Python and SQL: intro / SQL platforms

Ewa Weychert

Class 11: Django Data Explorer

Learning goals (today)

By the end of 90 minutes, students will be able to:

- Explain how Streamlit UI concepts map to Django (server-rendered) web apps
- Build a Django app that:
  - uploads a CSV,
  - shows overview + missingness + types,
  - renders plots for numeric/categorical columns
- Understand key web concepts: HTTP, forms, templates, sessions, static/media

Agenda (90 minutes)

1. (0–10) Why Django instead of Streamlit (architecture + tradeoffs)
2. (10–25) Project setup + URL routing + templates
3. (25–40) File upload flow (POST + CSV parsing)
4. (40–55) Data panels: overview + missingness + dtypes + unique counts
5. (55–70) Tabs + controls (Bootstrap + form inputs)
6. (70–85) Charts (Matplotlib → base64 images in HTML)
7. (85–90) Wrap-up + next steps

Recap: What the Streamlit app does

Streamlit features in the given code:

- **File uploader** for CSV
- **Sidebar sliders**: head rows, histogram bins, top-N categories
- **Tabs**: Overview, Missing & Types, Continuous, Categorical
- **Tables**: head(), describe(), dtypes, missingness
- **Plots**: histograms + bar charts (Matplotlib)

Mapping Streamlit → Django

Streamlit	**Django approach**
`st.file_uploader`	HTML form `<input type=file>` + POST
`st.tabs([...])`	Bootstrap nav-tabs + tab panes in template
`st.dataframe(df)`	`df.to_html()` rendered in template
`st.sidebar.slider`	Form inputs (GET/POST) + defaults
`st.pyplot(fig)`	Save plot to PNG + embed (base64)
Streamlit session state	Django session / hidden form fields / temp files

Django architecture (MVT)

- **Model**: database tables (optional for this demo)
- **View**: Python code handling request → response
- **Template**: HTML presentation

Key shift from Streamlit:

- UI is **HTML/CSS/JS**, not Python widgets
- Each interaction is an **HTTP request** (GET/POST)

Setup: create project and app

```
python -m venv venv
# activate venv (platform-specific)
pip install django pandas matplotlib

django-admin startproject dataexplorer
cd dataexplorer
python manage.py startapp explorer
```

Add app to `INSTALLED_APPS` in `settings.py`.

URLs: route requests to views

Create `explorer/urls.py` and include it in the project.

```
# dataexplorer/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
  path("admin/", admin.site.urls),
  path("", include("explorer.urls")),
]
```

```
# explorer/urls.py
from django.urls import path
from . import views

urlpatterns = [
  path("", views.upload_and_explore, name="upload"),
]
```

Templates: server-rendered HTML

We will use templates for:

- Upload form
- Tabs layout
- Tables (HTML)
- Images (base64)

Create folder structure:

```
explorer/
  templates/
    explorer/
      page.html
```

Design choice: one page with tabs

To match Streamlit:

- One endpoint `/` handling GET (blank state) + POST (uploaded CSV)
- After upload, render all tab contents
- Controls (bins, top_n, n_head) come from form inputs

Why?

- Simple for teaching
- Mirrors Streamlit "single script" feel

HTML: upload form + controls

```
<form method="post" enctype="multipart/form-data">
  {% csrf_token %}

  <input type="file" name="csv_file" accept=".csv"
     required>

  <label>Rows in preview</label>
  <input type="number" name="n_head" value="10" min="1
     " max="200">

  <label>Histogram bins</label>
  <input type="number" name="bins" value="20" min="5"
     max="50">

  <label>Top N categories</label>
  <input type="number" name="top_n" value="10" min="3"
      max="30">

  <button type="submit">Explore</button>
```

Security note: CSRF protection

Django requires a CSRF token for POST forms:

- Prevents cross-site request forgery
- Use `{% csrf_token %}` inside every POST form

Common student bug: “403 CSRF verification failed”.

View: request handling (GET vs POST)

```python
# explorer/views.py
import pandas as pd
from django.shortcuts import render

def upload_and_explore(request):
    if request.method == "POST" and request.FILES.get(
        "csv_file"):
        csv_file = request.FILES["csv_file"]
        df = pd.read_csv(csv_file)

        # read controls with defaults
        n_head = int(request.POST.get("n_head", 10))
        bins = int(request.POST.get("bins", 20))
        top_n = int(request.POST.get("top_n", 10))

        context = build_context(df, n_head, bins,
            top_n)
        return render(request, "explorer/page.html",
            context)
```

Compute panels: `build_context(df, ...)`

We package all panel outputs into a context dict:

- Tables as HTML strings
- Plot images as base64 strings
- Metadata: shape, columns list, numeric/categorical column lists

This keeps `upload_and_explore` small and readable.

Overview panel (matches Streamlit tab)

```
def overview_context(df, n_head):
    num_df = df.select_dtypes(include="number")
    cat_df = df.select_dtypes(exclude="number")

    return {
      "shape": df.shape,
      "columns": list(df.columns),
      "head_table": df.head(n_head).to_html(classes="
          table table-sm"),
      "num_desc": (
        num_df.describe().to_html(classes="table table
            -sm")
        if not num_df.empty else None
      ),
      "cat_desc": (
        cat_df.describe(include="all").to_html(classes
            ="table table-sm")
        if not cat_df.empty else None
      )
```

Missingness & types panel

```
def missing_types_context(df):
    miss_count = df.isna().sum().to_frame("
        missing_count")
    miss_pct = (df.isna().mean() * 100).to_frame("
        missing_%")
    dtypes = df.dtypes.to_frame("dtype")
    nunique = df.nunique().to_frame("n_unique")

    return {
      "miss_count": miss_count.to_html(classes="table
          table-sm"),
      "miss_pct": miss_pct.round(2).to_html(classes="
          table table-sm"),
      "dtypes": dtypes.to_html(classes="table table-sm
          "),
      "nunique": nunique.to_html(classes="table table-
          sm"),
    }
```

Plotting in Django: Matplotlib → base64

In Streamlit: `st.pyplot(fig)` shows the plot directly.
In Django: we must **serialize the plot** and embed it as an image.

```
import io, base64
import matplotlib.pyplot as plt

def fig_to_base64_png(fig):
    buf = io.BytesIO()
    fig.tight_layout()
    fig.savefig(buf, format="png", dpi=150)
    plt.close(fig)
    buf.seek(0)
    return base64.b64encode(buf.read()).decode("utf-8"
        )
```

Continuous plots: histograms for all numeric columns

```
def continuous_plots(df, bins):
    images = []
    for col in df.select_dtypes(include="number").
        columns:
        data = df[col].dropna()
        if data.empty:
            continue
        fig, ax = plt.subplots()
        ax.hist(data, bins=bins)
        ax.set_title(col)
        ax.set_xlabel(col)
        ax.set_ylabel("Count")
        images.append({"name": col, "png":
            fig_to_base64_png(fig)})
    return images
```

Categorical plots: top-N bar charts

```
def categorical_plots(df, top_n):
    images = []
    for col in df.select_dtypes(exclude="number").
        columns:
        vc = df[col].value_counts().head(top_n)
        if vc.empty:
            continue
        fig, ax = plt.subplots()
        ax.bar(vc.index.astype(str), vc.values)
        ax.set_title(col)
        ax.set_xlabel(col)
        ax.set_ylabel("Count")
        ax.tick_params(axis="x", rotation=45)
        images.append({"name": col, "png":
            fig_to_base64_png(fig)})
    return images
```

Putting it together: build_context

```
def build_context(df, n_head, bins, top_n):
    ctx = {"empty": False}
    ctx.update(overview_context(df, n_head))
    ctx.update(missing_types_context(df))
    ctx["continuous_imgs"] = continuous_plots(df, bins
        )
    ctx["categorical_imgs"] = categorical_plots(df,
        top_n)
    ctx["controls"] = {"n_head": n_head, "bins": bins,
         "top_n": top_n}
    return ctx
```

Tabs in HTML (Bootstrap idea)

We replicate Streamlit tabs with Bootstrap:

- A nav bar (tab buttons)
- Tab panes (divs) for each section

Students learn a real-world pattern: **framework + frontend library**.

Template: tab skeleton (page.html)

```
<ul class="nav nav-tabs" role="tablist">
  <li class="nav-item">
    <a class="nav-link active" data-bs-toggle="tab"
       href="#overview">Overview</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" data-bs-toggle="tab" href="#
       missing">Missing & Types</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" data-bs-toggle="tab" href="#
       cont">Continuous</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" data-bs-toggle="tab" href="#
       cat">Categorical</a>
  </li>
</ul>
```

4-section layout (like Streamlit columns)

We mimic the 2x2 panel layout using a grid:

- Bootstrap rows + columns
- Each section shows one table

```
<div class="row">
  <div class="col-md-6">Section 1</div>
  <div class="col-md-6">Section 2</div>
</div>
<div class="row mt-3">
  <div class="col-md-6">Section 3</div>
  <div class="col-md-6">Section 4</div>
</div>
```

Rendering DataFrames in templates

We pass `df.to_html()` strings. In templates:

- Use `{{ table_html|safe }}` to render as HTML
- Explain: "safe" disables escaping for that string

```
<h5>Preview</h5>
{{ head_table|safe }}

{% if num_desc %}
  <h5>Numeric describe()</h5>
  {{ num_desc|safe }}
{% else %}
  <p><em>No numeric columns found.</em></p>
{% endif %}
```

Rendering plots (base64 images)

```
{% for img in continuous_imgs %}
  <div class="col-md-6 mb-3">
    <h6>{{ img.name }}</h6>
    <img class="img-fluid"
         src="{{ img.png }}"
         alt="Histogram {{ img.name }}">
  </div>
{% endfor %}
```

Works without saving files to disk (nice for a classroom demo).

Robustness: common edge cases

Discuss and handle:

- Empty CSV / parsing errors
- Huge CSV (performance + memory)
- Columns with all missing values
- Non-UTF8 encoding

Minimal classroom handling:

- `try/except` around `pd.read_csv`
- Show friendly error message in template

Performance note (practical web dev)

Streamlit runs in one Python process with state. Django:

- Handles many users → state must be explicit
- Avoid storing large DataFrames in session

Simple strategies:

- Recompute from uploaded file (demo)
- Save upload to temp file and store filename (advanced)
- Cache computed outputs (advanced)

Live build checkpoints

Stop and verify after each:

1. Server runs: `python manage.py runserver`
2. Upload form appears (GET)
3. Upload works (POST) and preview table shows
4. Missingness panel shows 4 tables
5. Tabs switch correctly
6. At least one histogram renders
7. At least one categorical bar chart renders

Exercise (5 minutes): add a “Download cleaned CSV” button

Goal: teach HTTP responses.

- Create cleaned df: `df.dropna()`
- Return `HttpResponse` with CSV content and headers

Hint:

```
from django.http import HttpResponse

resp = HttpResponse(clean_df.to_csv(index=False),
                    content_type="text/csv")
resp["Content-Disposition"] = 'attachment; filename="
    cleaned.csv"'
return resp
```

Exercise (5 minutes): limit categories in bar charts

Add an “Other” bucket:

- Keep top_n
- Sum remaining counts into “Other”

This teaches data transformation before visualization.

Testing mindset (quick)

Even for demos, add one or two checks:

- “Does the view return 200 on GET?”
- “Does POST with a small CSV return tables?”

(If time allows) show Django test client briefly.

Wrap-up: what students should remember

- Streamlit is Python-first UI; Django is web-first (HTML + requests)
- The **data logic is reusable** (pandas + matplotlib)
- The big difference is **how UI is built and state is handled**

Next steps:

- Add authentication
- Save uploads to `MEDIA_ROOT`
- Add pagination / sampling for large data
- Convert to API (Django REST Framework) + frontend

Reference: minimal file list

- `dataexplorer/settings.py` (add app, templates)
- `dataexplorer/urls.py` (include explorer URLs)
- `explorer/urls.py`
- `explorer/views.py` (upload + build_context)
- `explorer/templates/explorer/page.html`

Q&A