1.Display Text

- **Purpose**: When to use it
- **Syntax**: How to use it
- **Example**: Real-world use case

✅ `st.title()`

Purpose:

Display the **main title** of the app. It's the largest text.

Syntax:

```
st.title("Your App Title")
```

Example:

```
st.title("📊 Sales Dashboard")
```

Output:

A big bold title with an emoji if desired.

✅ `st.header()`

Purpose:

Use for **main sections** under the title. Slightly smaller than `title`.

Syntax:

```
st.header("Data Overview")
```

Example:

```
st.header("📈 Monthly Revenue")
```

Output:

Bold header to start a section.

`st.subheader()`

Purpose:

Use for **subsections** within a header. Smaller than header.

Syntax:

```
st.subheader("Revenue in USD")
```

Example:

```
st.subheader("🔹 January")
```

Output:

Italic bold text, suitable for subsection titles.

`st.text()`

Purpose:

Display **plain text** exactly as written (no formatting).

Syntax:

```
st.text("This is plain text")
```

Example:

```
st.text("Note: Data collected from Jan to Dec")
```

Output:

Simple unformatted text.

✅ `st.markdown()`

Purpose:

Display **rich formatted text** using Markdown syntax.

Syntax:

```
st.markdown("# Title\n**bold** *italic* [link](https://example.com)")
```

Example:

```
st.markdown("""
### 🚀 Project Highlights
- **Accuracy**: 95%
- *Model*: Random Forest
- [GitHub Repo](https://github.com/)
""")
```

Output:

Richly formatted text with links, bullets, bold, headings, etc.

✅ `st.caption()`

Purpose:

Show **small descriptive text**, typically under a chart/image.

Syntax:

```
st.caption("Data Source: Kaggle 2024")
```

Example:

```
st.caption("📊 Figure 1: Monthly sales in USD")
```

Output:

Faded small text, used for footnotes or chart legends.

✅ `st.latex()`

Purpose:

Render **mathematical equations** using LaTeX syntax.

Syntax:

```
st.latex(r"\frac{a}{b} = c")
```

Example:

```
st.latex(r"E = mc^2")
```

Output:

Math formula beautifully rendered.

✅ `st.code()`

Purpose:

Display **code snippets** with syntax highlighting.

Syntax:

```
st.code("def hello():\n    print('Hello, world!')", language='python')
```

Example:

```
st.code("""
import pandas as pd

df = pd.read_csv('data.csv')
df.head()
""", language='python')
```

Output:

Code block with formatting (line numbers, colors).

`st.write()`

Purpose:

Universal display function—can show text, dataframes, markdown, even charts.

Syntax:

```
st.write("This is a string")
st.write(1234)
st.write(my_dataframe)
```

Example:

```
st.write("📌 Selected Options:")
st.write(["Option 1", "Option 2"])
```

Output:

Smart rendering based on the object passed. Very flexible.

Summary Table

Function	Best For	Formatting Support
`st.title()`	App Title	Large Header
`st.header()`	Section Titles	Medium Header
`st.subheader()`	Subsection Titles	Smaller Header
`st.text()`	Plain Text	No formatting
`st.markdown()`	Formatted text, links, lists	Markdown syntax
`st.caption()`	Footnotes or chart captions	Small italic text
`st.latex()`	Math equations	LaTeX
`st.code()`	Code display	Syntax Highlighting
`st.write()`	Dynamic display	Auto-detect formatting

Start coding or generate with AI.

2.Display Data

✅ `st.dataframe()`

Purpose:

Display a **scrollable, interactive table** (sortable columns, resizable).

Syntax:

```
st.dataframe(data)
```

Example:

```
import streamlit as st
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Score': [85, 92, 78],
    'Passed': [True, True, False]
})

st.dataframe(df)
```

Sample Output:

	Name	Score	Passed
0	Alice	85	✅
1	Bob	92	✅
2	Charlie	78	❌

(scrollable + column sorting)

✅ `st.table()`

Purpose:

Display a **static, non-scrollable table**. Use when interaction isn't needed.

Syntax:

```
st.table(data)
```

Example:

```
st.table(df)
```

Sample Output (looks similar to `dataframe` but without interactivity):

Name	Score	Passed
Alice	85	True
Bob	92	True
Charlie	78	False

`st.json()`

Purpose:

Display formatted, collapsible **JSON data** (e.g., API responses).

Syntax:

```
st.json(data)
```

Example:

```
person = {
    "name": "Alice",
    "age": 25,
    "skills": ["Python", "ML", "Streamlit"]
}
st.json(person)
```

Sample Output:

```
{
  "name": "Alice",
  "age": 25,
  "skills": [
    "Python",
    "ML",
    "Streamlit"
```

```
  ]
}
```

(collapsible, syntax-highlighted JSON viewer)

✅ `st.metric()`

Purpose:

Show **key performance indicators (KPIs)**—like a dashboard card.

Syntax:

```
st.metric(label, value, delta)
```

Example:

```
st.metric(label="Revenue", value="$120K", delta="+5%")
```

Sample Output:

Revenue 💰 $120K 📈 +5%

(auto-formats the delta and shows up/down arrows)

✅ `st.columns()`

Purpose:

Create **side-by-side layouts** for displaying multiple widgets or elements in a row.

Syntax:

```
col1, col2 = st.columns(2)
with col1:
    st.metric("Sales", "$10K")
with col2:
    st.metric("Growth", "15%", "+2%")
```

Sample Output:

| 📊 Sales: $10K | 📈 Growth: 15% (+2%) |

(Two cards displayed next to each other)

✅ `st.expander()`

Purpose:

Collapse content to **hide/show additional information** like FAQs, logs, or advanced settings.

Syntax:

```
with st.expander("See Details"):
    st.write("Here's more information...")
```

Example:

```
with st.expander("ℹ️ Click to expand"):
    st.markdown("""
    - Detail 1
    - Detail 2
    - Logs, outputs, and notes
    """)
```

Sample Output:

▶️ ℹ️ **Click to expand** *(Clicking expands to show more text)*

✅ Final Notes

Function	Purpose	Interactivity	Ideal Use Case
`st.dataframe`	Interactive table	✅	When users may need to explore/sort data
`st.table`	Static table	❌	Simple display of structured info
`st.json`	JSON viewer	✅	Showing API responses
`st.metric`	KPI or value display	✅	Dashboards
`st.columns`	Side-by-side layout	✅	Custom layouts
`st.expander`	Expandable section	✅	Hide advanced info/FAQs

Start coding or generate with AI.

3.Charts and Graphs

✅ 1. `st.line_chart()`

Purpose: Quick line chart from a DataFrame.

```
import streamlit as st
import pandas as pd
import numpy as np

data = pd.DataFrame(np.random.randn(20, 3), columns=['A', 'B', 'C'])
st.line_chart(data)
```

🟢 **Output**: A line plot with lines for A, B, and C over 20 rows.

✅ 2. `st.area_chart()`

Purpose: Filled area under the line.

```
st.area_chart(data)
```

🟢 **Output**: Like a line chart, but with the area under the curves filled with color.

✅ 3. `st.bar_chart()`

Purpose: Bar chart from a DataFrame or Series.

```
st.bar_chart(data)
```

🟢 **Output**: Vertical bars grouped by index for columns A, B, and C.

✅ 4. `st.pyplot()`

Purpose: Render **Matplotlib** plots.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(data['A'], label='A')
ax.set_title("Matplotlib Line")
```

```
ax.legend()
st.pyplot(fig)
```

🟢 **Output**: Full control over plot styling (good for customized plots).

✅ 5. `st.altair_chart()`

Purpose: Display charts using the **Altair** library (declarative grammar of graphics).

```
import altair as alt

chart = alt.Chart(data.reset_index()).mark_line().encode(
    x='index',
    y='A'
)
st.altair_chart(chart, use_container_width=True)
```

🟢 **Output**: Responsive, interactive chart with Altair's advanced features (tooltips, filters).

✅ 6. `st.vega_lite_chart()`

Purpose: Similar to Altair, uses a JSON-style Vega-Lite spec.

```
st.vega_lite_chart(
    data.reset_index(),
    {
        'mark': 'line',
        'encoding': {
            'x': {'field': 'index', 'type': 'quantitative'},
            'y': {'field': 'A', 'type': 'quantitative'}
        }
    }
)
```

🟢 **Output**: Custom line chart, built using Vega-Lite schema.

✅ 7. `st.plotly_chart()`

Purpose: Use **Plotly** for rich interactive plots.

```
import plotly.express as px

fig = px.line(data, y='B', title='Plotly Line Chart')
st.plotly_chart(fig, use_container_width=True)
```

🟢 **Output**: Zoomable, hoverable chart. Great for dashboards.

✅ 8. `st.bokeh_chart()`

Purpose: Render **Bokeh** plots (another Python viz lib).

```
from bokeh.plotting import figure

p = figure(title="Bokeh Line Chart", x_axis_label='x', y_axis_label='y')
p.line(data.index, data['C'], legend_label="C", line_width=2)
st.bokeh_chart(p, use_container_width=True)
```

🟢 **Output**: Interactive plot (like Plotly, but with Bokeh controls).

✅ 9. `st.graphviz_chart()`

Purpose: Draw **graph structures** (like flowcharts, trees).

```
import graphviz

dot = graphviz.Digraph()
dot.node("A")
dot.node("B")
dot.edge("A", "B", label="connects to")
st.graphviz_chart(dot)
```

🟢 **Output**: A diagram showing A → B with a connecting arrow.

✅ 10. `st.map()`

Purpose: Plot latitude and longitude points on a map (using PyDeck under the hood).

```
import pandas as pd

map_data = pd.DataFrame({
    'lat': [12.9716, 13.0827],
    'lon': [77.5946, 80.2707]
})
st.map(map_data)
```

🟢 **Output**: Points marked on a zoomable map (Bangalore, Chennai).

🔁 Summary Table

Function	Type	Library Used	Interaction	Use Case
`st.line_chart()`	Line Chart	Built-in	✅	Quick data trends
`st.area_chart()`	Area Chart	Built-in	✅	Emphasize volume/coverage
`st.bar_chart()`	Bar Chart	Built-in	✅	Category comparison
`st.pyplot()`	Any Plot	Matplotlib	❌	Custom scientific plotting
`st.altair_chart()`	Any Plot	Altair	✅	Declarative + interactive graphics
`st.vega_lite_chart()`	Any Plot	Vega-Lite	✅	Custom structured plots via dict
`st.plotly_chart()`	Any Plot	Plotly	✅ ✅	Hover, zoom, filter
`st.bokeh_chart()`	Any Plot	Bokeh	✅	Custom dashboard components
`st.graphviz_chart()`	Flow Diagrams	Graphviz	❌	Graphs, trees, networks
`st.map()`	Geo Map	PyDeck	✅	Latitude/longitude visualization

Start coding or generate with AI.

4.Input Widgets

🎯 BUTTONS

✅ `st.button(label)`

Displays a simple clickable button.

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

🟢 **Output**: A button labeled “Click Me”; clicking shows the message.

✅ `st.download_button(label, data, file_name)`

Lets users download data as a file.

```
text = "Hello, download me!"
st.download_button("Download Text", text, file_name="hello.txt")
```

🟢 **Output**: A button to download a `.txt` file containing the text.

✅ `st.link_button(label, url)`

Navigates to a URL on click.

```
st.link_button("Go to Google", "https://www.google.com")
```

🟢 **Output**: A button that opens Google in a new tab.

✅ CHECKS & CHOICES

✅ `st.checkbox(label)`

Toggle True/False.

```
if st.checkbox("Show content"):
    st.write("Checkbox checked!")
```

🟢 **Output**: A checkbox; when checked, it shows a message.

✅ `st.radio(label, options)`

Select one from many options.

```
option = st.radio("Choose color", ["Red", "Green", "Blue"])
st.write("You selected:", option)
```

🟢 **Output**: Vertical radio buttons.

✅ `st.selectbox(label, options)`

Dropdown menu (single selection).

```
selected = st.selectbox("Pick a number", [1, 2, 3])
st.write("You chose:", selected)
```

🟢 **Output**: Dropdown menu with selected option.

✅ `st.multiselect(label, options)`

Select multiple items.

```
choices = st.multiselect("Select your hobbies", ["Reading", "Gaming", "Coding"])
st.write("You selected:", choices)
```

🟢 **Output**: Dropdown with checkboxes.

🔁 SLIDERS

✅ `st.slider(label, min, max, value)`

Numeric slider (int or float).

```
age = st.slider("Select age", 0, 100, 25)
st.write("Age is:", age)
```

🟢 **Output**: Slider bar with selected value.

✅ `st.select_slider(label, options)`

Slider with custom string or list options.

```
level = st.select_slider("Select level", options=["Low", "Medium", "High"])
st.write("Selected:", level)
```

🟢 **Output**: Slider showing “Low”, “Medium”, “High”.

📝 TEXT & NUMBER INPUTS

✅ `st.text_input(label)`

Single-line text box.

```
name = st.text_input("Enter your name")
st.write("Hello,", name)
```

🟢 **Output**: Text box.

✅ `st.text_area(label)`

Multi-line text input.

```
bio = st.text_area("Enter your bio")
st.write("Your bio:", bio)
```

🟢 **Output**: Large text area for long input.

✅ `st.number_input(label)`

Numeric input with step, range.

```
num = st.number_input("Pick a number", min_value=0, max_value=10)
st.write("You picked:", num)
```

🟢 **Output**: Number box with increment/decrement buttons.

📅 TIME INPUTS

✅ `st.date_input(label)`

Date picker.

```
dob = st.date_input("Select your birthday")
st.write("DOB:", dob)
```

🟢 **Output**: Calendar widget.

✅ `st.time_input(label)`

Time picker.

```
time = st.time_input("Select time")
st.write("Selected time:", time)
```

🟢 **Output**: Time selector.

📁 FILES & CAMERA

✅ `st.file_uploader(label)`

Upload files.

```
file = st.file_uploader("Upload a CSV")
if file:
    st.write("Uploaded:", file.name)
```

🟢 **Output**: File browser; shows file name on upload.

✅ `st.camera_input(label)`

Take picture from webcam.

```
photo = st.camera_input("Take a selfie")
if photo:
    st.image(photo)
```

🟢 **Output**: Camera preview and capture.

🎨 COLOR

✅ `st.color_picker(label)`

Select a color.

```
color = st.color_picker("Pick a color", "#00f900")
st.write("You picked:", color)
```

🟢 **Output**: Color picker input.

5.Layout and Containers Control Flow Media Elements

✅ `st.container()`

Already covered — used for grouping widgets together.

✅ `st.columns()`

Already covered — splits layout into horizontal columns.

✅ `st.expander()`

Already covered — collapsible section to hide/show content.

✅ `st.sidebar`

Used like `st.sidebar.button()`, `st.sidebar.selectbox()`, etc.

```
option = st.sidebar.selectbox("Pick one:", ["Home", "Profile", "Settings"])
st.write(f"You chose: {option}")
```

✅ `st.empty()`

Placeholder for dynamic updates (loading, live updates).

```
import time

placeholder = st.empty()
for i in range(5):
    placeholder.text(f"Step {i+1}/5")
    time.sleep(0.5)
placeholder.text("Done!")
```

✅ `st.tabs()`

Tabbed UI for organizing sections.

```
tab1, tab2 = st.tabs(["📊 Data", "🛠️ Settings"])
with tab1:
    st.write("Here is the data...")
with tab2:
    st.write("Adjust your settings here.")
```

✅ `st.popover()`

Floating pop-up panel (like a mini modal). Requires Streamlit ≥1.32.

```
with st.popover("Advanced Options"):
    st.checkbox("Enable beta features")
    st.button("Apply Settings")
```

✅ `st.divider()`

Draws a horizontal line to separate content.

```
st.write("Section A")
st.divider()
st.write("Section B")
```

📺 Media Display

✅ `st.image()`

Displays an image (JPG, PNG, etc.)

```
st.image("https://placekitten.com/400/300", caption="Cute Kitten")
```

✅ `st.audio()`

Plays audio files.

```
audio_file = open("sample.mp3", "rb")
st.audio(audio_file.read(), format="audio/mp3")
```

✅ `st.video()`

Plays videos (MP4, YouTube links, etc.)

```
st.video("https://www.youtube.com/watch?v=5qap5aO4i9A")
```

🔲 Control Execution

✅ `st.stop()`

Halts script execution at that point (used for debugging or early exits).

```
st.write("Before stop")
st.stop()
st.write("This will never run")
```

📝 Forms

✅ `st.form()` + `st.form_submit_button()`

Group multiple inputs into a form with a submit button.

```
with st.form("my_form"):
    name = st.text_input("Name")
    age = st.number_input("Age")
    submitted = st.form_submit_button("Submit")

    if submitted:
        st.success(f"Hello {name}, age {age}")
```

🧪 Experimental Features

✅ `st.experimental_rerun()`

Re-runs the app from the top immediately.

```
st.write("This runs first")

if st.button("Rerun"):
    st.experimental_rerun()
```

✅ `st.experimental_fragment()`

Optimizes part of the app to re-render independently (useful for performance with heavy components).

```
@st.experimental_fragment
def expensive_plot():
    st.write("Rendering heavy chart...")
    st.line_chart(range(1000))
```

```
expensive_plot()
```

Summary Table

Function	Purpose
`st.sidebar`	Sidebar UI elements
`st.empty()`	Placeholder for dynamic/live updates
`st.tabs()`	Tabbed layout
`st.popover()`	Small floating menu
`st.divider()`	Horizontal line separator
`st.image()`	Displays image
`st.audio()`	Plays audio
`st.video()`	Plays video
`st.stop()`	Stops execution of code
`st.form()`	Wrap multiple inputs into a form
`st.form_submit_button()`	Submit button for form
`st.experimental_rerun()`	Force app rerun
`st.experimental_fragment()`	Selective rendering for performance

Double-click (or enter) to edit

Configuration and Caching

`st.get_option()`

Get current value of a Streamlit config setting.

```
theme = st.get_option("theme.base")
st.write(f"Current theme: {theme}")
```

`st.set_option()`

Set a Streamlit config option **dynamically**.

```
st.set_option("client.showSidebarNavigation", True)
```

⚠️ Limited use in practice — usually used in `.streamlit/config.toml`.

✅ `st.cache_data()`

Caches **data-processing functions** (pure functions).

```
@st.cache_data
def load_data():
    time.sleep(2)  # simulate delay
    return {"value": 42}

data = load_data()
st.write(data)
```

🔹 Re-runs **only when input arguments change**. Ideal for reading files, querying APIs, etc.

✅ `st.cache_resource()`

Caches **expensive resources** like models, DB connections, etc.

```
@st.cache_resource
def load_model():
    return SomeHeavyModel()

model = load_model()
```

🕰️ Deprecated:

❌ `st.cache()`

→ Replaced by `st.cache_data()` and `st.cache_resource()` (since Streamlit v1.18).

❌ `st.experimental_memo()`

→ Deprecated; use `st.cache_data()`.

❌ `st.experimental_singleton()`

→ Deprecated; use `st.cache_resource()`.

📊 Status & Progress

✅ `st.progress()`

Shows progress bar (0.0 to 1.0).

```
import time

progress = st.progress(0)
for i in range(100):
    progress.progress(i + 1)
    time.sleep(0.01)
```

✅ `st.spinner()`

Loading animation with message.

```
with st.spinner("Loading..."):
    time.sleep(2)
st.success("Done!")
```

✅ `st.status()`

Group spinner, success, warning, etc. under one status card (Streamlit ≥1.26).

```
with st.status("Processing...", expanded=True) as status:
    st.write("Step 1: Starting")
    time.sleep(1)
    st.write("Step 2: Finishing")
    status.update(label="Done", state="complete", expanded=False)
```

✅ `st.toast()`

Show a toast (small notification at the corner). New in v1.26+.

```
st.toast("File uploaded successfully!", icon="✅")
```

🎉 Animations & Effects

✅ `st.balloons()`

Celebrate success 🎈

```
st.balloons()
```

✅ `st.snow()`

Winter effect ❄️

```
st.snow()
```

⚠️ Messages and Errors

✅ `st.success()`

Green success box.

```
st.success("Operation completed!")
```

✅ `st.info()`

Blue info box.

```
st.info("This feature is in beta.")
```

✅ `st.warning()`

Yellow warning box.

```
st.warning("You are reaching your limit.")
```

✅ `st.error()`

Red error box.

```
st.error("Something went wrong.")
```

✅ `st.exception()`

Show an exception with traceback.

```
try:
    1 / 0
except Exception as e:
    st.exception(e)
```

✅ Summary Table

Function	Purpose
`st.get_option()`	Get Streamlit config
`st.set_option()`	Set Streamlit config dynamically
`st.cache_data()`	Cache data functions
`st.cache_resource()`	Cache models/resources
`st.progress()`	Progress bar
`st.spinner()`	Loading animation
`st.status()`	Status box with progress
`st.toast()`	Pop-up toast notification
`st.balloons()`	Celebration animation
`st.snow()`	Snow animation
`st.success()`	Green box
`st.info()`	Blue info box
`st.warning()`	Yellow warning box
`st.error()`	Red error box
`st.exception()`	Display exception traceback

---

6 Session and State

🔍 What is `st.session_state`?

Every time Streamlit runs top to bottom, it **restarts the script**. So if you want to store **state** (e.g., a counter, form input, or model), `st.session_state` is your solution.

It acts like a **persistent dictionary**, available across:

- Reruns
- Widgets
- Pages
- Tabs

- Conditional logic

🧠 Why use it?

Without `st.session_state`, this code loses count every rerun:

```
if st.button("Click"):
    st.write("Clicked!")
    count = 1
    count += 1  # lost after rerun
```

With `st.session_state`:

```
if "count" not in st.session_state:
    st.session_state.count = 0

if st.button("Click"):
    st.session_state.count += 1

st.write("Count:", st.session_state.count)
```

✅ Now the count persists across clicks.

🧪 Use Cases

1. **Counter Example**

```
if "count" not in st.session_state:
    st.session_state.count = 0

if st.button("Add"):
    st.session_state.count += 1

if st.button("Reset"):
    st.session_state.count = 0

st.write("Current count:", st.session_state.count)
```

---

2. **Using with Widgets**

Widgets store their values in `session_state` automatically **if you assign a `key`**.

```
st.text_input("Enter name", key="username")
st.write("Hello,", st.session_state.username)
```

---

3. **Triggering Events**

Simulate "once-only" actions like login or toggling UI.

```
if "logged_in" not in st.session_state:
    st.session_state.logged_in = False

if st.button("Login"):
    st.session_state.logged_in = True

if st.session_state.logged_in:
    st.success("Welcome! You're logged in.")
```

---

4. **Multi-Page Apps (Shared State)**

Shared state across pages.

```
# Page 1
st.text_input("Enter data", key="my_input")

# Page 2
st.write("You entered:", st.session_state.my_input)
```

---

5. **Callbacks with Buttons**

Define a callback for logic after button press.

```
def increase():
    st.session_state.count += 1

st.button("Increase", on_click=increase)
```

🔧 Methods and Tips

Method	Description
`st.session_state["key"]`	Access session variable

Method	Description
`st.session_state.key`	Shortcut for dot notation
`st.session_state.get("key")`	Safe access (returns `None` if not set)
`st.session_state.clear()`	Clears all session state
`st.session_state.update({...})`	Bulk update values
`del st.session_state["key"]`	Deletes a variable

💡 Best Practices

- Always check `if "key" not in st.session_state` before using.
- Use `key="some_name"` in widgets to sync with session state.
- Use **callbacks** (`on_click`, `on_change`) to modify state cleanly.
- Don't store large objects unnecessarily (to save memory).

🧪 Example: Wizard Flow App

```
if "step" not in st.session_state:
    st.session_state.step = 1

if st.button("Next"):
    st.session_state.step += 1

st.write(f"You are on step {st.session_state.step}")
```

✅ Summary

Feature	Supports ✅
Persistent across runs	✅
Shared across pages	✅
Widget interaction	✅ (with `key`)
Button/Callback logic	✅ (`on_click`)
Clean initialization	✅ (`if "x" not in st.session_state`)

Double-click (or enter) to edit

Start coding or generate with AI.

11.Experimental (Subject to change)

---

1. `st.experimental_get_query_params()`

What it does:

- Retrieves URL **query parameters** as a dictionary.
- Useful for **reading URL params** to control app state or navigation.

Example:

If your app URL is: `http://localhost:8501/?page=2&user=alice`

```
params = st.experimental_get_query_params()
st.write(params)
```

Output:

```
{'page': ['2'], 'user': ['alice']}
```

Each value is a **list of strings** (because URLs can have repeated params).

---

2. `st.experimental_set_query_params()`

What it does:

- Sets (or updates) the URL **query parameters** dynamically.
- Great for deep-linking or reflecting app state in URL.

Example:

```
st.experimental_set_query_params(page="3", user="bob")
st.write("URL updated!")
```

After running, your URL becomes: `...?page=3&user=bob`

---

3. `st.experimental_user()`

What it does:

- Returns information about the **current user** logged into Streamlit Cloud (if any).
- Provides a dict with user info like `id`, `email`, `username`.

Example:

```
user = st.experimental_user()
st.write(user)
```

Output example:

```
{'email': 'alice@example.com', 'id': '123', 'username': 'alice'}
```

⚠️ Only works on **Streamlit Cloud** or Enterprise with authentication.

---

4. `st.experimental_theme()`

What it does:

- Returns the **active theme** configuration (colors, fonts, etc.) as a dict.
- Useful if you want to adapt your app dynamically based on the theme.

Example:

```
theme = st.experimental_theme()
st.json(theme)
```

Output might include:

```
{
  "primaryColor": "#F63366",
  "backgroundColor": "#FFFFFF",
  "font": "sans serif",
  ...
}
```

---

5. `st.experimental_data_editor()`

What it does:

- Provides an **interactive data editor** widget to edit data frames or tables directly in the app.

- Similar to `st.dataframe()` but **editable** by users.

Example:

```
import pandas as pd

df = pd.DataFrame({
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [24, 30, 22]
})

edited_df = st.experimental_data_editor(df)
st.write("You edited:")
st.write(edited_df)
```

- Users can **modify** cells, add/remove rows, and `edited_df` reflects changes in real time.

Summary Table

Function	Purpose	Notes
`st.experimental_get_query_params()`	Get URL query parameters as dict	Values as lists
`st.experimental_set_query_params()`	Set/update URL query parameters	Updates browser URL
`st.experimental_user()`	Get current logged-in user info	Only on Streamlit Cloud/Enterprise

Start coding or generate with AI.