The TEST

1.Explain the key features of Streamlit that make it suitable for data science and machine learning applications.

Ans: Key Features of Streamlit for Data Science & Machine Learning Applications:

i.Easy to Use & No Frontend Required

Streamlit allows you to build web apps using only Python—no need for HTML, CSS, or JavaScript.

Simple syntax makes it accessible for beginners and quick prototyping.

ii.Fast Development & Deployment

Convert a Python script into an interactive web app with minimal code.

Apps can be deployed online using Streamlit Cloud, Hugging Face Spaces, or Heroku.

iii.Built-in Support for Data Science Libraries

Direct integration with Pandas, NumPy, Matplotlib, Seaborn, Plotly, Altair, and Bokeh for data visualization.

Supports Scikit-learn, TensorFlow, and PyTorch for machine learning models.

iv.Automatic UI Generation

Streamlit automatically creates a user-friendly interface for your app based on Python functions.

No need to design front-end elements manually.

v.Real-Time Data Updates

Supports live updates for streaming data and real-time visualizations.

Great for financial dashboards, stock market apps, and sensor monitoring.

vi.Machine Learning Model Deployment

Easily load and run ML models from Scikit-learn, TensorFlow, or Hugging Face Transformers.

Allows integration with APIs for real-world AI applications.

vii.Supports Data Processing & Analysis

Handles large datasets efficiently with built-in caching for faster performance.

Supports real-time data manipulation using Pandas and SQL databases.

viii.Open-Source & Free to Use

Streamlit is completely open-source and has an active community for support.

Regular updates add new features and improve performance.

2.How does Streamlit handle state management, and what are some ways to persist data across interactions?

Ans: Streamlit is stateless by default, meaning that each time a user interacts with a widget (button, slider, text input, etc.), the script reruns from top to bottom. This makes building apps simple but presents a challenge for persisting data between interactions.

Streamlit's State Management

1. Session State
Introduction: Streamlit introduced st.session_state to manage state across reruns. This allows you to store and persist user inputs or variables throughout the app's lifecycle.

Usage: You can use st.session_state to retain variables, form inputs, counters, or UI elements across user interactions, preventing data loss due to automatic reruns.

2. Automatic Reruns
Reactivity: Streamlit automatically reruns the script from top to bottom whenever a user interacts with a widget. Without state management, this can lead to loss of previous values.
State Preservation: Using st.session_state ensures that user data and other application states persist across reruns, maintaining a smooth user experience.

Strategies to Persist Data Across Interactions

1. Using st.session_state for Persistent Variables
Streamlit's st.session_state allows storing variables across interactions, avoiding loss of state when the app refreshes.

```
import streamlit as st


if "counter" not in st.session_state:
    st.session_state.counter = 0

if st.button("Increase Counter"):
    st.session_state.counter += 1

st.write(f"Counter: {st.session_state.counter}")
```

Explanation: This script initializes a counter that increments when the button is clicked. The value persists across reruns, ensuring the counter does not reset.

2. Caching with @st.cache_data for Data Caching
Streamlit provides @st.cache_data to store frequently used data, reducing reload time and optimizing performance.

```
import streamlit as st
import pandas as pd

@st.cache_data
def load_data():
```

```
    return pd.read_csv("data.csv")

df = load_data()
st.dataframe(df)
```
Explanation: The function loads a CSV file only once, and its results are cached. This prevents reloading the file every time the app reruns, improving efficiency.

3. File Uploads and Downloads for Local Persistence

Streamlit allows users to upload files, process them, and download results without losing data.

```
import streamlit as st
import pandas as pd

uploaded_file = st.file_uploader("Upload a CSV file", type=["csv"])
if uploaded_file:
    df = pd.read_csv(uploaded_file)
    st.write(df)

    st.download_button("Download Processed Data", df.to_csv(index=False),
"processed_data.csv", "text/csv")
```
Explanation: Users can upload a CSV file, process it, and download the modified version without losing data across reruns.

4. Using External Storage (Databases) for Long-Term Persistence

For multi-user applications, storing data in a database ensures persistence even after app restarts.

```
import sqlite3
import streamlit as st

 Connect to SQLite Database
conn = sqlite3.connect("users.db")
cursor = conn.cursor()

Create Table
cursor.execute("CREATE TABLE IF NOT EXISTS users (name TEXT)")
conn.commit()

User Input
name = st.text_input("Enter Name:")
```

```
if st.button("Save to Database"):
    cursor.execute("INSERT INTO users (name) VALUES (?)", (name,))
    conn.commit()
    st.success("Name saved!")

 Fetch and Display Data
cursor.execute("SELECT * FROM users")
names = cursor.fetchall()
st.write("Saved Users:", names)
```

Explanation: A SQLite database is used to store user inputs permanently.
The name entered by the user is saved to the database and displayed in the app.
Even after the app restarts, the saved names remain in the database.
Conclusion:
Streamlit's state management capabilities, particularly st.session_state, provide an effective way to handle user interactions and persist data across reruns.

3.Compare Streamlit with Flask and Django. In what scenarios would you prefer Streamlit over these traditional web frameworks?

Ans : Streamlit vs Flask vs Django: Key Differences

1. Development Speed & Complexity

Streamlit → Fastest to develop. No need for HTML, CSS, or JavaScript. Just write Python code, and it auto-generates the UI. Best for quick prototypes.
Flask → Moderate complexity. Requires manual routing and HTML templates but offers flexibility for lightweight web apps.
Django → Most complex. Uses the Model-View-Controller (MVC) architecture, which is great for full-stack applications but has a steep learning curve.

2. Use Cases

Streamlit → Best for data science dashboards, ML model deployment, quick prototypes, and interactive visualizations.
Flask → Ideal for REST APIs, small web applications, and lightweight backends. Used when you need to build a custom UI.
Django → Best for large-scale, enterprise applications like e-commerce sites, authentication systems, and full-stack web platforms.

3. Frontend Requirements

Streamlit → No frontend needed. It auto-generates UI using Python.
Flask → Requires HTML, CSS, JavaScript (or a frontend framework like React).
Django → Uses Django Templates but also supports frontend frameworks like React or Vue.js.

4. Performance & Scalability

Streamlit → Optimized for data apps but not ideal for high-traffic applications.
Flask → Faster than Django because it's lightweight but requires manual optimization.

Django → Best scalability for large applications due to built-in database management and security features.
5. Database & Backend Features
Streamlit → Limited database support. You must use external storage solutions like Firebase, PostgreSQL, or SQLite.
Flask → Supports SQLAlchemy ORM, but database handling is manual.
Django → Has built-in ORM with support for PostgreSQL, MySQL, and SQLite. Best for complex database-driven apps.
6. Deployment & Hosting
Streamlit → Easiest to deploy (Supports Streamlit Cloud, Hugging Face Spaces, AWS, and Heroku).
Flask → Moderate complexity (Can be deployed on Heroku, AWS, or GCP but requires setup).
Django → Most complex (Often deployed using Docker, Kubernetes, AWS Elastic Beanstalk).
When to Choose Streamlit Over Flask or Django?
Use Streamlit When:
You need a quick, interactive data app for machine learning models, dashboards, or visualizations.
You want a simple Python-based UI without frontend development.
You're building an internal tool or proof-of-concept with limited users.

Don't Use Streamlit If:
You need a highly scalable, multi-user web app (Flask/Django are better).
You require complex database relationships and authentication (Django is better).
You want to build a REST API (Flask is better).

4.Describe the role of caching (@st.cache_data and @st.cache_resource) in Streamlit. How does it improve performance?
Ans : Role of Caching in Streamlit (@st.cache_data & @st.cache_resource)
Caching in Streamlit improves performance by storing expensive computations so they don't rerun unnecessarily, reducing execution time and server load.

1. @st.cache_data → Used for caching function outputs like datasets, API responses, and expensive calculations. It prevents reloading data unless inputs change.
2. @st.cache_resource → Used for caching objects like ML models, database connections, or API sessions, ensuring they persist across reruns.

How It Improves Performance

Reduces recomputation → Cached results are retrieved instantly.
Enhances app speed → Improves responsiveness for large datasets or ML models.
Optimizes resource usage → Minimizes repeated processing, reducing CPU and memory load.

Key Difference: @st.cache_data is for function results, while @st.cache_resource is for persistent objects. Both help streamline Streamlit apps by making them faster and more efficient.
5.How can you integrate a database with a Streamlit app? Provide an example using SQLite or PostgreSQL.

Ans : Integrating a Database with a Streamlit App
Streamlit can connect to databases like SQLite, PostgreSQL, MySQL, and Firebase to store and retrieve data dynamically. This allows apps to persist user inputs, fetch real-time data, and manage structured information efficiently.

Steps to Integrate a Database in Streamlit
1 Install required libraries → pip install sqlite3 psycopg2 pandas (for PostgreSQL).
2 Connect to the database → Establish a connection using SQLite or PostgreSQL.
3 Create a table → Define a structure for storing data.
4 Insert & retrieve data → Save user input and display stored data.

Example 1: Using SQLite (Lightweight Database for Local Storage)

```
import streamlit as st
import sqlite3

# Connect to SQLite database
conn = sqlite3.connect("users.db")
cursor = conn.cursor()

# Create table if not exists
cursor.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)")
conn.commit()

# Input fields for user data
name = st.text_input("Enter your name:")
age = st.number_input("Enter your age:", min_value=1, step=1)
```

```
# Save to database
if st.button("Save Data"):
    cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", (name, age))
    conn.commit()
    st.success("User data saved!")

# Display stored data
cursor.execute("SELECT * FROM users")
users = cursor.fetchall()
st.write("### Stored Users:")
st.table(users)
```

Best for small-scale, local applications.
No need for an external database server.

6.Discuss how you can deploy a Streamlit application. Mention at least two deployment platforms.

Ans : Deploying a Streamlit App

You can deploy Streamlit apps online using various platforms. Here are two popular options:

1. Streamlit Community Cloud (Free & Easy)
Best For: Quick deployment for personal projects.
Steps:

Push your code to GitHub (include requirements.txt).
Go to Streamlit Cloud → Click "Deploy".
Select your GitHub repo & script (app.py), then deploy.
App is ready

2 Heroku (For More Control & Databases)
Best For: Apps needing custom dependencies & database support. Steps:

Install Heroku CLI and login:

```
pip install heroku
heroku login
```

Add a Procfile:
less

```
web: streamlit run app.py --server.port=$PORT --server.address=0.0.0.0
```

Deploy:
git init
heroku create my-app
git add . && git commit -m "Deploy"
git push heroku main
heroku open
Other Options
Hugging Face Spaces → Best for ML/AI apps.
AWS/GCP → For enterprise-scale apps.
7.What are some limitations of Streamlit, and how can you overcome them when building production-grade applications?
Ans : Limitations of Streamlit & How to Overcome Them
While Streamlit is great for building quick data apps, it has some limitations that can affect production-grade applications. Here's how to overcome them:

1 Stateless Nature (No Built-in Multi-User Sessions)
Issue: Every user interaction triggers a full script rerun, making multi-user applications difficult.
Solution:

Use st.session_state to store user-specific variables.
Store user data in a database (PostgreSQL, Firebase) to persist across sessions.
2 Limited Backend & API Support
Issue: Streamlit lacks built-in support for REST APIs, authentication, or advanced backend logic.
Solution:

Use Flask or FastAPI as a backend to handle complex logic and connect Streamlit via API calls.
Implement OAuth or JWT authentication for user access control.
3 Performance Issues for Large Datasets
Issue: Streamlit loads everything in memory, which can slow down large data processing.
Solution:

Use @st.cache_data to store loaded data and avoid reloading.
Process data in chunks or use SQL databases instead of loading everything into memory.
4 Limited UI Customization
Issue: Cannot create highly customized dashboards like React, Vue.js, or Dash.

Solution:

Use custom HTML, CSS, and JavaScript components via st.components.v1.
Embed Plotly, Altair, and other visualization tools for better UI.
5 Deployment Constraints
Issue: Streamlit Cloud has limited resources, and Heroku/AWS setup can be complex.
Solution:

For scalable apps, deploy on Heroku, AWS, or GCP instead of Streamlit Cloud.
Use Docker for better environment control.
Conclusion
Streamlit is best for prototypes and internal tools, but for scalable, production-grade apps, combine it with Flask, FastAPI, or cloud databases to enhance performance and functionality.

8. Explain the process of creating an interactive dashboard in Streamlit. What components would you use?
Ans : Creating an Interactive Dashboard in Streamlit
Steps to Build a Dashboard
1 Install Dependencies
pip install streamlit pandas matplotlib plotly
2 Load Data

```
import streamlit as st
import pandas as pd

st.title("Sales Dashboard ")
df = pd.read_csv("sales_data.csv")
st.dataframe(df)
```

3 Add Filters & Widgets

```
category = st.selectbox("Select Category:", df["Category"].unique())
price_range = st.slider("Price Range", min(df["Price"]), max(df["Price"]), (10, 100))
filtered_df = df[(df["Category"] == category) & (df["Price"].between(*price_range))]
st.write(filtered_df)
```

4 Add Visualizations

```
import plotly.express as px
fig = px.bar(filtered_df, x="Product", y="Sales", color="Category")
st.plotly_chart(fig)
```

5 Show KPIs

```
st.metric("Total Sales", f"${filtered_df['Sales'].sum():,.2f}")
st.metric("Avg Price", f"${filtered_df['Price'].mean():,.2f}")
```

Key Components

st.selectbox() → Filter data

st.plotly_chart() → Visualize trends

st.metric() → Display KPIs

Conclusion

Streamlit makes dashboards fast & interactive—no frontend needed!

9.How would you implement user authentication in a Streamlit app? Provide possible solutions.

Ans : User Authentication in Streamlit

Streamlit lacks built-in authentication, but you can implement it using different methods:

1 Basic Authentication with st.session_state

```
import streamlit as st

users = {"admin": "password123", "user": "pass456"}
if "authenticated" not in st.session_state:
    st.session_state.authenticated = False

username = st.text_input("Username:")
password = st.text_input("Password:", type="password")
if st.button("Login") and users.get(username) == password:
    st.session_state.authenticated = True
    st.success("Login successful!")

if st.session_state.authenticated:
    st.write("Welcome!")
```

Simple but insecure (no password hashing).

2 Secure Authentication with streamlit-authenticator

```
pip install streamlit-authenticator
import streamlit_authenticator as stauth

credentials = {"usernames": {"admin": {"name": "Admin", "password":
stauth.Hasher(["password123"]).generate()[0]}}}
```

```
authenticator = stauth.Authenticate(credentials, "app_home", "auth_key",
cookie_expiry_days=1)
name, status, username = authenticator.login("Login", "main")

if status: st.write(f"Welcome, {name}!")
elif status == False: st.error("Invalid credentials")
```
Supports hashed passwords & session management.

3 OAuth (Google/Firebase) for Large Apps
Use Google OAuth or Firebase Authentication for secure login.
Best for multi-user enterprise apps.
Conclusion
Use st.session_state → For quick login.
Use streamlit-authenticator → For secure apps.
Use Firebase/OAuth → For large-scale authentication.

10. Describe a real-world use case where you have implemented or would implement a Streamlit application.

Ans :

Real-World Use Case: Streamlit App for Image Display & Adjustment
A simple web app was created using Streamlit to display a beautiful stadium image, add a caption, and allow users to adjust brightness using a slider. A button was also included for interactivity.

Features of the App
Title – Displays a heading at the top.
Image Upload & Display – Shows the stadium picture.
Caption – Provides context below the image.
Brightness Slider – Adjusts image brightness dynamically.
Button – Can trigger an action, like resetting brightness.

The code :

```
import streamlit as st
from PIL import Image

st.title("My Streamlit App")

image = Image.open("alessio-patron-7uEaIrv2-Wk-unsplash.jpg")

st.image(image, caption="The Greatest football stadium in the world",
use_container_width=True)
value = st.slider("Select a value", 0, 100, 50)
```

```
st.write(f"You selected: {value}")

if st.button("Click Me"):
    st.write("Button clicked!")
```

Why Streamlit Was Used?

- Quick & easy UI setup (No need for frontend coding).
- Interactive elements like sliders & buttons improve user experience.
- Ideal for image processing applications (photo editors, filters, etc.).