DATA SCIENCE

Building a Modern Dashboard with Python and Gradio

Data insights made simple

Thomas Reid

Jun 4, 2025 13 min read

Image by AI (Gpt4-o)

This is the second in a short series on developing data dashboards using the latest Python-based GUI

development tools, Streamlit, Gradio, and Taipy.

The source dataset for each dashboard will be the same, but stored in different formats. As much as possible, I'll also try to make the actual dashboard layouts for each tool resemble each other and have the same functionality.

In the first part of this series, I created a Streamlit version of the dashboard that retrieves its data from a local PostgreSQL database. You can view that article here.

This time, we're exploring the use of the Gradio library.

The data for this dashboard will be in a local CSV file, and Pandas will be our primary data processing engine.

> If you want to see a quick demo of the app, I have deployed it to Hugging Face Spaces. You can run it using the link below, but note

that the two input date picker pop-ups do not work due to a known bug in the Hugging Face environment. This is only the case for deployed apps on HF, you can still change the dates manually. Running the app locally works fine and doesn't have this issue.

Dashboard demo on HuggingFace

What is Gradio?

Gradio is an open-source Python package that simplifies the process of building demos or web applications for machine learning models, APIs, or any Python function. With it, you can create demos or web applications without needing JavaScript, CSS, or web hosting experience. By writing just a few lines of Python code, you can unlock the power of Gradio and seamlessly showcase your machine-learning models to a broader audience.

Gradio simplifies the development process by providing an intuitive framework that eliminates the complexities associated with building user interfaces from scratch. Whether you are a machine learning developer, researcher, or enthusiast, Gradio allows you to create beautiful and interactive demos that enhance the understanding and accessibility of your machine learning models.

This open-source Python package helps you bridge the gap between your machine learning expertise and a broader audience, making your models accessible and actionable.

What we'll develop

We're developing a data dashboard. Our source data will be a single CSV file containing 100,000 synthetic sales records.

The actual source of the data isn't ***that*** important. It could just as

easily be a text file, an Excel file, SQLite, or any database you can connect to.

This is what our final dashboard will look like.

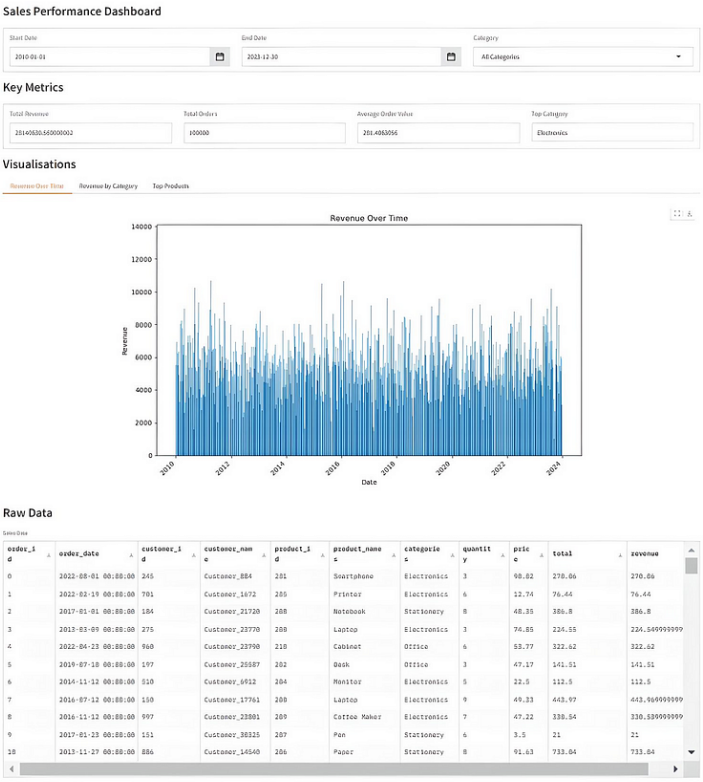


Image by Author

There are four main sections.

- The top row enables the user to select specific start and end dates and/or product categories using date pickers and a drop-down list, respectively.

- The second row — Key metrics — shows a top-level summary of the chosen data.
- The Visualisation section allows the user to select one of three graphs to display the input dataset.
- The raw data section is precisely what it claims to be. This tabular representation of the chosen data effectively shows a snapshot of the underlying CSV data file.

Using the dashboard is easy. Initially, stats for the whole data set are displayed. The user can then narrow the data focus using the three filter fields at the top of the display. The graphs, key metrics, and raw data sections dynamically update to reflect the user's choices in the filter fields.

The underlying data

As mentioned, the dashboard's

source data is contained in a single comma-separated values (CSV) file. The data consists of 100,000 synthetic sales-related records. Here are the first ten records of the file to give you an idea of what it looks like.

```
+----------+------------+----------
| order_id | order_date | customer_i
+----------+------------+----------
| 0        | 01/08/2022 | 245
| 1        | 19/02/2022 | 701
| 2        | 01/01/2017 | 184
| 3        | 09/03/2013 | 275
| 4        | 23/04/2022 | 960
| 5        | 10/07/2019 | 197
| 6        | 12/11/2014 | 510
| 7        | 12/07/2016 | 150
| 8        | 12/11/2016 | 997
| 9        | 23/01/2017 | 151
+----------+------------+----------
```

And here is some Python code you can use to generate a similar dataset. Ensure that both the NumPy and Pandas libraries are installed first.

```
# generate the 100K record CSV file
#
import polars as pl
```

```
import numpy as np
from datetime import datetime, timed

def generate(nrows: int, filename: s
    names = np.asarray(
        [
            "Laptop",
            "Smartphone",
            "Desk",
            "Chair",
            "Monitor",
            "Printer",
            "Paper",
            "Pen",
            "Notebook",
            "Coffee Maker",
            "Cabinet",
            "Plastic Cups",
        ]
    )
    categories = np.asarray(
        [
            "Electronics",
            "Electronics",
            "Office",
            "Office",
            "Electronics",
            "Electronics",
            "Stationery",
            "Stationery",
            "Stationery",
            "Electronics",
            "Office",
            "Sundry",
        ]
    )
    product_id = np.random.randint(1
```

```
        quantity = np.random.randint(1,
        price = np.random.randint(199, 1
        # Generate random dates between
        start_date = datetime(2010, 1, 1
        end_date = datetime(2023, 12, 31
        date_range = (end_date - start_d
        # Create random dates as np.arra
        order_dates = np.array([(start_d
        # Define columns
        columns = {
            "order_id": np.arange(nrows)
            "order_date": order_dates,
            "customer_id": np.random.ran
            "customer_name": [f"Customer
            "product_id": product_id + 2
            "product_names": names[produ
            "categories": categories[pro
            "quantity": quantity,
            "price": price,
            "total": price * quantity,
        }
        # Create Polars DataFrame and wr
        df = pl.DataFrame(columns)
        df.write_csv(filename, separator

# Generate 100,000 rows of data with
generate(100_000, "/mnt/d/sales_data
```

Installing and using Gradio

Installing Gradio is easy using **pip**, but for coding, the best practice is to set up a separate Python environment for all your work. I use

Miniconda for that purpose, but feel free to use whatever method suits your work practice.

> *If you want to go down the conda route and don't already have it, you must install Miniconda (recommended) or Anaconda first.*
>
> *Please note that, at the time of writing,* Gradio needs at least Python 3.8 installed to work correctly.

Once the environment is created, switch to it using the **'activate'** command, and then run **'pip install'** to install our required Python libraries.

```
#create our test environment
(base) C:\Users\thoma>conda create -

# Now activate it
(base) C:\Users\thoma>conda activate

# Install python libraries, etc ...
(gradio_dashboard) C:\Users\thoma>pi
```

Key differences between Streamlit and Gradio

As I'll demonstrate in this article, it's possible to produce very similar data dashboards using Streamlit and Gradio. However, their ethos differs in several key ways.

Focus

- Gradio specialises in creating interfaces for machine learning models, whilst Streamlit is more designed for general-purpose data applications and visualisations.

Ease of use

- Gradio is known for its simplicity and rapid prototyping capabilities, making it easier for beginners to use. Streamlit offers more advanced features and customisation options, which may require a steeper learning curve.

Interactivity

- Streamlit uses a reactive Programming model where any input change triggers a complete script rerun, updating all components immediately. Gradio, by default, updates only when a user clicks a submit button, though it can be configured for live updates.

Customization

- Gradio focuses on pre-built components for quickly demonstrating AI models. Streamlit provides more extensive customisation options and flexibility for complex projects.

Deployment

- Having deployed both a Streamlit and a Gradio app, I would say it’s easier to deploy a Streamlit app than a Gradio app. In Streamlit, deployment

can be done with a single click via the Streamlit Community Cloud. This functionality is built into any Streamlit app you create. Gradio offers deployment using Hugging Face Spaces, but it involves more work. Neither method is particularly complex, though.

Use cases

Streamlit excels in creating data-centric applications and interactive dashboards for complex projects. Gradio is ideal for quickly showcasing machine learning models and building simpler applications.

The Gradio Dashboard Code

I'll break down the code into sections and explain each one as we proceed.

We begin by importing the required external libraries and loading the

full dataset from the CSV file into a Pandas DataFrame.

```
import gradio as gr
import pandas as pd
import matplotlib.pyplot as plt
import datetime
import warnings
import os
import tempfile
from cachetools import cached, TTLCa

warnings.filterwarnings("ignore", ca

# ---------------------------------
# 1) Load CSV data once
# ---------------------------------
csv_data = None

def load_csv_data():
    global csv_data

    # Optional: specify column dtype
    dtype_dict = {
        "order_id": "Int64",
        "customer_id": "Int64",
        "product_id": "Int64",
        "quantity": "Int64",
        "price": "float",
        "total": "float",
        "customer_name": "string",
        "product_names": "string",
        "categories": "string"
    }

    csv_data = pd.read_csv(
```

```
            "d:/sales_data/sales_data.cs
            parse_dates=["order_date"],
            dayfirst=True,       # if you
            low_memory=False,
            dtype=dtype_dict
        )

    load_csv_data()
```

Next, we configure a time-to-live cache with a maximum of 128 items and an expiration of 300 seconds. This is used to store the results of expensive function calls and speed up repeated lookups

The **get_unique_categories** function returns a list of unique, cleaned (capitalised) categories from the `csv_data` DataFrame, caching the result for quicker access.

The **get_date_range** function returns the minimum and maximum order dates from the dataset, or None if the data is unavailable.

The **filter_data** function filters the csv_data DataFrame based on a specified date range and optional

category, returning the filtered DataFrame.

The **get_dashboard_stats** function retrieves summary metrics — total revenue, total orders, average order value, and top category — for the given filters. Internally it uses `filter_data()` to scope the dataset and then calculate these key statistics.

The **get_data_for_table f**unction returns a detailed DataFrame of filtered sales data, sorted by **order_id** and **order_date**, including additional revenue for each sale.

The **get_plot_data** function formats data for generating a plot by summing revenue over time, grouped by date.

The **get_revenue_by_category** function aggregates and returns revenue by category, sorted by revenue, within the specified date range and category.

The **get_top_products** function returns the top 10 products by revenue, filtered by date range and category.

Based on the orientation argument, the **create_matplotlib_figure** function generates a bar plot from the data and saves it as an image file, either vertical or horizontal.

```
cache = TTLCache(maxsize=128, ttl=30

@cached(cache)
def get_unique_categories():
    global csv_data
    if csv_data is None:
        return []
    cats = sorted(csv_data['categori
    cats = [cat.capitalize() for cat
    return cats

def get_date_range():
    global csv_data
    if csv_data is None or csv_data.
        return None, None
    return csv_data['order_date'].mi

def filter_data(start_date, end_date
    global csv_data

    if isinstance(start_date, str):
        start_date = datetime.dateti
    if isinstance(end_date, str):
```

```
        end_date = datetime.datetime

    df = csv_data.loc[
        (csv_data['order_date'] >= p
        (csv_data['order_date'] <= p
    ].copy()

    if category != "All Categories":
        df = df.loc[df['categories']

    return df

def get_dashboard_stats(start_date,
    df = filter_data(start_date, enc
    if df.empty:
        return (0, 0, 0, "N/A")

    df['revenue'] = df['price'] * df
    total_revenue = df['revenue'].su
    total_orders = df['order_id'].nu
    avg_order_value = total_revenue

    cat_revenues = df.groupby('categ
    top_category = cat_revenues.inde

    return (total_revenue, total_orc

def get_data_for_table(start_date, e
    df = filter_data(start_date, enc
    if df.empty:
        return pd.DataFrame()

    df = df.sort_values(by=["order_i

    columns_order = [
        "order_id", "order_date", "c
        "product_id", "product_names
```

```
        "price", "total"
    ]
    columns_order = [col for col in
    df = df[columns_order].copy()

    df['revenue'] = df['price'] * df
    return df

def get_plot_data(start_date, end_da
    df = filter_data(start_date, enc
    if df.empty:
        return pd.DataFrame()
    df['revenue'] = df['price'] * df
    plot_data = df.groupby(df['order
    plot_data.rename(columns={'order
    return plot_data

def get_revenue_by_category(start_da
    df = filter_data(start_date, enc
    if df.empty:
        return pd.DataFrame()
    df['revenue'] = df['price'] * df
    cat_data = df.groupby('categorie
    cat_data = cat_data.sort_values(
    return cat_data

def get_top_products(start_date, enc
    df = filter_data(start_date, enc
    if df.empty:
        return pd.DataFrame()
    df['revenue'] = df['price'] * df
    prod_data = df.groupby('product_
    prod_data = prod_data.sort_value
    return prod_data

def create_matplotlib_figure(data, x
    plt.figure(figsize=(10, 6))
```

```
    if data.empty:
        plt.text(0.5, 0.5, 'No data
    else:
        if orientation == 'v':
            plt.bar(data[x_col], dat
            plt.xticks(rotation=45,
        else:
            plt.barh(data[x_col], da
            plt.gca().invert_yaxis()

    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.tight_layout()

    with tempfile.NamedTemporaryFile
        plt.savefig(tmpfile.name)
    plt.close()
    return tmpfile.name
```

The **update_dashboard** function retrieves key sales statistics (total revenue, total orders, average order value, and top category) by calling the `get_dashboard_stats` function. It gathers data for three distinct visualisations (revenue over time, revenue by category, and top products), then uses `create_matplotlib_figure` to generate plots. It prepares and returns a data table (via the

`get_data_for_table()` function) along with all generated plots and stats so they can be displayed in the dashboard.

The **create_dashboard** function sets the date boundaries (minimum and maximum dates) and establishes the initial default filter values. It uses Gradio to construct a user interface (UI) featuring date pickers, category drop-downs, key metric displays, plot tabs, and a data table. It then wires up the filters so that changing any of them triggers a call to the **update_dashboard** function, ensuring the dashboard visuals and metrics are always in sync with the selected filters. Finally, it returns the assembled Gradio interface launched as a web application.

```
def update_dashboard(start_date, enc
    total_revenue, total_orders, avg

    # Generate plots
    revenue_data = get_plot_data(sta
    category_data = get_revenue_by_c
    top_products_data = get_top_proc
```

```
        revenue_over_time_path = create_
            revenue_data, 'date', 'rever
            "Revenue Over Time", "Date",
        )
        revenue_by_category_path = creat
            category_data, 'categories',
            "Revenue by Category", "Cate
        )
        top_products_path = create_matpl
            top_products_data, 'product_
            "Top Products", "Revenue", '
        )

        # Data table
        table_data = get_data_for_table(

        return (
            revenue_over_time_path,
            revenue_by_category_path,
            top_products_path,
            table_data,
            total_revenue,
            total_orders,
            avg_order_value,
            top_category
        )

    def create_dashboard():
        min_date, max_date = get_date_ra
        if min_date is None or max_date
            min_date = datetime.datetime
            max_date = datetime.datetime

        default_start_date = min_date
        default_end_date = max_date

        with gr.Blocks(css="""
```

```
        footer {display: none !impor
        .tabs {border: none !importa
        .gr-plot {border: none !impo
    """) as dashboard:

        gr.Markdown("# Sales Perform

        # Filters row
        with gr.Row():
            start_date = gr.DateTime
                label="Start Date",
                value=default_start_
                include_time=False,
                type="datetime"
            )
            end_date = gr.DateTime(
                label="End Date",
                value=default_end_da
                include_time=False,
                type="datetime"
            )
            category_filter = gr.Dro
                choices=["All Catego
                label="Category",
                value="All Categorie
            )

        gr.Markdown("# Key Metrics")

        # Stats row
        with gr.Row():
            total_revenue = gr.Numbe
            total_orders = gr.Number
            avg_order_value = gr.Num
            top_category = gr.Textbo

        gr.Markdown("# Visualisation
```

```
# Tabs for Plots
with gr.Tabs():
    with gr.Tab("Revenue Ove
        revenue_over_time_im
    with gr.Tab("Revenue by
        revenue_by_category_
    with gr.Tab("Top Product
        top_products_image =

gr.Markdown("# Raw Data")
# Data Table (below the plot
data_table = gr.DataFrame(
    label="Sales Data",
    type="pandas",
    interactive=False
)

# When filters change, updat
for f in [start_date, end_da
    f.change(
        fn=lambda s, e, c: u
        inputs=[start_date,
        outputs=[
            revenue_over_tim
            revenue_by_categ
            top_products_ima
            data_table,
            total_revenue,
            total_orders,
            avg_order_value,
            top_category
        ]
    )

# Initial load
dashboard.load(
    fn=lambda: update_dashbo
```

```
            outputs=[
                revenue_over_time_im
                revenue_by_category_
                top_products_image,
                data_table,
                total_revenue,
                total_orders,
                avg_order_value,
                top_category
            ]
        )

    return dashboard

if __name__ == "__main__":
    dashboard = create_dashboard()
    dashboard.launch(share=False)
```

Running the program

Create a Python file, e.g. gradio_test.py, and insert all the above code snippets. Save it, and run it like this,

```
(gradio_dashboard) $ python gradio_t

* Running on local URL:  http://127.

To create a public link, set `share=
```

Click on the local URL shown, and the dashboard will open full screen

in your browser.

Summary

This article provides a comprehensive guide to building an interactive sales performance dashboard using Gradio and a CSV file as its source data.

Gradio is a modern, Python-based open-source framework that simplifies the creation of data-driven dashboards and GUI applications. The dashboard I developed allows users to filter data by date ranges and product categories, view key metrics such as total revenue and top-performing categories, explore visualisations like revenue trends and top products, and navigate through raw data with pagination.

I also mentioned some key differences between developing visualisation tools using Gradio and Streamlit, another popular front-

end Python library.

This guide provides a comprehensive implementation of a Gradio data dashboard, covering the entire process from creating sample data to developing Python functions for querying data, generating plots, and handling user input. This step-by-step approach demonstrates how to leverage Gradio's capabilities to create user-friendly and dynamic dashboards, making it ideal for data engineers and scientists who want to build interactive data applications.

Although I used a CSV file for my data, modifying the code to use another data source, such as a relational database management system (RDBMS) like SQLite, should be straightforward. For example, in my other article in this series on creating a similar dashboard using Streamlit, the data source is a PostgreSQL database.

WRITTEN BY

Thomas Reid

See all from Thomas Reid

Topics:

Data Visualization | Deep Dives | Gradio | Programming | Python

Share this article:

Related Articles

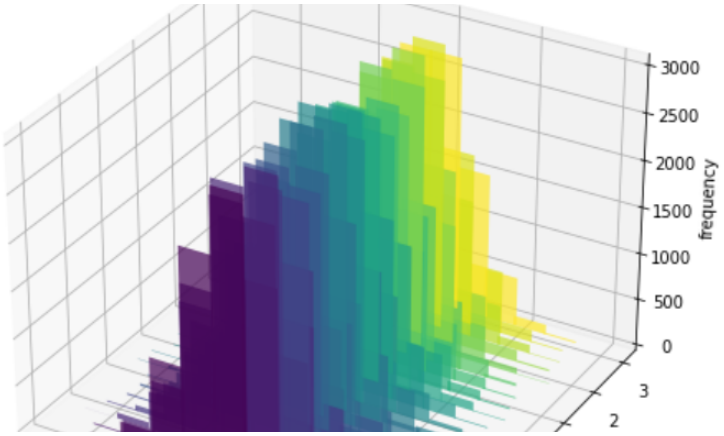
DATA SCIENCE

Solving a Constrained Project Scheduling Problem with Quantum Annealing

Solving the resource constrained project scheduling problem (RCPSP) with D-Wave's hybrid constrained quadratic model (CQM)

Luis Fernando PÉREZ ARMAS, Ph.D.

August 20, 2024 29 min read



DATA SCIENCE

Must-Know in Statistics: The Bivariate Normal Projection Explained

Derivation and practical examples of this powerful concept

Luigi Battistoni

August 14, 2024 7 min read

DATA SCIENCE

Towards Generalization on Graphs: From Invariance to Causality

This blog post shares recent papers on out-of-distribution generalization on graph-

structured data

Qitian Wu

July 18, 2024 19 min read

CINEMA

Evaluating Cinematic Dialogue - Which syntactic and semantic features are predictive of genre?

This article explores the relationship between a movie's dialogue and its genre, leveraging domain-driven data...

Christabelle Pabalan

January 20, 2024 18 min read

DATA SCIENCE

Squashing the Average: A Dive into Penalized Quantile Regression for Python

How to build penalized quantile regression

models (with code!)

Álvaro Méndez Civieta

August 16, 2024 5 min read

DATA SCIENCE

The Math Behind Keras 3 Optimizers: Deep Understanding and Application

This is a bit different from what the books say.

Peng Qian

August 17, 2024 9 min read

DATA SCIENCE

Quantization, Linear Regression, and Hardware for AI: Our Best Recent Deep Dives

Our weekly selection of must-read Editors' Picks and original features

TDS Editors

April 18, 2024 3 min read

April 15, 2024 · 5 min read

Your home for data science and AI. The world's leading publication for data science, data analytics, data engineering, machine learning, and artificial intelligence professionals.

Subscribe to Our Newsletter