I'll explain every line in extreme detail and why each choice was made:

**1. Import Statements - Why Each One?**

python

import streamlit as st

**Why streamlit?**

* Streamlit is specifically designed for creating web apps from Python scripts
* It handles all the HTML/CSS/JavaScript complexity behind the scenes
* Perfect for data applications because it integrates seamlessly with pandas/dataframes
* Can run natively inside Snowflake (Snowflake Streamlit Apps)
* Alternative would be Flask/Django, but those require much more web development knowledge

python

from snowflake.snowpark.context import get\_active\_session

**Why get\_active\_session specifically?**

* When running inside Snowflake's Streamlit environment, this automatically provides the database connection
* No need to handle credentials, connection strings, or authentication
* Alternative would be creating a manual connection with username/password, but that's insecure and unnecessary in Snowflake Streamlit
* This is the Snowpark way (modern Snowflake Python connector) vs older snowflake-connector-python

python

import pandas as pd

**Why pandas?**

* Streamlit's st.data\_editor() expects pandas DataFrames as input
* Pandas provides easy data manipulation (like dropna(), iterrows())
* Industry standard for data manipulation in Python
* Alternative would be working with raw Python lists/dicts, but much more complex

**2. Page Setup**

python

st.title("Insert Masking Master Data")

**Why st.title() and not st.header()?**

* st.title() creates the largest heading (H1) - appropriate for main page title
* st.header() would be H2 - used for sections within a page
* Sets clear context for users about what this app does
* Alternative: st.markdown("# Title") but st.title() is more semantic

python

session = get\_active\_session()

**Why store in a variable?**

* We'll use session multiple times throughout the app
* More efficient than calling get\_active\_session() repeatedly
* Cleaner code - easier to read
* If connection method changes, only need to update one line

**3. Function Definition - insert\_data(df)**

python

def insert\_data(df):

**Why create a function instead of inline code?**

* **Separation of concerns** - UI logic separate from database logic
* **Reusability** - could call this function from multiple places
* **Testability** - easier to test database operations separately
* **Readability** - main code flow is cleaner
* **Error handling** - contained in one place

python

try:

**Why use try/except?**

* Database operations can fail for many reasons (network, permissions, SQL syntax, data types)
* Want to show user-friendly error messages instead of Python stack traces
* Prevents app from crashing completely
* Good practice for any external system interaction

python

df = df.dropna(how='all')

**Why dropna(how='all')?**

* st.data\_editor might create empty rows when users click "+"
* how='all' only removes rows where EVERY column is empty/null
* If user partially fills a row, we keep it (and validation will catch it later)
* Alternative dropna() would remove rows with ANY empty cell - too aggressive
* Alternative how='any' same as dropna() - would lose partial data

python

if df.empty: return False

**Why return False instead of raising an exception?**

* Empty dataframe isn't really an "error" - just nothing to do
* Caller can handle this gracefully
* Follows pattern where True = success, False = nothing to do
* More efficient than exception handling

**4. VALUES Clause Generation**

python

values = ','.join([f"('{r['DATABASE\_NAME']}','{r['SCHEMA\_NAME']}','{r['TABLE\_NAME']}','{r['COLUMN\_NAME']}','{r['CLASSIFICATION\_NAME']}')" for \_, r in df.iterrows()])

**Why this complex approach instead of using pandas.to\_sql()?**

* **Custom business logic** - We need the CASE statements for policy assignment
* **Control over SQL** - Your specific INSERT...SELECT...FROM VALUES pattern
* **Performance** - Single INSERT vs multiple individual INSERTs
* **Atomicity** - All records succeed or all fail together

**Breaking down the list comprehension:**

python

[f"('{r['DATABASE\_NAME']}','{r['SCHEMA\_NAME']}','{r['TABLE\_NAME']}','{r['COLUMN\_NAME']}','{r['CLASSIFICATION\_NAME']}')" for \_, r in df.iterrows()]

* df.iterrows() - loops through each row as (index, row\_data)
* \_, r - we don't need the index, just the row data
* f"(...)" - f-string for string formatting
* '{r['COLUMN\_NAME']}' - wraps each value in single quotes for SQL
* Creates a list like: ["('DB1','SCH1','TBL1','COL1','PII')", "('DB2','SCH2','TBL2','COL2','PHI')"]

**Why ','.join()?**

* Converts list to single string with commas between elements
* SQL VALUES clause needs format: VALUES ('a','b','c'),('d','e','f')
* More efficient than string concatenation in a loop

**5. The Complex SQL Query**

sql

INSERT INTO COMMON\_DB.UTIL\_SCH.MASKING\_MASTER\_INFO (DATABASE\_NAME, SCHEMA\_NAME, TABLE\_NAME, COLUMN\_NAME, CLASSIFICATION\_NAME, POLICY\_NAME, MASKING\_STATUS, USER\_REQUESTED)

**Why specify column names?**

* **Explicit is better than implicit** - makes code self-documenting
* **Future-proof** - if table structure changes, this won't break
* **Skips auto-generated columns** - ID, INSERT\_TIME, DELETED\_TIME are handled by defaults
* **Required for this approach** - the SELECT needs to match these columns

sql

SELECT DATA.DATABASE\_NAME, DATA.SCHEMA\_NAME, DATA.TABLE\_NAME, DATA.COLUMN\_NAME, DATA.CLASSIFICATION\_NAME,

**Why SELECT instead of direct INSERT VALUES?**

* **Enables transformation** - we can add business logic with CASE statements
* **Allows complex logic** - duplicate checking with EXISTS
* **Single statement** - all logic in one atomic operation
* **Performance** - set-based operation instead of row-by-row

**Policy Assignment CASE Statement:**

sql

CASE WHEN UPPER(DATA.CLASSIFICATION\_NAME) IN ('PII','RESTRICTED','CONFIDENTIAL','PHI') THEN 'SENSITIVE'

WHEN UPPER(DATA.CLASSIFICATION\_NAME) IN ('ELEVATED') THEN 'ELEVATED'

ELSE 'NO\_POLICY\_TO\_APPLY' END

**Why UPPER()?**

* **Case-insensitive matching** - 'pii', 'PII', 'Pii' all work
* **User-friendly** - users don't have to worry about exact capitalization
* **Data consistency** - standardizes input

**Why this specific business logic?**

* **Data sensitivity levels** - PII/RESTRICTED/CONFIDENTIAL/PHI are high-risk data types
* **Different policies** - SENSITIVE vs ELEVATED require different masking approaches
* **Default fallback** - NO\_POLICY\_TO\_APPLY for unknown classifications

**Status Assignment Logic:**

sql

CASE WHEN (previous CASE) = 'NO\_POLICY\_TO\_APPLY' THEN 'NO\_POLICY\_TO\_APPLY'

WHEN EXISTS(SELECT 1 FROM COMMON\_DB.UTIL\_SCH.MASKING\_MASTER\_INFO MM WHERE ...) THEN 'ALREADY\_APPLIED'

ELSE 'NEED\_TO\_APPLY' END

**Why nested CASE statement?**

* **Efficiency** - if no policy applies, skip expensive EXISTS check
* **Logic flow** - policy determines whether we even check for duplicates
* **Business rules** - different status meanings for different scenarios

**Why EXISTS instead of COUNT?**

* **Performance** - EXISTS stops at first match, COUNT scans all matches
* **Clarity** - we only care if ANY matching record exists, not how many
* **Standard practice** - EXISTS is the SQL standard for "does this exist?" queries

**Why check DELETED\_TIME IS NULL?**

* **Soft deletes** - records aren't physically deleted, just marked
* **Reactivation** - previously deleted records can be reprocessed
* **Audit trail** - keeps history of what was masked when

**Why CURRENT\_USER()?**

* **Audit requirement** - track who requested each masking operation
* **Automatic** - no need for users to enter their name
* **Accurate** - uses actual database user, can't be faked

**6. SQL Execution**

python

session.sql(query).collect()

return True

**Why .collect()?**

* **Forces execution** - Snowpark uses lazy evaluation
* **Gets results** - even though we don't use them, ensures query ran
* **Error detection** - exceptions happen here if SQL fails
* Alternative .execute() exists but .collect() is more explicit

**Why return True?**

* **Success indicator** - tells caller the operation completed
* **Simple contract** - True/False is easier than exceptions
* **Allows chaining** - if insert\_data(df): show\_success()

**7. Error Handling**

python

except Exception as e:

st.error(f"Error: {e}")

return False

**Why catch generic Exception?**

* **Broad safety net** - database errors, network issues, SQL syntax errors all caught
* **User experience** - show friendly message instead of crash
* **Graceful degradation** - app continues working

**Why st.error()?**

* **Visual feedback** - red error box is clear to users
* **Streamlit integration** - appears in the right place in UI
* **Non-blocking** - doesn't stop app execution

**8. Data Editor Setup**

python

data = st.data\_editor(pd.DataFrame(columns=['DATABASE\_NAME', 'SCHEMA\_NAME', 'TABLE\_NAME', 'COLUMN\_NAME', 'CLASSIFICATION\_NAME']), num\_rows="dynamic", use\_container\_width=True)

**Why st.data\_editor() instead of st.dataframe()?**

* **Editable** - st.dataframe() is read-only display
* **Interactive** - users can add/remove rows, edit cells
* **Built-in validation** - handles data types automatically
* **Modern UI** - better user experience than form fields

**Why pd.DataFrame(columns=[...])?**

* **Structure definition** - establishes the 5 required columns
* **Empty starting state** - no data initially, users add their own
* **Column order** - ensures columns appear in logical sequence
* **Type inference** - pandas helps Streamlit understand data structure

**Why these specific column names?**

* **Database mapping** - exact match to your table schema
* **Descriptive** - clear what each column represents
* **Consistent** - matches your existing database naming convention

**Why num\_rows="dynamic"?**

* **Flexibility** - users can add as many rows as needed
* **User control** - + and - buttons for row management
* **Efficient** - no arbitrary limits on data entry
* Alternative: num\_rows=10 would fix row count, less flexible

**Why use\_container\_width=True?**

* **Responsive design** - table uses full available width
* **Better readability** - columns have more space for data
* **Professional appearance** - looks more polished
* Alternative: fixed width might cut off content

**9. Button and Validation Logic**

python

if st.button("Insert Data"):

**Why st.button() instead of st.form\_submit\_button()?**

* **Simplicity** - no need for complex form state management
* **Immediate feedback** - button press triggers immediate action
* **Less code** - don't need st.form() wrapper
* **Appropriate for simple case** - single action, no complex form logic

**Complex validation logic:**

python

if not data.empty and all(not data[col].isna().any() and not (data[col] == '').any() for col in data.columns):

**Breaking this down:**

1. not data.empty - ensures user entered at least one row
2. all(...) - ensures EVERY condition is true
3. for col in data.columns - checks every column
4. not data[col].isna().any() - no NaN/null values in this column
5. not (data[col] == '').any() - no empty strings in this column

**Why this complex validation?**

* **Data quality** - prevents incomplete records in database
* **User experience** - clear feedback about what's wrong
* **Database integrity** - avoids SQL errors from null values
* **Business rules** - all 5 fields are required for masking logic

**Why check both .isna() and empty strings?**

* **Different empty types** - pandas differentiates null vs empty string
* **User behavior** - users might delete content (→ empty string) vs never enter it (→ null)
* **Comprehensive** - catches all forms of "missing" data

**10. Success/Error Messages**

python

if insert\_data(data):

st.success("Data inserted successfully!")

else:

st.warning("Please fill all required fields.")

**Why st.success() vs st.info()?**

* **Visual hierarchy** - green success box clearly indicates completion
* **User psychology** - positive reinforcement for successful action
* **Standard UX** - green = success is universal design pattern

**Why st.warning() vs st.error()?**

* **Severity level** - missing fields is user mistake, not system error
* **Color coding** - yellow warning vs red error
* **Guidance** - warning suggests what to do, error suggests something broke

**Design Decisions Summary:**

1. **Minimal dependencies** - only essential imports
2. **Error resilience** - comprehensive exception handling
3. **User experience** - clear feedback, intuitive interface
4. **Performance** - single SQL operation, efficient queries
5. **Maintainability** - clean separation of concerns
6. **Business logic** - embedded complex rules in SQL
7. **Security** - uses authenticated session, no SQL injection risk
8. **Scalability** - handles multiple rows efficiently

Every choice was made to balance simplicity, robustness, and user experience while meeting your specific business requirements for data masking management.