GOOGLE

DATA ANALYST INTERVIEW EXPERIENCE

YOE:0-3

CTC: 22-24LPA

SQL

Question 1: Write a query to calculate the bounce rate for a website using session and page view data.

Concept: Bounce rate is the percentage of single-page sessions (sessions in which the user viewed only one page) divided by all sessions.

Assumptions:

- You have two tables: sessions and page_views.
- sessions table contains session_id and potentially other session-related details.
- page_views table contains session_id and page_url (or a similar identifier for a page).

Input Tables:

1. sessions table:

session_id	start_time	user_id
S001	2025-06-19 10:00:00	U101
S002	2025-06-19 10:05:00	U102
S003	2025-06-19 10:10:00	U101
S004	2025-06-19 10:15:00	U103
S005	2025-06-19 10:20:00	U102

2. page_views table:

page_view_id	session_id	page_url	view_time
PV001	S001	/home	2025-06-19 10:00:15
PV002	S001	/product	2025-06-19 10:01:00
PV003	S002	/about	2025-06-19 10:05:30
PV004	S003	/contact	2025-06-19 10:10:20
PV005	S003	/faq	2025-06-19 10:11:00
PV006	S003	/privacy	2025-06-19 10:11:45
PV007	S004	/index	2025-06-19 10:15:40
PV008	S005	/services	2025-06-19 10:20:10
PV009	S005	/pricing	2025-06-19 10:21:00

SQL Query:

```
SELECT
  (COUNT(DISTINCT CASE WHEN pv_count = 1 THEN session_id ELSE NULL END) * 1.0 / COUNT(DISTINCT session_id)) AS bounce_rate
FROM (
  SELECT
    session_id,
    COUNT(page_url) AS pv_count
  FROM
    page_views
  GROUP BY
    session_id
) AS session_page_counts;
```

Explanation:

1. **Inner Query (session_page_counts):**
    - SELECT session_id, COUNT(page_url) AS pv_count FROM page_views GROUP BY session_id: This subquery calculates the total number of page views for each session_id.
2. **Outer Query:**
    - COUNT(DISTINCT session_id): This counts the total number of unique sessions.
    - COUNT(DISTINCT CASE WHEN pv_count = 1 THEN session_id ELSE NULL END): This counts the number of unique sessions where pv_count (page views for that session) is equal to 1. These are our "bounced" sessions.
    - * 1.0: We multiply by 1.0 to ensure floating-point division, giving us a decimal bounce rate.
    - The result is the ratio of bounced sessions to total sessions.

Output:

bounce_rate
0.4000000000000000

(Interpretation: 2 out of 5 sessions were bounces (S002 and S004 each had only one page view)).

Question 2: From a user_activity table, find the number of users who were active on at least 15 days in a given month.

Concept: This requires counting distinct days a user was active within a specific month and then filtering for users who meet the 15-day threshold.

Assumptions:

- You have a user_activity table with user_id and activity_date.
- "Active" means there's at least one entry for that user on that day.

Input Table:

1. user_activity table:

activity_id	user_id	activity_date	activity_type
1	U101	2025-05-01	login
2	U101	2025-05-01	view_product
3	U102	2025-05-01	login
4	U101	2025-05-02	add_to_cart
5	U103	2025-05-03	login
6	U101	2025-05-10	purchase
7	U101	2025-05-15	login
8	U101	2025-05-16	logout
9	U101	2025-05-17	login
10	U101	2025-05-18	view_product
11	U101	2025-05-19	add_to_cart
12	U101	2025-05-20	purchase
13	U101	2025-05-21	login
14	U101	2025-05-22	logout
15	U101	2025-05-23	login
16	U101	2025-05-24	view_product
17	U101	2025-05-25	add_to_cart
18	U101	2025-05-26	purchase
19	U101	2025-05-27	login
20	U101	2025-05-28	logout

21	U102	2025-05-05	login
22	U102	2025-05-10	view_product
23	U102	2025-05-12	purchase
24	U102	2025-05-14	login
25	U103	2025-05-05	login
26	U103	2025-05-10	view_product
27	U103	2025-05-12	purchase

SQL Query:

```
SELECT
  COUNT(user_id) AS num_users_active_15_days
FROM (
  SELECT
    user_id,
    COUNT(DISTINCT activity_date) AS distinct_active_days
  FROM
    user_activity
  WHERE
    STRFTIME('%Y-%m', activity_date) = '2025-05' -- For a given month, e.g., May 2025
  GROUP BY
    user_id
  HAVING
    distinct_active_days >= 15
) AS active_users_summary;
```

Explanation:

1. **Inner Query (active_users_summary):**

- SELECT user_id, COUNT(DISTINCT activity_date) AS distinct_active_days: This counts the number of unique activity_date entries for each user_id.
- FROM user_activity: Specifies the table.
- WHERE STRFTIME('%Y-%m', activity_date) = '2025-05': This filters the data for a specific month (May 2025 in this example). STRFTIME (or similar date formatting functions like TO_CHAR in PostgreSQL/Oracle, FORMAT in SQL Server, DATE_FORMAT in MySQL) extracts the year and month from the activity_date.
- GROUP BY user_id: Groups the results by user to count distinct days per user.
- HAVING distinct_active_days >= 15: Filters these grouped results, keeping only those users who have been active on 15 or more distinct days.

2. **Outer Query:**
   - COUNT(user_id) AS num_users_active_15_days: This simply counts the number of user_ids that met the criteria from the inner query.

Output:

num_users_active_15_days
1

(Interpretation: Only User U101 was active on 15 or more days in May 2025.)

Question 3: You have a search_logs table with query, timestamp, and user_id. Find the top 3 most frequent search queries per week.

Concept: This involves grouping by week, then by query, counting the occurrences, and finally ranking queries within each week to get the top 3.

Assumptions:

- You have a search_logs table with query, timestamp, and user_id.

Input Table:

1. search_logs table:

log_id	query	timestamp	user_id
1	"data analyst"	2025-06-03 10:00:00	U101
2	"SQL basics"	2025-06-03 11:00:00	U102
3	"data analyst"	2025-06-04 09:30:00	U103
4	"Python"	2025-06-05 14:00:00	U101
5	"data analyst"	2025-06-05 16:00:00	U102
6	"SQL basics"	2025-06-06 10:00:00	U101
7	"Python"	2025-06-06 11:00:00	U103
8	"machine learning"	2025-06-07 10:00:00	U101
9	"data analyst"	2025-06-10 09:00:00	U102
10	"SQL advanced"	2025-06-10 10:00:00	U101
11	"SQL advanced"	2025-06-11 11:00:00	U103
12	"Python"	2025-06-11 12:00:00	U102
13	"data analyst"	2025-06-12 13:00:00	U101
14	"machine learning"	2025-06-13 14:00:00	U103
15	"Python"	2025-06-13 15:00:00	U101
16	"data visualization"	2025-06-14 16:00:00	U102

(Note: Assuming week starts on Monday for simplicity, but the exact week start day depends on the SQL dialect's date functions.)

SQL Query:

SELECT

```
    week_start_date,
    query,
    query_count
FROM (
    SELECT
        STRFTIME('%Y-%W', timestamp) AS week_identifier, -- Or DATE_TRUNC('week', timestamp) for PostgreSQL, etc.
        MIN(DATE(timestamp, 'weekday 0')) AS week_start_date, -- Adjust 'weekday 0' for your desired week start (Sunday). Use 'weekday 1' for Monday.
        query,
        COUNT(query) AS query_count,
        ROW_NUMBER() OVER (PARTITION BY STRFTIME('%Y-%W', timestamp) ORDER BY COUNT(query) DESC) AS rn
    FROM
        search_logs
    GROUP BY
        week_identifier,
        query
) AS weekly_query_counts
WHERE
    rn <= 3
ORDER BY
    week_start_date,
    query_count DESC;
```

Explanation:

1. **Inner Query (weekly_query_counts):**

- STRFTIME('%Y-%W', timestamp) AS week_identifier: This extracts the year and week number from the timestamp. %W typically represents the week number of the year, with the first Monday as the first day of week 01. (For different SQL dialects, you'd use functions like DATE_TRUNC('week', timestamp) in PostgreSQL, DATEPART(week, timestamp) in SQL Server, or WEEK(timestamp) in MySQL).
- MIN(DATE(timestamp, 'weekday 0')) AS week_start_date: This tries to get a clear start date for the week. DATE(timestamp, 'weekday 0') in SQLite will give you the most recent Sunday. Adjust 'weekday 1' for Monday, etc., based on your database. This is important for a more readable output of the week.
- query: The search query itself.
- COUNT(query) AS query_count: Counts the occurrences of each query within each week_identifier.
- GROUP BY week_identifier, query: Groups the data first by week, then by query, to get counts for each unique query in each week.
- ROW_NUMBER() OVER (PARTITION BY STRFTIME('%Y-%W', timestamp) ORDER BY COUNT(query) DESC) AS rn: This is a window function:
  - PARTITION BY STRFTIME('%Y-%W', timestamp): It divides the data into partitions (groups) for each week.
  - ORDER BY COUNT(query) DESC: Within each week, it orders the queries by their query_count in descending order (most frequent first).
  - ROW_NUMBER(): Assigns a unique rank (1, 2, 3...) to each query within its week, based on the ordering.

2. **Outer Query:**

- SELECT week_start_date, query, query_count: Selects the relevant columns.
- FROM (...) AS weekly_query_counts: Uses the result of the inner query as a subquery.
- WHERE rn <= 3: Filters the results to include only the top 3 ranked queries for each week.
- ORDER BY week_start_date, query_count DESC: Orders the final output by week and then by query count for better readability.

Output:

week_start_date	query	query_count
2025-06-01	data analyst	3
2025-06-01	SQL basics	2
2025-06-01	Python	2
2025-06-08	data analyst	2
2025-06-08	Python	2
2025-06-08	SQL advanced	2

(Note: The week_start_date might vary slightly depending on your database's exact week-starting conventions for the date functions used. I used 'weekday 0' for Sunday in SQLite, which makes 2025-06-01 a Sunday for the first week.)

POWER BI

1. How do you optimize a Power BI dashboard with millions of rows for performance and user experience?

Optimizing a Power BI dashboard with millions of rows is crucial for responsiveness and user satisfaction. This involves a multi-pronged approach covering data modeling, DAX, visuals, and infrastructure.

Here's a detailed breakdown:

A. Data Model Optimization (Most Impactful):

1. **Import Mode vs. DirectQuery/Live Connection:**
    - **Import Mode:** Generally offers the best performance because data is loaded into Power BI's in-memory engine (VertiPaq). This is where most optimizations apply.

- **DirectQuery/Live Connection:** Data remains in the source. Performance heavily depends on the source system's speed and network latency. Optimize the source database queries/views first.
- **Hybrid (Composite Models):** Combine Import and DirectQuery tables. Use DirectQuery for large fact tables where real-time data is critical and Import for smaller, static dimension tables. This is a powerful optimization.

2. **Reduce Cardinality:**
    - **Remove Unnecessary Columns:** Delete columns not used for reporting, filtering, or relationships. This reduces model size significantly.
    - **Reduce Row Count:** Apply filters at the source or during data loading (e.g., only load the last 5 years of data if that's all that's needed).
    - **Optimize Data Types:** Use the smallest appropriate data types (e.g., Whole Number instead of Decimal where possible). Avoid text data types for columns that could be numbers or dates.
    - **Cardinality of Columns:** High-cardinality columns (unique values per row, like timestamps with milliseconds, free-text fields) consume more memory and slow down performance. Reduce precision for dates/times if not needed (e.g., date instead of datetime).
3. **Optimize Relationships:**
    - **Correct Cardinality:** Ensure relationships are set correctly (One-to-Many, One-to-One).
    - **Disable Cross-Filter Direction if not needed:** By default, Power BI often sets "Both" directions. Change to "Single" if filtering only flows one way. "Both" directions can create ambiguity and negatively impact performance.
    - **Avoid Bidirectional Relationships:** Use them sparingly and only when absolutely necessary, as they can lead to performance issues and unexpected filter behavior.
4. **Schema Design (Star Schema/Snowflake Schema):**
    - **Star Schema is King:** Organize your data into fact tables (measures) and dimension tables (attributes). This is the most efficient design for Power BI's VertiPaq engine, enabling fast slicing and dicing.

- **Denormalization:** For dimensions, consider denormalizing (flattening) tables if they are small and frequently joined, to reduce relationship traversal overhead.

5. **Aggregations:**
    - **Pre-aggregate Data:** For very large fact tables, create aggregate tables (e.g., daily sums of sales instead of individual transactions).
    - **Power BI Aggregations:** Power BI allows you to define aggregations within the model, where Power BI automatically redirects queries to a smaller, aggregated table if possible, improving query speed without changing the report logic.

B. DAX Optimization:

1. **Efficient DAX Formulas:**
    - **Avoid Iterators (X-functions) on Large Tables:** Functions like SUMX, AVERAGEX can be slow if used on entire large tables. Where possible, use simpler aggregate functions (SUM, AVERAGE).
    - **Use Variables (VAR):** Store intermediate results in variables to avoid recalculating expressions multiple times. This improves readability and performance.
    - **Minimize Context Transitions:** Context transitions (e.g., using CALCULATE without explicit filters) can be expensive. Understand how DAX calculates.
    - **Use KEEPFILTERS and REMOVEFILTERS strategically:** To control filter context precisely.
    - **Measure Branching:** Break down complex measures into simpler, reusable base measures.
2. **Optimize Calculated Columns:**
    - **Avoid Heavy Calculations in Calculated Columns:** Calculated columns are computed during data refresh and stored in the model, increasing its size. If a calculation can be a measure, make it a measure.
    - **Push Calculations Upstream:** Perform complex data transformations and calculations in Power Query (M language) or even better, in the source database (SQL views, stored procedures).

C. Visual and Report Design Optimization:

1. **Limit Number of Visuals:** Too many visuals on a single page can lead to slower rendering.
2. **Optimize Visual Types:** Some visuals are more performant than others. Table and Matrix visuals with many rows/columns can be slow.
3. **Use Filters and Slicers Effectively:**
    - **Pre-filtered Pages:** Create initial views that are already filtered to a smaller data set.
    - **"Apply" Button for Slicers:** For many slicers, enable the "Apply" button so queries only run after all selections are made.
    - **Hierarchy Slicers:** Use hierarchy slicers if appropriate, as they can sometimes be more efficient than many individual slicers.
4. **Conditional Formatting:** Complex conditional formatting rules can impact performance.
5. **Measure Headers in Matrix/Table:** Avoid placing measures in the "Rows" or "Columns" of a matrix/table, as this significantly increases cardinality and memory usage.

D. Power Query (M Language) Optimization:

1. **Query Folding:** Ensure Power Query steps are "folded back" to the source database as much as possible. This means the transformation happens at the source, reducing the data transferred to Power BI. Check the query plan for folding indicators.
2. **Remove Unnecessary Steps:** Clean up your Power Query steps; remove redundant transformations.
3. **Disable Load:** Disable loading for staging queries or queries that are only used as intermediate steps.

E. Power BI Service and Infrastructure:

1. **Premium Capacity:** For very large datasets and many users, consider Power BI Premium (per user or capacity). This provides dedicated resources, larger memory limits, and features like XMLA endpoint for advanced management.

2. **Scheduled Refresh Optimization:** Use incremental refresh (discussed in the next question).
3. **Monitoring:** Use Power BI Performance Analyzer to identify slow visuals and DAX queries. Use external tools like DAX Studio to analyze and optimize DAX expressions and monitor VertiPaq memory usage.

User Experience Considerations:

- **Clear Navigation:** Use bookmarks, buttons, and drill-throughs for intuitive navigation.
- **Performance Awareness:** Inform users about initial load times for large reports.
- **Clean Design:** Avoid cluttered dashboards. Focus on key metrics.
- **Responsiveness:** Ensure the dashboard adapts well to different screen sizes.

---

2. Explain how incremental data refresh works and why it's important.

How Incremental Data Refresh Works:

Incremental refresh is a Power BI Premium feature (also available with Power BI Pro for datasets up to 1GB, but typically used for larger datasets) that allows Power BI to efficiently refresh large datasets by **only loading new or updated data**, instead of reprocessing the entire dataset with every refresh.

Here's the mechanism:

1. **Defining the Policy:** You configure an incremental refresh policy in Power BI Desktop for specific tables (usually large fact tables). This policy defines:
    - **Date/Time Column:** A column in your table that Power BI can use to identify new or changed rows (e.g., OrderDate, LastModifiedDate). This column *must* be of Date/Time data type.
        - **Range Start (RangeStart) and Range End (RangeEnd) Parameters:** These are two reserved DateTime parameters that Power BI automatically generates and passes to your data source query. They define the "window" of data to be refreshed.

- **Archive Period:** How many past years/months/days of data you want to keep in the Power BI model. This data will be loaded once and then not refreshed.
- **Refresh Period:** How many recent years/months/days of data should be refreshed *incrementally* with each refresh operation. This is the "sliding window" for new/updated data.

2. **Partitioning:** When you publish the report to the Power BI Service, Power BI dynamically creates partitions for the table based on your incremental refresh policy:
    - **Historical Partitions:** For the "Archive Period," Power BI creates partitions that contain historical data. This data is loaded once and then *not refreshed* in subsequent refreshes.
    - **Incremental Refresh Partition(s):** For the "Refresh Period," Power BI creates one or more partitions. Only these partitions are refreshed in subsequent refresh cycles.
    - **Real-time Partition (Optional):** If you configure a DirectQuery partition, this can fetch the latest data directly from the source for the freshest view.
3. **Refresh Process:**
    - When a scheduled refresh runs, Power BI calculates the RangeStart and RangeEnd values based on the current refresh time and your policy.
    - It then issues a query to your data source using these parameters, fetching only the data within the defined refresh window.
    - This new/updated data is loaded into the incremental partition(s), and older incremental partitions might be rolled into the archive partitions or removed, as the window slides.

Why It's Important:

Incremental refresh is vital for several reasons, especially with large datasets:

1. **Faster Refreshes:** This is the primary benefit. Instead of reloading millions or billions of rows, Power BI only fetches tens or hundreds of thousands, dramatically cutting down refresh times from hours to minutes or seconds.
2. **Reduced Resource Consumption:**

- **Less Memory:** Fewer resources are consumed on the Power BI service side during refresh because less data is being processed.
- **Less Network Bandwidth:** Less data needs to be transferred from the source system to Power BI.
- **Less Load on Source System:** The source database experiences less strain because queries are filtered to a smaller range, reducing query execution time and resource usage on the database server.

3. **Higher Refresh Frequency:** Because refreshes are faster and less resource-intensive, you can schedule them more frequently (e.g., hourly instead of daily), providing users with more up-to-date data.
4. **Increased Reliability:** Shorter refresh windows reduce the chances of refresh failures due to network timeouts, source system issues, or hitting refresh limits.
5. **Scalability:** Enables Power BI to handle datasets that would otherwise be too large or too slow to refresh regularly, making it viable for enterprise-level reporting solutions.
6. **Better User Experience:** Users get access to fresh data faster, improving their decision-making capabilities.

---

3. What's the difference between calculated columns and measures in Power BI, and when would you use each?

Calculated columns and measures are both powerful DAX (Data Analysis Expressions) constructs in Power BI, but they serve fundamentally different purposes and have distinct characteristics.

Feature	Calculated Column	Measure
Calculation Time	**During Data Refresh (load time)**	**At Query Time (when used in a visual)**
Storage	**Stored in the Data Model** (adds to model size)	**Not Stored** (calculated on the fly)

Context	**Row Context** (can refer to values in the same row)	**Filter Context** (and Row Context within iterators)
Output	**A new column** added to the table	**A single scalar value** (number, text, date)
Impact on Size	**Increases PBIX file size & memory usage**	**Minimal impact on PBIX file size**
Aggregation	Can be aggregated like any other column	**Always aggregated** (implicit or explicit)
Visibility	Appears as a column in the Fields pane	Appears as a measure in the Fields pane

When to Use Each:

Use Calculated Columns When:

1. **You need to create a new categorical attribute:**
    - Full Name = [FirstName] & " " & [LastName]
    - Age Group = IF([Age] < 18, "Child", IF([Age] < 65, "Adult", "Senior"))
2. **You need to perform row-level calculations that will be used for slicing, dicing, or filtering:**
    - Profit Margin % = ([Sales] - [Cost]) / [Sales] (if you need to filter or group by this margin on a row-by-row basis).
    - Fiscal Quarter = "Q" & ROUNDUP(MONTH([Date])/3,0)
3. **You need to define relationships:** Calculated columns can be used as the key for relationships if a direct column from your source isn't suitable. (However, it's often better to handle this in Power Query if possible).
4. **You are creating a static value for each row that doesn't change based on filters applied in the report.**

Use Measures When:

1. **You need to perform aggregations or calculations that respond dynamically to filters and slicers applied in the report:**
    - Total Sales = SUM(FactSales[SalesAmount])

    - Average Order Value = DIVIDE([Total Sales], COUNTROWS(FactSales))
    - Sales YTD = TOTALYTD([Total Sales], 'Date'[Date])
2. **You need to calculate a ratio, percentage, or difference that changes based on the selected context:**
    - % of Total Sales = DIVIDE([Total Sales], CALCULATE([Total Sales], ALL(Product[Category])))
3. **You want to perform complex time-intelligence calculations:**
    - Sales Last Year = CALCULATE([Total Sales], SAMEPERIODLASTYEAR('Date'[Date]))
4. **You want to minimize the model size and optimize performance:** Since measures are calculated on the fly and not stored, they are generally preferred for performance over calculated columns, especially for large datasets.
5. **Your calculation logic changes based on the filter context of the visual.**

General Rule of Thumb:

- **If you can do it in Power Query (M Language), do it there.** This pushes the calculation closest to the source, often leveraging query folding.
- **If it's a row-level calculation that defines a characteristic of that row and you need to slice/dice by it, use a Calculated Column.**
- **For all other aggregations and dynamic calculations that react to user interaction, use a Measure.**

Choosing correctly between calculated columns and measures is fundamental for building efficient, performant, and maintainable Power BI models.

---

4. How would you implement cross-report drillthrough in Power BI for navigating between detailed reports?

Cross-report drillthrough in Power BI allows users to jump from a summary visual in one report to a more detailed report page in a *different* report, passing the filter context along. This is incredibly powerful for creating a guided analytical experience across a suite of related reports.

Here's how you would implement it:

Scenario:

- **Source Report (Summary):** Sales Overview Dashboard.pbix with a chart showing "Sales by Region."
- **Target Report (Detail):** Regional Sales Details.pbix with a table showing individual sales transactions for a specific region.

Steps to Implement Cross-Report Drillthrough:

1. Prepare the Target Report (Regional Sales Details.pbix):

- **Create the Detail Page:** Open Regional Sales Details.pbix. Create a new page dedicated to displaying the detailed information (e.g., "Sales Transactions").
- **Add Drillthrough Fields:**
    - In the "Fields" pane for your detail page, locate the fields that will serve as the drillthrough filters (e.g., Region Name, Product Category). These are the fields that will be passed from the source report.
    - Drag these fields into the "Drill through" section of the "Visualizations" pane.
    - **Crucial:** Ensure that the data types and column names of these drillthrough fields are **identical** in both the source and target reports. If they aren't, the drillthrough won't work correctly.
- **Set "Keep all filters":** By default, "Keep all filters" is usually on. This ensures that any other filters applied to the source visual (e.g., date range, product type) are also passed to the target report. You can turn it off if you only want to pass the drillthrough fields explicitly.
- **Add Visuals:** Add the detailed visuals (e.g., a table showing Date, Product, Customer, Sales Amount) to this drillthrough page.
- **Add a Back Button (Optional but Recommended):** Power BI automatically adds a "back" button for *intra-report* drillthrough. For cross-report, you usually add a custom button (Insert > Buttons > Back) and configure its action to "Back" or a specific bookmark if you have complex navigation. This allows users to easily return to the summary report.
- **Publish the Target Report:** Publish Regional Sales Details.pbix to a Power BI workspace in the Power BI Service. Make sure it's in a workspace that both you and your users have access to.

2. Prepare the Source Report (Sales Overview Dashboard.pbix):

- **Ensure Data Model Consistency:** Verify that the drillthrough fields (e.g., Region Name, Product Category) exist in the source report's data model and have the same name and data type as in the target report.
- **Select the Source Visual:** Choose the visual from which you want to initiate the drillthrough (e.g., your "Sales by Region" bar chart).
- **Configure Drillthrough Type:**
    - Go to the "Format" pane for the selected visual.
    - Under the "Drill through" card, ensure "Cross-report" is enabled.
- **Choose the Target Report:**
    - In the "Drill through" card, you'll see a dropdown list of available reports in your workspace that have drillthrough pages configured.
    - Select Regional Sales Details from this list.
- **Publish the Source Report:** Publish Sales Overview Dashboard.pbix to the *same* Power BI workspace as the target report. This is essential for cross-report drillthrough to work.

3. User Experience in Power BI Service:

- **Navigation:** When a user views the Sales Overview Dashboard report in the Power BI Service, they can right-click on a data point in the configured source visual (e.g., a bar representing "East" region sales).
- **Drillthrough Option:** A context menu will appear, and they will see an option like "Drill through" -> "Regional Sales Details."
- **Context Passing:** Clicking this option will open the Regional Sales Details report, automatically navigating to the specified drillthrough page. Critically, the Region Name (e.g., "East") and any other filters from the source visual will be applied to the Regional Sales Details report, showing only the transactions for the "East" region.

Key Considerations for Cross-Report Drillthrough:

- **Workspace:** Both reports **must** be published to the *same Power BI workspace*. This is a fundamental requirement.

- **Field Matching:** Column names and data types of the drillthrough fields must be an exact match across both reports. Case sensitivity can also be an issue.
- **Data Models:** While the column names must match, the underlying data models don't have to be identical. The source report only needs the columns to pass as filters, and the target report needs those columns to filter its detailed data.
- **User Permissions:** Users must have at least "Viewer" access to both the source and target reports in the Power BI Service.
- **Security (Row-Level Security - RLS):** RLS applied in the target report will respect the user's RLS role, even if the drillthrough filters pass data that the user wouldn't normally see. The RLS will act as an additional filter layer.
- **Performance:** Be mindful of the performance of the target report, especially if it's loading large volumes of detailed data. Optimize it as per the first question.
- **Clarity:** Make it clear to users that a drillthrough option exists (e.g., by adding text instructions or using appropriate visual cues).

Cross-report drillthrough is an advanced feature that significantly enhances the navigability and analytical depth of your Power BI solutions by allowing you to break down complex business problems into manageable, linked reports.

PYTHON

---

1. Write a Python function to detect outliers in a dataset using the IQR method.

The Interquartile Range (IQR) method is a robust way to identify outliers, as it's less sensitive to extreme values than methods based on the mean and standard deviation.

Concept:

1. Calculate the First Quartile (Q1) - 25th percentile.
2. Calculate the Third Quartile (Q3) - 75th percentile.
3. Calculate the Interquartile Range (IQR) = Q3 - Q1.
4. Define the Lower Bound: Q1 - 1.5 * IQR

5. Define the Upper Bound: Q3 + 1.5 * IQR
6. Any data point below the Lower Bound or above the Upper Bound is considered an outlier.

Python Function:

```
import numpy as np

import pandas as pd


def detect_iqr_outliers(data, column):
  """
  Detects outliers in a specified column of a pandas DataFrame using the IQR method.


  Args:
    data (pd.DataFrame): The input DataFrame.
    column (str): The name of the column to check for outliers.


  Returns:
    pd.DataFrame: A DataFrame containing the detected outliers.
    dict: A dictionary containing the outlier bounds (lower_bound, upper_bound).
  """
  if column not in data.columns:
    raise ValueError(f"Column '{column}' not found in the DataFrame.")


  # Ensure the column is numeric
  if not pd.api.types.is_numeric_dtype(data[column]):
    raise TypeError(f"Column '{column}' is not numeric. IQR method requires numeric data.")
```

```
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    outliers = data[(data[column] < lower_bound) | (data[column] > upper_bound)]

    bounds = {'lower_bound': lower_bound, 'upper_bound': upper_bound}

    return outliers, bounds

# --- Example Usage ---
if __name__ == "__main__":
    # Sample Dataset
    np.random.seed(42)
    data = {
        'id': range(1, 21),
        'value': np.random.normal(loc=50, scale=10, size=20)
    }
    df = pd.DataFrame(data)

    # Introduce some outliers
    df.loc[3, 'value'] = 150  # Outlier
    df.loc[15, 'value'] = 5   # Outlier
```

```
df.loc[19, 'value'] = 120 # Outlier

print("Original DataFrame:")
print(df)
print("\n" + "="*30 + "\n")

# Detect outliers in the 'value' column
outliers_df, bounds = detect_iqr_outliers(df, 'value')

print(f"Calculated Bounds for 'value':")
print(f"  Q1: {df['value'].quantile(0.25):.2f}")
print(f"  Q3: {df['value'].quantile(0.75):.2f}")
print(f"  IQR: {(df['value'].quantile(0.75) - df['value'].quantile(0.25)):.2f}")
print(f"  Lower Bound: {bounds['lower_bound']:.2f}")
print(f"  Upper Bound: {bounds['upper_bound']:.2f}")
print("\n" + "="*30 + "\n")

print("Detected Outliers:")
if not outliers_df.empty:
    print(outliers_df)
else:
    print("No outliers detected.")

print("\n" + "="*30 + "\n")

# Example with no outliers (after removing manual outliers)
```

```
df_clean = df.drop([3, 15, 19]).copy()
print("DataFrame without extreme manual outliers:")
print(df_clean)
outliers_clean, bounds_clean = detect_iqr_outliers(df_clean, 'value')
print("\nDetected Outliers (cleaned data):")
if not outliers_clean.empty:
    print(outliers_clean)
else:
    print("No outliers detected.")
```

Explanation:

1. **Import numpy and pandas**: Essential libraries for numerical operations and DataFrame manipulation.
2. **detect_iqr_outliers(data, column) function:**
    - **Input Validation:** Checks if the column exists in the DataFrame and if it's a numeric data type. This makes the function more robust.
    - **Calculate Quartiles:** data[column].quantile(0.25) and data[column].quantile(0.75) directly compute Q1 and Q3 using pandas' built-in quantile method.
    - **Calculate IQR:** IQR = Q3 - Q1.
    - **Calculate Bounds:** lower_bound = Q1 - 1.5 * IQR and upper_bound = Q3 + 1.5 * IQR. The 1.5 factor is a commonly used convention.
    - **Identify Outliers:** data[(data[column] < lower_bound) | (data[column] > upper_bound)] uses boolean indexing to filter the DataFrame and select rows where the specified column's value falls outside the calculated bounds.
    - **Return Values:** The function returns two things: a DataFrame containing the outlier rows and a dictionary with the calculated bounds. This allows the caller to not only see the outliers but also understand the thresholds used.
3. **Example Usage (if __name__ == "__main__":)**
    - A sample DataFrame df is created with normally distributed data.

- Specific rows are then manually modified to introduce clear outliers for demonstration.
- The detect_iqr_outliers function is called, and its output (outlier DataFrame and bounds) is printed.
- A second example demonstrates the function on a cleaner dataset where no "extreme" outliers are introduced to show a case with no detected outliers.

---

2. You have two DataFrames: clicks and installs. Merge them and calculate the install-to-click ratio per campaign.

This question tests your knowledge of DataFrame merging, aggregation, and basic arithmetic operations in pandas.

Concept:

1. Merge the clicks and installs DataFrames. A campaign_id is the natural key for merging. A "left merge" (or "left join") is appropriate if we want to retain all click data and bring in matching install data.
2. After merging, group the data by campaign_id.
3. For each campaign, count the total clicks and total installs.
4. Calculate the ratio: total_installs / total_clicks. Handle division by zero.

Python Code:

Python

```
import pandas as pd


def calculate_install_to_click_ratio(clicks_df, installs_df):
    """
    Merges clicks and installs DataFrames and calculates the install-to-click ratio per campaign.

    Args:
```

```
    clicks_df (pd.DataFrame): DataFrame with click data, expected columns: ['campaign_id', 'click_id', ...].
    installs_df (pd.DataFrame): DataFrame with install data, expected columns: ['campaign_id', 'install_id', ...].

  Returns:
    pd.DataFrame: A DataFrame with 'campaign_id', 'total_clicks', 'total_installs', and 'install_to_click_ratio'.
  """
  # 1. Merge the two DataFrames
  # We'll perform a left merge from clicks_df to installs_df to ensure all clicks are considered.
  # If a click doesn't have a corresponding install, the install_id will be NaN.
  # We only need campaign_id and a count for clicks, and campaign_id and a count for installs.

  # Aggregate clicks and installs first by campaign_id to reduce merge size
  # This is often more efficient for large datasets than merging raw rows
  clicks_agg = clicks_df.groupby('campaign_id').size().reset_index(name='total_clicks')
  installs_agg = installs_df.groupby('campaign_id').size().reset_index(name='total_installs')

  # Merge the aggregated data
  merged_df = pd.merge(
    clicks_agg,
    installs_agg,
    on='campaign_id',
    how='left' # Use left join to keep all campaigns that had clicks
```

```
    )

    # 2. Handle campaigns with no installs (NaN in total_installs after left join)
    merged_df['total_installs'] = merged_df['total_installs'].fillna(0).astype(int)

    # 3. Calculate the install-to-click ratio
    # Handle division by zero by replacing it with 0 or NaN, depending on desired behavior.
    # Using np.where or a simple check for 0 clicks.
    merged_df['install_to_click_ratio'] = merged_df.apply(
        lambda row: row['total_installs'] / row['total_clicks'] if row['total_clicks'] > 0 else 0,
        axis=1
    )

    # Or using np.where for potentially better performance on large datasets
    # merged_df['install_to_click_ratio'] = np.where(
    #     merged_df['total_clicks'] > 0,
    #     merged_df['total_installs'] / merged_df['total_clicks'],
    #     0 # Set to 0 if no clicks, or np.nan if you prefer
    # )

    return merged_df[['campaign_id', 'total_clicks', 'total_installs', 'install_to_click_ratio']]

# --- Example Usage ---
if __name__ == "__main__":
    # Sample Clicks DataFrame
    clicks_data = {
```

```
    'campaign_id': ['C1', 'C1', 'C2', 'C3', 'C1', 'C2', 'C4', 'C3', 'C1'],
    'click_id': range(101, 110),
    'timestamp': pd.to_datetime(['2023-01-01', '2023-01-01', '2023-01-02', '2023-01-02', '2023-01-03', '2023-01-03', '2023-01-04', '2023-01-05', '2023-01-05'])
  }
  clicks_df = pd.DataFrame(clicks_data)

  # Sample Installs DataFrame
  installs_data = {
    'campaign_id': ['C1', 'C2', 'C1', 'C3', 'C1', 'C2'],
    'install_id': range(201, 207),
    'timestamp': pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-03', '2023-01-05', '2023-01-05'])
  }
  installs_df = pd.DataFrame(installs_data)

  print("Clicks DataFrame:")
  print(clicks_df)
  print("\nInstalls DataFrame:")
  print(installs_df)
  print("\n" + "="*30 + "\n")

  # Calculate the ratio
  ratio_df = calculate_install_to_click_ratio(clicks_df, installs_df)

  print("Install-to-Click Ratio per Campaign:")
  print(ratio_df)
```

```
print("\n" + "="*30 + "\n")


# Example with a campaign with clicks but no installs
clicks_data_2 = {
   'campaign_id': ['C1', 'C1', 'C5', 'C5'], # C5 has clicks but no installs
   'click_id': [1,2,3,4]
}
installs_data_2 = {
   'campaign_id': ['C1', 'C1'],
   'install_id': [10,11]
}
clicks_df_2 = pd.DataFrame(clicks_data_2)
installs_df_2 = pd.DataFrame(installs_data_2)


ratio_df_2 = calculate_install_to_click_ratio(clicks_df_2, installs_df_2)
print("Install-to-Click Ratio (with campaign with no installs):")
print(ratio_df_2)
```

Explanation:

1. **calculate_install_to_click_ratio(clicks_df, installs_df) function:**
    - **Aggregation Before Merge (Optimization):** Instead of merging the full clicks_df and installs_df (which could be very large), we first aggregate them by campaign_id using groupby('campaign_id').size().reset_index(name='total_clicks'). This creates smaller DataFrames (clicks_agg and installs_agg) containing just the campaign_id and the total count for that campaign. Merging these smaller DataFrames is much more efficient.

- **Merge:** pd.merge(clicks_agg, installs_agg, on='campaign_id', how='left') performs a left join. This ensures that every campaign_id that had clicks is included in the final result, even if it had zero installs.
- **Handle Missing Installs:** merged_df['total_installs'].fillna(0).astype(int) replaces NaN values (which occur for campaigns with clicks but no installs) with 0 and converts the column to integer type.
- **Calculate Ratio:**
  - merged_df.apply(lambda row: row['total_installs'] / row['total_clicks'] if row['total_clicks'] > 0 else 0, axis=1): This calculates the ratio row by row. It includes a if row['total_clicks'] > 0 else 0 check to prevent ZeroDivisionError. If total_clicks is 0, the ratio is set to 0. You could also set it to np.nan if that's more appropriate for your use case.
  - The commented-out np.where alternative is generally more performant for very large DataFrames as it's vectorized.
- **Return Value:** The function returns a DataFrame with the campaign_id, total clicks, total installs, and the calculated ratio.

2. **Example Usage:**
   - Sample clicks_df and installs_df are created to demonstrate the function.
   - The results are printed, showing the counts and the calculated ratios for each campaign.
   - A second example shows how the function handles a campaign that has clicks but no installs, correctly assigning an install count of 0 and a ratio of 0.

---

3. How would you use Python to automate a weekly reporting task that includes querying data, generating a chart, and emailing it?

Automating this type of task is a classic use case for Python in data analysis. It involves several key libraries and concepts.

Overall Workflow:

1. **Configuration:** Store sensitive information (database credentials, email passwords, recipient lists) securely.
2. **Data Querying:** Connect to a database (e.g., SQL, PostgreSQL, etc.) and fetch the necessary data.
3. **Data Processing:** Use pandas to clean, transform, and aggregate the queried data.
4. **Chart Generation:** Use a plotting library (e.g., Matplotlib, Seaborn, Plotly) to create a visual representation of the data.
5. **Emailing:** Use Python's smtplib and email.mime modules to send the report as an email with the chart attached.
6. **Scheduling:** Use operating system tools (Cron on Linux/macOS, Task Scheduler on Windows) or a Python scheduler (e.g., APScheduler) to run the script automatically every week.

Python Libraries Used:

- **pandas**: For data manipulation.
- **sqlalchemy / psycopg2 / mysql-connector-python**: For connecting to databases (example below uses a generic SQL connection via sqlalchemy).
- **matplotlib.pyplot / seaborn**: For plotting.
- **smtplib**: For sending emails via SMTP.
- **email.mime.multipart / email.mime.text / email.mime.image**: For constructing email messages with attachments.
- **configparser / python-dotenv**: For managing configurations and credentials (recommended for security).

Example Code Structure:

```
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import sqlalchemy # or specific db driver like psycopg2

import smtplib

from email.mime.multipart import MIMEMultipart
```

```
from email.mime.text import MIMEText

from email.mime.image import MIMEImage

import datetime

import os

import configparser # For storing credentials securely (basic example)


# --- Configuration (Highly Recommended to use .env or a dedicated config file) ---

# Create a config.ini file:

# [DATABASE]

# DB_TYPE = postgresql

# DB_USER = your_db_user

# DB_PASSWORD = your_db_password

# DB_HOST = localhost

# DB_PORT = 5432

# DB_NAME = your_database


# [EMAIL]

# SENDER_EMAIL = your_email@example.com

# SENDER_PASSWORD = your_email_app_password # Use app passwords for
Gmail/Outlook

# RECEIVER_EMAIL = recipient_email@example.com

# SMTP_SERVER = smtp.gmail.com

# SMTP_PORT = 587 # or 465 for SSL


config = configparser.ConfigParser()

config.read('config.ini') # Make sure config.ini is in the same directory or provide full path
```

```
# Database Credentials
DB_TYPE = config['DATABASE']['DB_TYPE']
DB_USER = config['DATABASE']['DB_USER']
DB_PASSWORD = config['DATABASE']['DB_PASSWORD']
DB_HOST = config['DATABASE']['DB_HOST']
DB_PORT = config['DATABASE']['DB_PORT']
DB_NAME = config['DATABASE']['DB_NAME']

# Email Credentials
SENDER_EMAIL = config['EMAIL']['SENDER_EMAIL']
SENDER_PASSWORD = config['EMAIL']['SENDER_PASSWORD']
RECEIVER_EMAIL = config['EMAIL']['RECEIVER_EMAIL']
SMTP_SERVER = config['EMAIL']['SMTP_SERVER']
SMTP_PORT = int(config['EMAIL']['SMTP_PORT'])

def get_data_from_db():
    """
    Connects to the database and fetches the required data.
    """
    # Example for PostgreSQL connection string
    # For MySQL: 'mysql+mysqlconnector://user:password@host:port/database'
    # For SQLite: 'sqlite:///your_database.db'
    db_connection_str =
f'{DB_TYPE}://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}'
```

```
try:
    engine = sqlalchemy.create_engine(db_connection_str)
    # Query data for the last week
    # Adjust your SQL query based on your table structure and desired date filtering
    query = """
    SELECT
        campaign_name,
        SUM(clicks) AS total_clicks,
        SUM(installs) AS total_installs
    FROM
        your_ad_data_table
    WHERE
        date_column >= CURRENT_DATE - INTERVAL '7 days' -- Example for PostgreSQL
        -- date_column >= DATE('now', '-7 days') -- Example for SQLite
        -- date_column >= DATEADD(day, -7, GETDATE()) -- Example for SQL Server
    GROUP BY
        campaign_name
    ORDER BY
        total_clicks DESC;
    """
    df = pd.read_sql(query, engine)
    print("Data queried successfully.")
    return df
except Exception as e:
    print(f"Error querying data: {e}")
    return pd.DataFrame() # Return empty DataFrame on error
```

```
def generate_chart(data_df, chart_filename="campaign_performance.png"):
    """
    Generates a bar chart from the data and saves it as an image.
    """
    if data_df.empty:
        print("No data to generate chart.")
        return None

    # Calculate ratio for plotting
    data_df['install_to_click_ratio'] = data_df.apply(
        lambda row: row['total_installs'] / row['total_clicks'] if row['total_clicks'] > 0 else 0,
        axis=1
    )

    plt.figure(figsize=(12, 7))
    sns.barplot(x='campaign_name', y='total_clicks', data=data_df, color='skyblue',
label='Total Clicks')
    sns.barplot(x='campaign_name', y='total_installs', data=data_df, color='lightcoral',
label='Total Installs')

    # Optional: Add a line for ratio if meaningful on the same chart, or create a separate one
    # plt.twinx()
    # sns.lineplot(x='campaign_name', y='install_to_click_ratio', data=data_df, color='green',
marker='o', label='Install-to-Click Ratio')
    # plt.ylabel("Ratio")
```

```
    plt.title(f"Weekly Campaign Performance - {datetime.date.today().strftime('%Y-%m-%d')}")
    plt.xlabel("Campaign Name")
    plt.ylabel("Count")
    plt.xticks(rotation=45, ha='right')
    plt.legend()
    plt.tight_layout()

    plt.savefig(chart_filename)
    plt.close() # Close the plot to free memory
    print(f"Chart saved as {chart_filename}")
    return chart_filename

def send_email_report(chart_path, sender_email, sender_password, receiver_email, smtp_server, smtp_port):
    """
    Sends the generated chart as an email attachment.
    """
    if not chart_path or not os.path.exists(chart_path):
        print("Chart file not found, cannot send email.")
        return

    msg = MIMEMultipart()
    msg['From'] = sender_email
    msg['To'] = receiver_email
    msg['Subject'] = f"Weekly Campaign Performance Report - {datetime.date.today().strftime('%Y-%m-%d')}"
```

```
    # Email body
    body = """
    <html>
    <body>
      <p>Hi Team,</p>
      <p>Please find attached the weekly campaign performance report.</p>
      <p>This report covers data for the last 7 days ending today.</p>
      <p>Best regards,<br>Your Reporting Automation Bot</p>
      <img src="cid:my_chart_image">
    </body>
    </html>
    """
    msg.attach(MIMEText(body, 'html'))

    # Attach the chart image
    with open(chart_path, 'rb') as fp:
      img = MIMEMultipart('image', 'png') # Correct MIME type for PNG image
      img.add_header('Content-Disposition', 'attachment',
filename=os.path.basename(chart_path))
      img.add_header('Content-ID', '<my_chart_image>') # CID for embedding in HTML
      img.add_header('X-Attachment-Id', 'my_chart_image')
      img.set_payload(fp.read())
      from email import encoders
      encoders.encode_base64(img)
      msg.attach(img)
```

```
    try:
        with smtplib.SMTP(smtp_server, smtp_port) as server:
            server.starttls() # Secure the connection
            server.login(sender_email, sender_password)
            server.send_message(msg)
        print("Email sent successfully!")
    except Exception as e:
        print(f"Error sending email: {e}")

# --- Main Automation Function ---
def automate_weekly_report():
    print("Starting weekly report automation...")

    # 1. Query Data
    data_df = get_data_from_db()

    if data_df.empty:
        print("No data retrieved. Exiting automation.")
        return

    # 2. Generate Chart
    chart_filename = generate_chart(data_df)

    # 3. Email Report
    if chart_filename:
```

```
        send_email_report(chart_filename, SENDER_EMAIL, SENDER_PASSWORD,
RECEIVER_EMAIL, SMTP_SERVER, SMTP_PORT)
        # Clean up the generated chart file
        os.remove(chart_filename)
        print(f"Cleaned up {chart_filename}")
    else:
        print("Chart generation failed, skipping email.")

    print("Weekly report automation finished.")

if __name__ == "__main__":
    automate_weekly_report()

    # To schedule this weekly:
    # On Linux/macOS:
    # 1. Save the script as e.g., `weekly_report.py`
    # 2. Create a cron job: `crontab -e`
    # 3. Add a line like (e.g., every Monday at 9 AM):
    #    `0 9 * * 1 /usr/bin/python3 /path/to/your/weekly_report.py`
    # On Windows:
    # Use Task Scheduler to create a task that runs the Python script weekly.
    # Alternatively, for more complex Python-native scheduling, use APScheduler.
```

Explanation:

1. **Configuration (configparser):**
    - **config.ini:** A separate file (config.ini) is used to store database credentials and email settings. This is *much better* than hardcoding them directly in the

script, as it keeps sensitive info separate from your code and allows easy modification.

- **Security Note:** For production systems, consider more robust secrets management solutions like environment variables, Azure Key Vault, AWS Secrets Manager, or Google Secret Manager. An app-specific password (if your email provider supports it, like Gmail) is better than your main password.

2. **get_data_from_db():**
    - **Database Connection:** Uses sqlalchemy.create_engine to establish a connection to your database. You'd replace the DB_TYPE and connection string parts to match your specific database (e.g., mysql, postgresql, sqlite). You might need to install specific drivers like psycopg2 for PostgreSQL or mysql-connector-python for MySQL.
    - **SQL Query:** Contains a placeholder SQL query to fetch campaign data for the last 7 days. **You'll need to adapt this query** to your actual table names, column names, and date filtering syntax for your database.
    - **pd.read_sql():** Reads the results of the SQL query directly into a pandas DataFrame.
    - **Error Handling:** Includes a try-except block to catch database connection or query errors.
3. **generate_chart():**
    - **Data Preparation:** Calculates the install_to_click_ratio within this function, as it's specific to the report's visual.
    - **Plotting:** Uses matplotlib.pyplot and seaborn to create a visually appealing bar chart.
    - **Customization:** You can customize colors, titles, labels, rotations, and add a legend for clarity.
    - **Saving Chart:** plt.savefig(chart_filename) saves the generated chart as a PNG image. plt.close() is important to release memory after saving.
    - **Return:** Returns the filename of the saved chart.
4. **send_email_report():**

- **MIMEMultipart:** Creates a multipart email message, allowing you to combine text (HTML) and attachments.
- **MIMEText:** Sets the HTML body of the email.
- **MIMEImage:** Attaches the generated chart as an image. cid:my_chart_image and Content-ID are used to embed the image directly within the HTML body, so it appears inline.
- **SMTP Connection:**
  - smtplib.SMTP(smtp_server, smtp_port): Connects to the SMTP server (e.g., smtp.gmail.com for Gmail).
  - server.starttls(): Initiates Transport Layer Security (TLS) for a secure connection.
  - server.login(sender_email, sender_password): Logs into your email account.
  - server.send_message(msg): Sends the constructed email.
- **Error Handling:** Includes a try-except block for email sending errors.

5. **automate_weekly_report() (Main Function):**
   - Orchestrates the entire process: calls get_data_from_db(), then generate_chart(), and finally send_email_report().
   - Includes basic checks and messages for user feedback.
   - **Cleanup:** os.remove(chart_filename) deletes the temporary chart image file after the email is sent, keeping your system clean.
6. **Scheduling (if __name__ == "__main__":)**
   - The if __name__ == "__main__": block ensures automate_weekly_report() runs when the script is executed directly.
   - Comments provide guidance on how to schedule this script using cron (for Linux/macOS) or Windows Task Scheduler. For more advanced Python-native scheduling within a long-running application, APScheduler is a good choice.