Lesson 43: Streamlit Dashboards

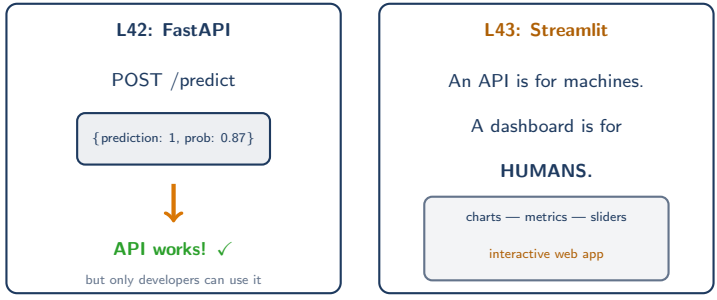
Data Science with Python – BSc Course

Data Science Program

BSc Course

45 Minutes

Previously: Your Model Goes Online



L42 built the API. Now: build a visual interface that anyone can use – no coding required.

APIs serve machines. Dashboards serve people. Both are essential.

Learning Objectives

The Problem: APIs are great for machine-to-machine communication. But what about humans? How do we create interactive web apps where users can explore data and get predictions – without learning JavaScript?

After this lesson, you will be able to:

- Build interactive web apps with Streamlit (Python only)
- Add widgets for user input (sliders, dropdowns, file uploads)
- Display charts, DataFrames, and model predictions
- Create financial dashboards for portfolio analysis

Finance Application: Interactive stock dashboards, portfolio analyzers, ML model demos

An API is for Machines. A Dashboard is for HUMANS.

The Problem:

- Your manager wants to see model predictions – but can't use curl
- Your client wants to explore different scenarios – with sliders
- Your team needs a demo – not a Jupyter notebook on a projector

The Solution: Streamlit

- Streamlit = Python library that turns scripts into web apps
- No HTML, CSS, or JavaScript – pure Python
- Write a script → run `streamlit run app.py` → web app

Analogy – PowerPoint for Data Apps:

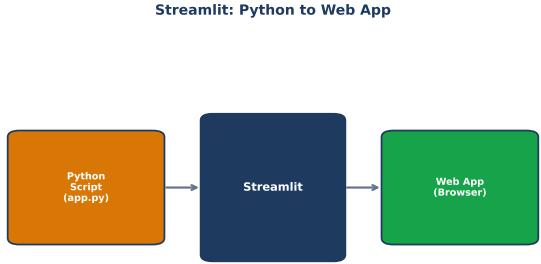
- PowerPoint: create presentations without web design skills
- Streamlit: create web apps without frontend development skills
- Focus on your DATA and MODEL, not on HTML/CSS/JS

Streamlit: from Python script to interactive web app in minutes. No frontend skills.

How Streamlit Works

The Reactive Execution Model

- Write Python top-to-bottom, like a normal script
- When a user interacts with a widget, the **entire script reruns**
- Variables update automatically with new widget values



Streamlit reruns your entire script on every interaction – simple but powerful

Streamlit Basics

Core Display Functions:

- `st.title('My App')` – page title
- `st.write('Hello')` – text, DataFrames, charts (auto-detects type)
- `st.dataframe(df)` – interactive table with sorting
- `st.metric('Price', '$142', delta='+2.3%')` – KPI card

Minimal App (app.py):

- `import streamlit as st`
- `import pandas as pd`
- `st.title('Stock Dashboard')`
- `df = pd.read_csv('stocks.csv')`
- `st.dataframe(df)`

Run: `streamlit run app.py` → opens browser at `localhost:8501`

---

st.write() is the Swiss Army knife – handles text, DataFrames, charts, and more

Widgets: Interactive Controls

Every Widget Returns Its Current Value

- User changes a slider → script reruns → variables update
- No callbacks, no event handlers – just variables

Widgets return values directly. Change a widget = script reruns with new values.

Widget Code Examples

Common Widgets:

- **Slider:** `risk = st.slider('Risk tolerance', 0.0, 1.0, 0.5)`
- **Select:** `stock = st.selectbox('Stock', ['AAPL', 'GOOG', 'MSFT'])`
- **Number:** `amount = st.number_input('Investment', min_value=0)`
- **Date:** `date = st.date_input('Start date')`
- **File:** `file = st.file_uploader('Upload CSV', type='csv')`
- **Button:** `if st.button('Predict'): run_model()`

Key Pattern:

- `risk = st.slider('Risk', 0.0, 1.0, 0.5)`
- `st.write(f'Selected risk: {risk}')` – updates live!

Finance Example:

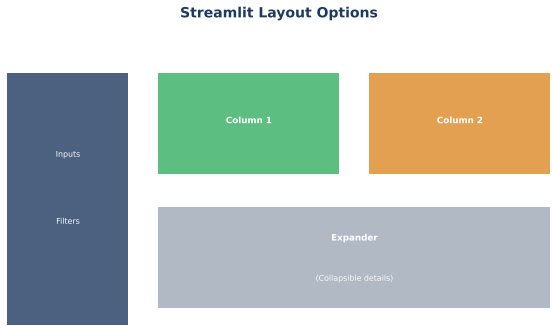
- `ticker = st.selectbox('Ticker', ['AAPL', 'MSFT', 'JPM'])`
- `ma_period = st.slider('Moving Average', 5, 200, 50)`

Each widget is one line of code. Returns the current value. Updates on interaction.

Layouts: Organizing Your App

Structure Your Dashboard

- Columns, sidebar, tabs, expanders, containers
- Sidebar for controls, main area for content



Sidebar for controls, main area for charts. Tabs for multiple views.

Layout Code Examples

Columns – Side by Side:

- `col1, col2 = st.columns(2)`
- `col1.metric('Price', '$142.50')`
- `col2.metric('Volume', '1.2M')`

Sidebar – Settings Panel:

- `ticker = st.sidebar.selectbox('Stock', tickers)`
- `start = st.sidebar.date_input('Start')`

Tabs – Multiple Views:

- `tab1, tab2 = st.tabs(['Charts', 'Data'])`
- `with tab1: st.line_chart(prices)`
- `with tab2: st.dataframe(df)`

Common Finance Layout:

- Sidebar: stock selector, date range, parameters
- Main: metrics on top, chart in middle, data at bottom
- Tabs: Overview — Technical Analysis — Predictions

Charts Integration

Display Visualizations:

- **Built-in:** `st.line_chart(df)`, `st.bar_chart(df)` – one line, quick
- **Matplotlib:** `st.pyplot(fig)` – publication-quality static charts
- **Plotly:** `st.plotly_chart(fig)` – interactive (zoom, hover, pan)
- **Altair:** `st.altair_chart(chart)` – declarative grammar

Finance Chart Examples:

- Stock price line chart: `st.line_chart(prices_df)`
- Candlestick: `st.plotly_chart(go.Candlestick(...))`
- Portfolio comparison: `st.line_chart(returns_df)`

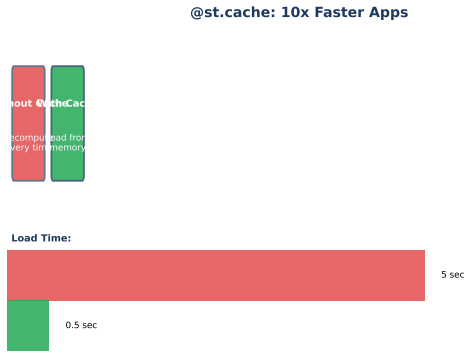
Which Library?

- Quick exploration → `st.line_chart()` (built-in)
- Interactive analysis → Plotly
- Publication quality → matplotlib

Plotly for interactivity, matplotlib for quality, built-in for speed

Caching: Speed Up Your App

The Problem: Streamlit reruns the ENTIRE script on every widget interaction. Loading data or models every time is slow!



Solution: `@st.cache_data` for DataFrames. `@st.cache_resource` for models.

Cache data with @st.cache_data, models with @st.cache_resource. Essential!

Caching: Code Examples

Cache Data Loading:

```
@st.cache_data
def load_data():
  return pd.read_csv('big_file.csv')
```

Cache ML Models:

```
@st.cache_resource
def load_model():
  return joblib.load('model.joblib')
```

How It Works:

1. First call: runs the function, caches the result
2. Subsequent calls: returns cached result instantly
3. Cache invalidation: change function code or clear manually

Key Difference:

- `@st.cache_data` – for serializable data (DataFrames, arrays, strings)
- `@st.cache_resource` – for non-serializable objects (models, DB connections)

Checkpoint: Display Functions

Which function would you use?

1. Show a stock's current price with daily change indicator?
2. Display a DataFrame with sorting and filtering?
3. Show a matplotlib chart?
4. Display a success message after a prediction?

Answers:

1. `st.metric('AAPL', '$142', delta='+2.3%')` – KPI with change
2. `st.dataframe(df)` – interactive table
3. `st.pyplot(fig)` – static chart
4. `st.success('Buy signal detected!')` – green alert box

Other alert types: `st.warning()`, `st.error()`, `st.info()`

---

st.metric for KPIs, st.dataframe for tables, st.pyplot for charts, st.success for alerts

Multi-Page Apps

Structure for Larger Applications:

- Streamlit supports multi-page apps natively
- Each page is a separate Python file in a pages/ folder
- Navigation appears automatically in the sidebar

File Structure:

- `app.py` – main page (home/landing)
- `pages/1_Price_Analysis.py` – page 1
- `pages/2_ML_Predictions.py` – page 2
- `pages/3_Portfolio.py` – page 3

Navigation:

- Number prefix controls order: 1_, 2_, 3_
- Underscores become spaces in the sidebar
- Each page has its own widgets and state

---

pages/ folder = automatic multi-page navigation. One file per page.

Session State: Remembering Values

The Problem: Script reruns on every interaction. How do you remember values across reruns?

Solution: st.session_state

```
if 'count' not in st.session_state:
  st.session_state.count = 0
if st.button('Increment'):
  st.session_state.count += 1
st.write(f'Count: {st.session_state.count}')
```

Finance Use Cases:

- Store user's portfolio across page navigations
- Remember selected stocks between tab switches
- Track prediction history during a session
- Maintain login state for authenticated dashboards

Without session_state: count resets to 0 every click.
With session_state: count persists across reruns.

---

st.session_state persists values across script reruns. Essential for stateful apps.

Finance: Portfolio Dashboard

Building a Stock Portfolio Analyzer:

1. **Sidebar:** Stock ticker selector, date range, MA periods
2. **Header:** Current price + daily change with `st.metric()`
3. **Tab 1 – Price:** Candlestick chart with moving averages
4. **Tab 2 – Technical:** RSI, MACD, Bollinger Bands
5. **Tab 3 – ML:** Model prediction with confidence score
6. **Bottom:** Raw data table with download button

Key Code:

- `ticker = st.sidebar.selectbox('Stock', tickers)`
- `st.plotly_chart(candlestick_fig)`
- `st.download_button('Download CSV', data.to_csv())`

A professional financial dashboard in approximately 100 lines of Python

Finance: Backtesting Interface

Interactive Backtesting Dashboard:

- **Inputs:** Strategy parameters via sliders (MA periods, thresholds)
- **Date range:** Start/end date pickers for historical window
- **Output:** Equity curve, drawdown chart, performance metrics
- **Comparison:** Strategy vs buy-and-hold benchmark

Code Pattern:

- `fast_ma = st.slider('Fast MA', 5, 50, 20)`
- `slow_ma = st.slider('Slow MA', 20, 200, 50)`
- `results = backtest(data, fast_ma, slow_ma)`
- `st.metric('Total Return', f'{results.total_return:.1%}')`
- `st.line_chart(results.equity_curve)`

Power: User adjusts sliders → backtest reruns instantly → results update live

---

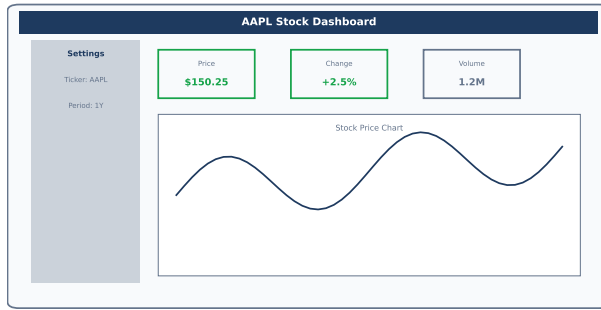
Sliders + caching = interactive backtesting that updates in real time

Dashboard Architecture

Putting It All Together

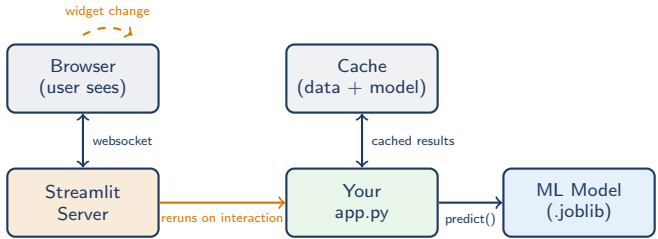
- Model loaded with `@st.cache_resource`
- User input via sidebar widgets
- Predictions displayed with metrics and charts

Finance Dashboard Example



Sidebar controls + cached model + live charts = complete ML dashboard

Streamlit Under the Hood



Flow: User interacts → websocket message → script reruns → cached data reused → new output sent to browser.

Streamlit = Python backend + React frontend + websocket communication

Deployment Options

Where to Host Your Streamlit App:

- **Streamlit Cloud** (free):
  - Connect GitHub repo, click deploy
  - Free tier: 1GB RAM, public repos
  - Best for demos and student projects
- **Hugging Face Spaces** (free):
  - Push to HF repo, auto-deploys
  - Great for ML model demos
- **Docker + Cloud** (production):
  - Full control, custom resources
  - AWS, GCP, Azure, Heroku
  - Covered in L44

For This Course: Streamlit Cloud is the simplest path to a live URL.

Streamlit Cloud for demos, Docker for production. L44 covers cloud deployment.

Hands-On Exercise (25 min)

Task: Build a Stock Analysis Dashboard

1. Create Streamlit app with sidebar stock selector
2. Display price chart with moving averages
3. Add sliders for MA period selection
4. Load your saved model and add prediction tab
5. Add download button for data export

Requirements:

- At least 3 interactive widgets
- At least 1 chart (line, bar, or plotly)
- Use `@st.cache_data` for data loading
- Use `st.metric()` for at least one KPI

Deliverable: Running Streamlit app with at least 3 interactive elements.

Extension: Add portfolio comparison – select multiple stocks and overlay returns

Models People Can See

Your model has a face now. Next step: put it on the internet so anyone can access it.

A dashboard turns "interesting notebook" into "impressive product"

Lesson Summary

Problem Solved: We can now create interactive web apps for data exploration and ML predictions using only Python.

Key Takeaways:

- **Streamlit:** Python script $\rightarrow$ web app (no HTML/CSS/JS)
- **Widgets:** sliders, selects, inputs – each returns its value
- **Caching:** `@st.cache_data` for data, `@st.cache_resource` for models
- **Layouts:** sidebar, columns, tabs for organization
- **Session state:** persist values across script reruns

Next Lesson: Cloud Deployment (L44) – putting your app on the internet for the world

---

L41: Save. L42: API. L43: Dashboard. L44: Cloud – the deployment pipeline completes.