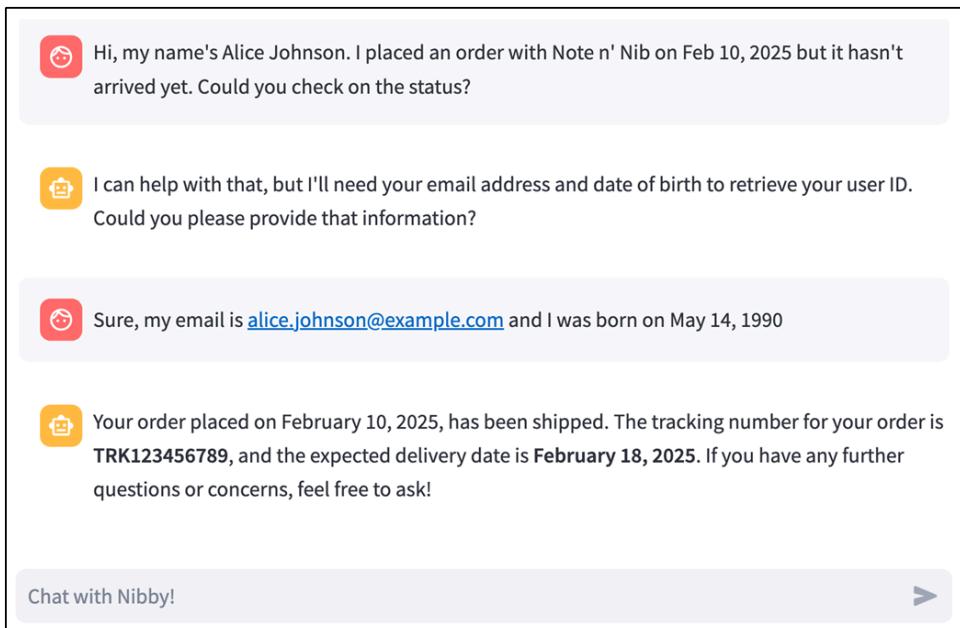


Figure 10.21 Nibby can handle real-world actions like tracking and canceling orders (see `chapter_10/in_progress_06` in the GitHub repo for the full code).

By allowing Nibby to access external tools, we've given it superpowers and saved Note n' Nib's customer support division some serious time!

This has been our most advanced app yet. If you've been following along and working on these projects yourself, you probably appreciate that the more complex an app becomes, the more things can go wrong in the real world when users actually start interacting with it. In the next chapter, we'll explore how to catch these problems beforehand and test your app to make it as robust as possible.

10.7 Summary

- Real-world AI applications require advanced capabilities, such as knowledge retrieval and action execution.

- LangGraph structures AI workflows as graphs; nodes represent steps in the AI process, and edges define the flow between them.
- Each node in LangGraph takes in the graph state and modifies it.
- `st.chat_input` renders a textbox where users can type messages.
- `st.chat_message` displays human and AI messages appropriately.
- LangGraph uses checkpointers to persist information across multiple graph executions.
- It's important to instruct the LLM explicitly to stay on-topic and ignore irrelevant requests; the system message is a good place for this.
- Embeddings involve converting an object—such as a piece of text—into lists of numbers called vectors to find relevant content using similarity search.
- Retrieval Augmented Generation (RAG) is a technique for providing custom knowledge to a pre-trained LLM by retrieving chunks of context relevant to the user query from a vector database like Pinecone.
- A *tool* is simply a well-documented function that an LLM can choose to call in response to a user query.
- An agentic app—or simply an *agent*—is one that can use tools to interact with the real world.
- LangGraph makes it extremely easy to write agentic apps through its `ToolNode` and `tools_condition` abstractions.

Appendix A. Installing Python and Streamlit

This appendix covers

- How to check what version of Python you have installed (if at any)
- Installing Python on macOS and Windows
- Using pip to install Streamlit

Depending on what operating system you're using, you may have to follow slightly different steps to get Streamlit working on your computer.

This guide covers these steps for the two dominant desktop operating systems, macOS and Windows.

A.1 Checking your current Python version

Streamlit supports versions 3.8 and above of Python, so if you have a version older than that, you need to install a newer one.

To check what version of Python you have installed in your system (or if you have Python at all), open up a terminal window (this is the program Terminal on macOS, and Command Prompt—or better yet, PowerShell—on Windows) and enter:

```
python --version
```

```
python -V
```

```
'python': command not found
```

```
'python' is not recognized as an internal or external command, operable program or batch file.
```

```
'Python' was not found; run without arguments to install from the Microsoft Store
```

If you *do* have Python installed, the command you typed in will give you a version number like:

```
Python 3.7.3
```

If this version number is 3.8 or above (e.g., 3.8.0 or 3.12.2), you're good!

A.2 Installing the right Python version

A.2.1 On Windows

If you're using Windows, you can use the Microsoft Store to install Python. To do this, simply open the Microsoft Store and search for "python." As seen in figure A.1, the search returns several versions of Python. Select the latest version you can find (in this case Python 3.12) and click "Get."

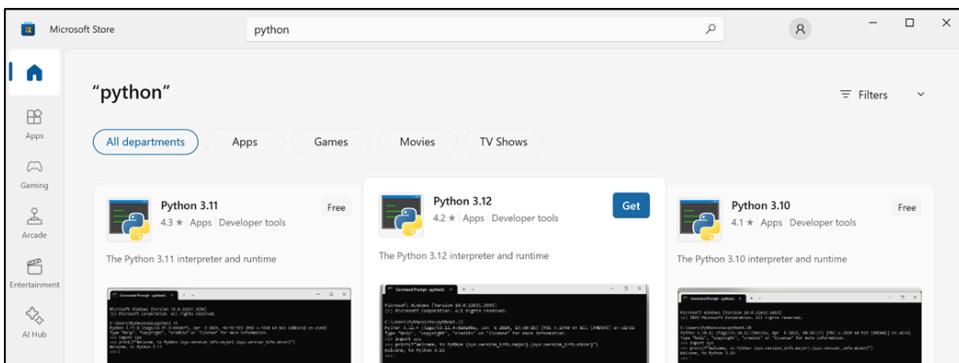


Figure A.1 A search for "python" in the Microsoft Store gives you an easy way to install the latest Python

Once the process finishes, enter `python -v` in the Command Prompt again to see if the default Python version on your system has been updated. If it gives you the version you just installed, great!

If it gives you an older version, the easiest way to proceed is simply to uninstall the lower version. You can do this in the Settings app by going to "Apps & features."

If you type `python -v` now, you should be able to see the newer version.

A.2.2 On macOS

Python comes pre-installed on macOS, so if you're here, you probably need to install a newer version.

There are several ways to do this, but one of the easiest is to go to <https://www.python.org/downloads/macos/> and download an installer.

On that page, pick the installer for the latest version listed under "Stable Releases." For instance, in figure A.2, you would click on "Download macOS 64-bit universal2 installer" under "Python 3.12.4."

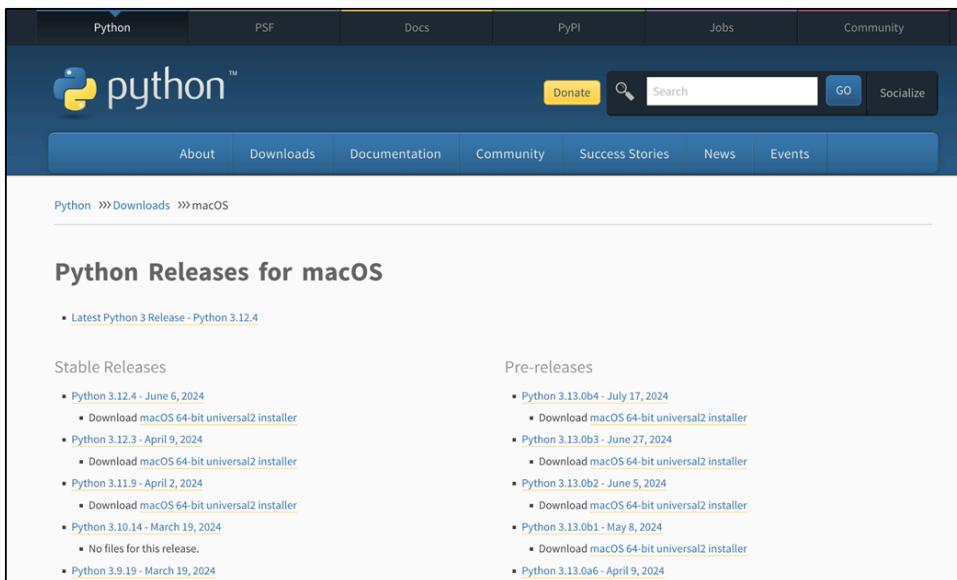


Figure A.2 The URL <https://www.python.org/downloads/macos/> gives you a link to download Python on macOS

Once the download is done, open the installer and follow the on-screen instructions.

When the installation is complete, open Terminal and type `python -V` again. If it displays the version you installed, you're done.

Otherwise, you'll need to perform a few extra steps to make sure that when you type `python` or `pip` in the command line, it's calling the right version.

Again, there are multiple ways to do this, but here's an easy one:

1. Type `nano ~/.bash_profile` to open the text file `.bash_profile` in a text editor.
2. At the bottom, append the lines shown in listing A.1.
 - o **Note:** Replace the version numbers in these lines with the version you actually installed! For example, if you installed Python 3.11, you would write `python3.11` and `pip3.11`.

Listing A.1

```
alias python=python3.12
alias pip=pip3.12
```

3. Restart the terminal

When you type `python -v`, it should now give you the right version.

A.2.3 What if I want to have multiple versions of Python on my system?

It's quite common for Python developers to have more than one version of Python installed, especially if they're working on multiple projects.

Managing this can be quite painful, so if you're on this path, I recommend the tool `pyenv` (or `pyenv-win` for Windows). Detailed installation instructions are out of the scope of this appendix, but you can find them at <https://github.com/pyenv/pyenv> (and <https://github.com/pyenv-win/pyenv-win> for `pyenv-win`).

Once you have `pyenv`, it's really easy to install, manage, and switch between different versions of Python.

For instance, to see what versions of Python you currently have installed, you could type

```
pyenv versions
```

To install Python 3.12.2, you would type

```
pyenv install 3.12.2
```

To switch to that version of Python, you could do:

```
pyenv global 3.12.2
```

And to switch back:

```
pyenv global system
```

Easy peasy.

A.3 Installing Streamlit

Once you have Python installed, you can use `pip` to install Streamlit.

`pip` (a recursive acronym for "Pip Installs Packages") is a package manager for Python, used to install and manage Python libraries.

It comes bundled with Python starting with version 3.4, so as long as you've followed the Python installation instructions above, you don't have to install pip separately.

Once you have pip, adding popular third party libraries (like Streamlit) to your Python installation becomes simple; you just have to type the following in your terminal:

```
pip install <package_name>
```

So, to install Streamlit, you would type:

```
pip install streamlit
```

To execute this command, pip will find the latest version of the Streamlit library that's compatible with your version of Python and install that version, plus its dependencies.

You may also sometimes want to install a *specific* version of a library. In such cases, you can add that condition to your command. For example, to install version 1.23.0 of Streamlit:

```
pip install streamlit==1.23.0
```

Perhaps more commonly, you may want to specify a *minimum* version of the library to install, for instance if you are relying on a feature that was introduced in that version. This is as simple as:

```
pip install streamlit<=1.32.0
```

(to install a version of Streamlit that's at least 1.32.0).

Once this is done, try the following if you're on macOS to see what version of Streamlit you have installed:

```
pip list | grep streamlit
```

Or the following if you're on Windows:

```
pip list | findstr streamlit
```

This should give you output similar to the following:

```
streamlit      1.33.0
```

This signals that Streamlit has been installed correctly as a Python module. However, there's one more step to take before we can run Streamlit apps effortlessly.

A.4 Getting the `streamlit` command to work

When you start creating Streamlit apps, you'll be running them using the `streamlit run` command in a terminal window like this:

```
streamlit run <path to filename.py>
```

At this point, the `streamlit` module may already be installed correctly, but you *may* need to perform additional steps to get this command to work.

First, type the following to check if it's already working:

```
streamlit --version
```

You should see something similar to the following:

```
Streamlit, version 1.33.0
```

If you see this (or a different version number), the `streamlit` command is already working, and you can skip the rest of this section.

If instead you're seeing a message that tells you that "streamlit" is not recognized as a command, you'll need to take further steps.

The issue here is that your computer doesn't know the command "streamlit" yet. You need to let it know where to find it by adding the path to the command in an *environment variable*.

The steps to do this differ by operating system.

A.4.1 On Windows

To start, we need to figure out where on your computer the `streamlit` executable was installed.

For this, spin up a new Python shell by typing `python` into the command line. Then enter the following:

```
>>> import os  
>>> import streamlit  
>>> print(os.path.dirname(streamlit.__file__))
```

The last command should give you a path to a file in your computer. For example, when I typed this in, I got:

```
C:\Users\Aneev  
Kochakadan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n2kfra8p0\Loca  
packages\Python312\site-packages\streamlit
```

This is the path where the `streamlit` module is installed, which is not quite what we're looking for. What we need is the path to the directory that contains the `streamlit executable`, which we can derive by replacing the ending `site-packages\streamlit` from the path above with `Scripts`.

In my case, the path to the executable was:

```
C:\Users\Aneev  
Kochakadan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n2kfra8p0\Loca  
packages\Python312\Scripts
```

We now need to add this path to the Path environment variable. To do this, first open the Start Menu and search for "Environment Variables."

Then select "Edit the system environment variables." This should open up the "Advanced" tab in the System Properties dialog as shown in figure A.3.

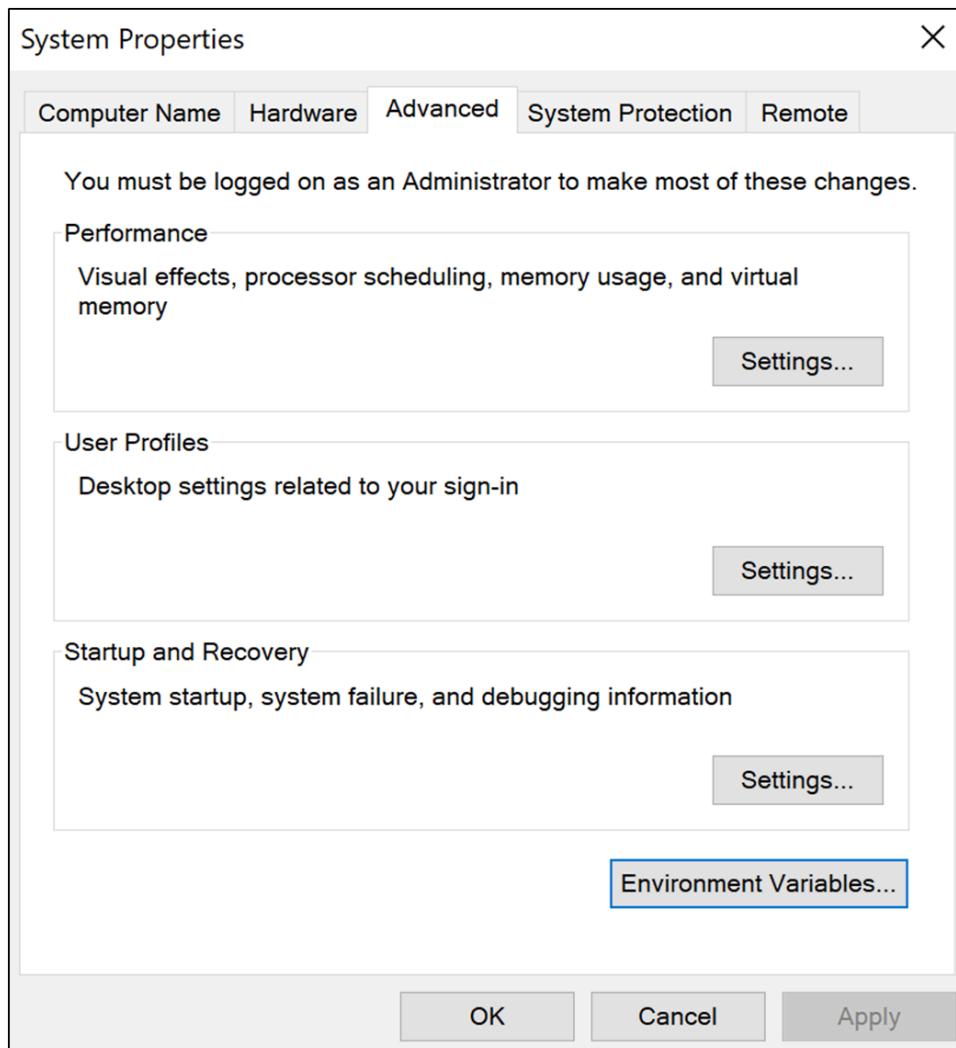


Figure A.3 The "Advanced" tab in the "System Properties" dialog has a button at the bottom to manage your environment variables

Click "Environment Variables..." at the bottom to see a list of your current environment variables. The one we care about is the "Path" variable under your user variables (the top box), so go ahead and select that one and click "Edit..." (figure A.4).

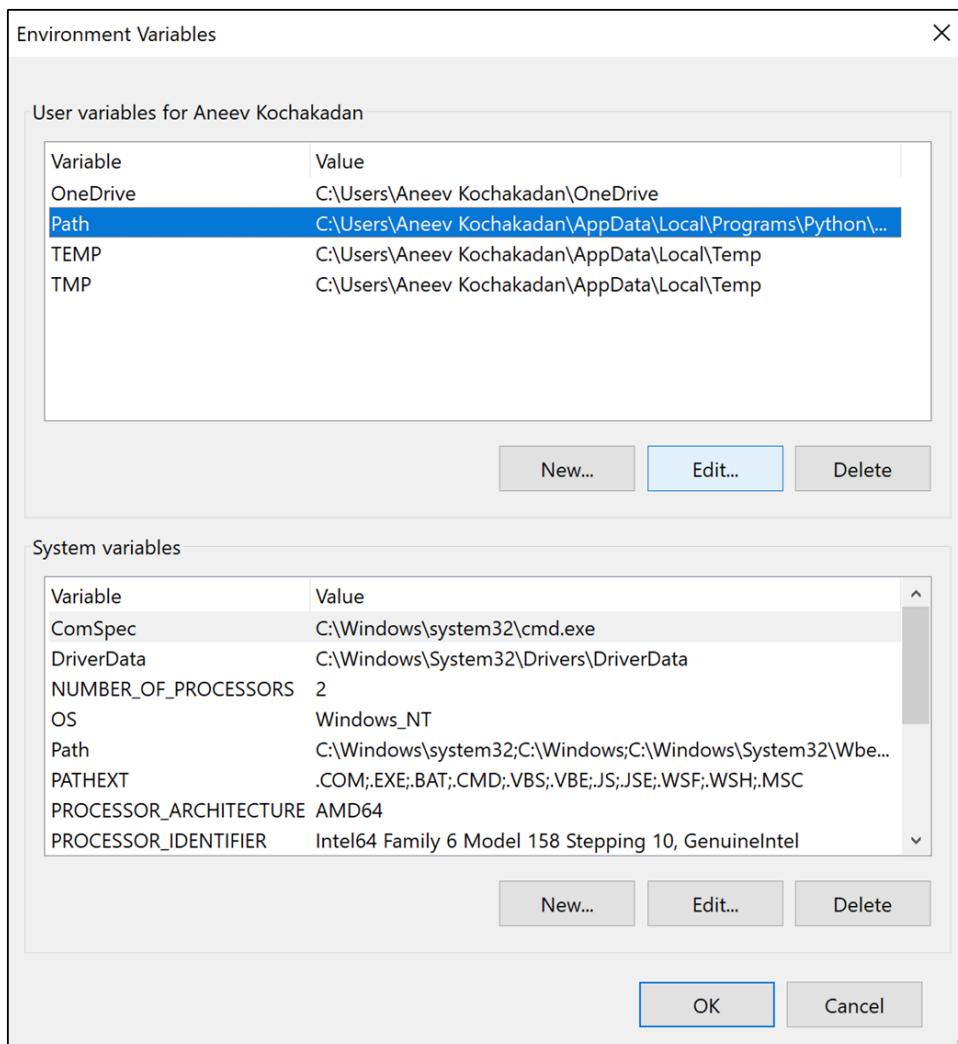


Figure A.4 The "Path" environment variable is the one we need to edit

In the dialog that now opens up, click "New" and enter the path you noted earlier (in my case, it was C:\Users\Aneev Kochakadan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n2kfra8p0\LocalCache\local-packages\Python312\Scripts) as shown in figure A.5.

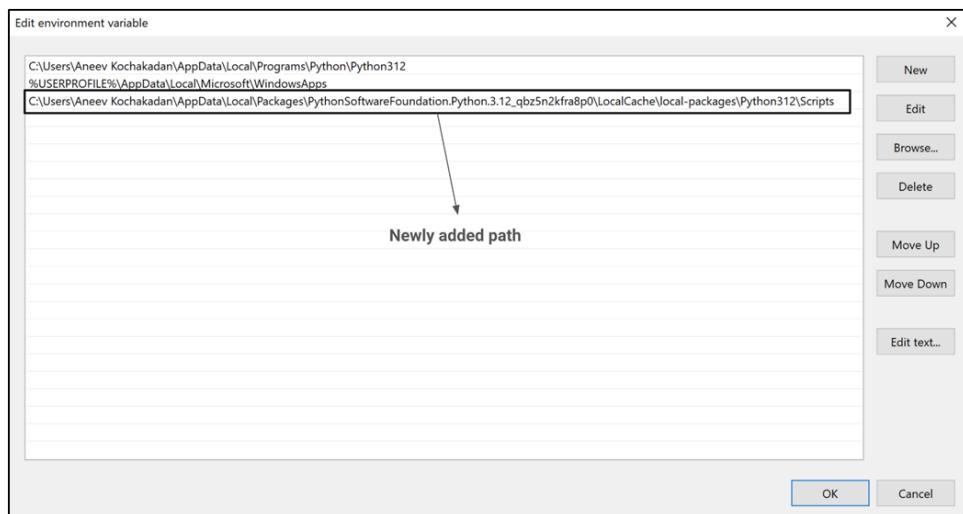


Figure A.5 You can add a new path to the "Path" environment variable by clicking "New"

Click "OK" to apply the changes, and close the window.

Restart your terminal and try entering `streamlit --version` again. This time, it should give you a version number, which means that the `streamlit` command is now working.

A.4.2 On macOS

On macOS, just as in Windows, we first need to find where the `streamlit` executable is, so go ahead and open a Python shell by typing `python` in a terminal, and then enter the following commands:

```
>>> import os
>>> import streamlit
>>> print(os.path.dirname(streamlit.__file__))
```

This should print out a path to where the `streamlit` module is installed. For example, on my computer, I saw:

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/streamlit
```

This is the path to where the `streamlit` module is installed. The path we really need, however, is the path to the `streamlit` executable, which we can get by replacing `lib/python3.12/site-packages/streamlit` at the end of the above path with `bin`. In my case, this works out to be:

```
/Library/Frameworks/Python.framework/Versions/3.12/bin
```

To wrap up, we need to add this path to the PATH environment variable. For this, follow these steps:

1. Type `nano ~/.bash_profile` to open the file `.bash_profile` in a text editor.
2. At the bottom, append the following lines:

```
PATH="/Library/Frameworks/Python.framework/Versions/3.12/bin:${PATH}"
export PATH
```

Restart your terminal window and enter `streamlit --version`. It should work this time and give the version number you installed.