Section: Calculated Columns vs Measures in Power BI

🎯 Objective:

Understand the key differences between **calculated columns** and **measures**, how and when to use each, and practice creating them using DAX (Data Analysis Expressions).

□ Key Concepts:

📌 What is a Calculated Column?

- A **calculated column** is a **new physical column** added to a table using a formula (DAX).
- It is computed **row-by-row** and **stored in memory**.
- Useful when you need **row-level data**, such as custom identifiers or derived fields.

📌 What is a Measure?

- A **measure** is a **dynamic calculation** created using DAX.
- It is **not stored in the table** but is calculated **only when needed**, i.e., during visualizations.
- Ideal for **aggregated values**, like totals, averages, or KPIs.
- **Better performance** over calculated columns in large datasets.

🛠□ Practice Steps: Creating Calculated Columns

✅ Goal 1: Pull Product Price into Sales Table

1. Navigate to **Data View** in Power BI.
2. **Right-click** on the `Sales` table → select **New column**.
3. In the formula bar, type:

```
DAX

Price = RELATED(Products[Price])
```

4. Format the column to **2 decimal places** from the **Modeling tab**.

📝 *This uses the relationship via `Product ID` to fetch product price.*

✅ Goal 2: Calculate Revenue (Quantity × Price)

1. In the same way, create another column in the `Sales` table:

DAX

```
Revenue = Sales[Quantity] * Sales[Price]
```

2. Format as **Currency**.

📝 *This creates a revenue value for each row in the table.*

🛠️ Practice Steps: Creating Measures

✅ Goal 3: Create a Simple Total Measure

1. Switch to **Report View**.
2. **Right-click** on the `Sales` table → choose **New measure**.
3. In the formula bar, type:

DAX

```
Total Quantity = SUM(Sales[Quantity])
```

4. Add it to a visual (e.g., Card or Table) to see it in action.

✅ Goal 4: Create Revenue as a Measure Using `SUMX`

1. Again, create a new measure in the `Sales` table:

DAX

```
Revenue Measure = SUMX(Sales, Sales[Price] * Sales[Quantity])
```

2. Add to a visual, optionally along with `LocationID` or other grouping fields.

📝 *`SUMX` evaluates the expression row by row and sums the results.*

🔍 Summary:

Feature	Calculated Column	Measure
Storage	Stored physically in table	Virtual (calculated at runtime)
Evaluation Context	Row-by-row	Depends on visual context
Best For	Row-level computations	Aggregated values (KPIs, totals, averages)

Feature	Calculated Column	Measure
Performance Impact	Higher memory usage	Better performance
Examples	`Revenue = Quantity * Price`	`SUMX(Sales, Quantity * Price)`

□ Practice Challenge:

Try recreating both **revenue** as a calculated column and as a measure. Compare their behavior:

- Can both be used in visuals?
- Are both visible in Data view?
- What happens when you slice by `LocationID`?

◆ Section 2: Creating a Date Table Using `CALENDARAUTO` in Power BI

🎯 Objective:

Learn how to quickly create a **comprehensive date table** using the `CALENDARAUTO()` DAX function and understand how fiscal years affect the generated date range.

□ Key Concepts:

📌 Why a Date Table?

- A **date table** is essential for time-based analysis (e.g., YoY growth, trends).
- Helps in creating **time intelligence measures** like YTD, QTD, MTD, etc.
- Should include a **continuous date range** with **no missing dates**.

🛠□ Method 1: Auto-Generating a Date Table with `CALENDARAUTO()`

✅ Step-by-Step Instructions:

1. Go to **Modeling** tab in Power BI.
2. Click on **New Table**.
3. In the formula bar, write:

DAX

```
DateTable = CALENDARAUTO()
```

4. Press **Enter**. A new table named `DateTable` will be created.

📌 What Does `CALENDARAUTO()` Do?

- Automatically scans all date columns in the model.
- Identifies the **earliest** and **latest** dates across all tables.
- Returns a continuous range of dates from the **start of the fiscal year** containing the earliest date to the **end of the fiscal year** containing the latest date.

Example:

- Earliest date in model: `May 1, 2017`
- Fiscal year default: Starts in January
- Output: Dates from `January 1, 2017` to `December 31, 2020`

Fiscal Year Customization

By default, fiscal year ends in **December (12)**. But you can specify another end month.

🛠 Practice:

Create another version of the date table with fiscal year ending in **March (3)**:

```
DAX

DateTable_FY_March = CALENDARAUTO(3)
```

🔍 This will return:

- Start: `April 1` of the earliest year
- End: `March 31` of the latest year

👀 Visualization Tip:

- Add the new `DateTable` to a **Table Visual**.
- Power BI auto-applies **Date Hierarchy** (Year → Quarter → Month → Day).
- To view raw dates:
  - Click on the field in the visual
  - Select **"Date"** instead of **"Date Hierarchy"**

□ Summary:

Feature	`CALENDARAUTO()`
Output	Table with a single `Date` column
Date Source	All date fields in your model
Default Fiscal Year End	December (`12`)
Custom Fiscal Year End	Pass parameter (e.g., `CALENDARAUTO(3)`)
Useful For	Creating base for time intelligence functions

□ Practice Challenge:

Try the following:

1. Create a `DateTable` using `CALENDARAUTO()`.
2. Change the fiscal year end to March (`CALENDARAUTO(3)`).
3. Compare the output ranges.
4. Visualize both tables side-by-side and observe the differences in start and end dates.

◆ Section 3: Creating a Date Table Using `CALENDAR()` in Power BI

🎯 Objective:

Learn how to **manually define a date table** using the `CALENDAR()` function and gain more control over the start and end of your date range — with static or dynamic inputs.

□ Key Concepts:

📌 What is the `CALENDAR()` Function?

- Creates a table with a single column (`Date`) of **continuous dates** between a **start date** and an **end date**.
- You can:
  - Provide fixed/static dates.
  - Or dynamically determine start and end based on data (e.g., from a Sales table).

🛠 Method 1: Create a Date Table with Fixed Dates

✅ Step-by-Step Instructions:

1. Go to **Modeling** → click on **New Table**.
2. Use the `DATE()` function to specify your date range:

```
DAX

ManualDateTable = CALENDAR(DATE(2018, 5, 1), DATE(2020, 5, 1))
```

📝 This creates a continuous list of dates from **May 1, 2018** to **May 1, 2020**.

🛠 Method 2: Create a Date Table with Dynamic Dates from Data

You can dynamically use `MINX()` and `MAXX()` to extract earliest and latest dates from your dataset.

✅ Step-by-Step Instructions:

```
DAX

DynamicDateTable = CALENDAR(
    MINX(Sales, Sales[Date]),
    MAXX(Sales, Sales[Date])
)
```

🔍 This will:

- Look inside the `Sales` table.
- Find the **earliest and latest dates**.
- Automatically update if new data is added later.

Summary:

Feature	`CALENDAR()`
Output	Table with one `Date` column
Start & End Input	Required (manual or dynamic)
Fiscal Year Support	❌ No direct fiscal year handling
Customization	✅ High (choose source columns/tables)
Auto-Updates on Data Refresh	✅ Yes (if using `MINX()`/`MAXX()` logic)

Practice Challenge:

1. Create a `ManualDateTable` using fixed `DATE()` values.
2. Create a `DynamicDateTable` using `MINX()` and `MAXX()` on your `Sales` table.
3. Compare both tables visually in a **Table visual**.
4. Add new records to your Sales data (with different dates) and observe the update behavior of the dynamic date table.

◆ Section 4: Creating an Advanced Date Table with Time Intelligence Features

🎯 Objective:

Automatically create a **rich date table** with extended features like Year, Month Name, Quarter, Day of Week, etc., using a reusable DAX script. This date table will help power **time-based analysis** and **slicers** in reports.

□ Key Concepts:

📌 Why an Enriched Date Table?

- Simplifies **time intelligence** calculations.
- Enables dynamic grouping (e.g., by Month Name, Year-Month, Quarter).
- Supports **hierarchical drill-downs** in visuals.
- Allows for **custom sorting** (e.g., month names sorted correctly).

🛠□ Step-by-Step: Create a Smart Date Table Using Script

✅ Step 1: Create New Table

1. In Power BI, go to **Modeling** → click **New Table**.
2. Paste the following reusable DAX script into the formula bar:

```
DAX

DateTable =
ADDCOLUMNS(
    CALENDAR(MINX(Sales, Sales[Date]), MAXX(Sales, Sales[Date])),
    "Year", YEAR([Date]),
    "Month Number", MONTH([Date]),
    "Month Name", FORMAT([Date], "MMMM"),
    "Short Month", FORMAT([Date], "MMM"),
    "Year-Month", FORMAT([Date], "YYYY-MM"),
    "Quarter", "Q" & FORMAT([Date], "Q"),
    "Day", DAY([Date]),
    "Day of Week", WEEKDAY([Date], 2),
    "Weekday Name", FORMAT([Date], "dddd"),
    "IsWeekend", IF(WEEKDAY([Date], 2) >= 6, TRUE, FALSE),
```

```
    "Week Number", WEEKNUM([Date], 2)
)
```

📋 What This Script Adds:

Column Name	Description
`Date`	The actual date
`Year`	Calendar year
`Month Number`	Numeric month (1–12)
`Month Name`	Full month name (e.g., January)
`Short Month`	Short month name (e.g., Jan)
`Year-Month`	Combined year and month (e.g., 2024-07)
`Quarter`	Quarter of the year (e.g., Q1)
`Day`	Day of the month
`Day of Week`	Numeric day of week (Mon=1, Sun=7)
`Weekday Name`	Name of the weekday (e.g., Monday)
`IsWeekend`	TRUE if Sat/Sun, FALSE otherwise
`Week Number`	ISO-like week number (week starts on Monday)

Practice Activity:

1. Add a **Table visual** to your report.
2. Select fields from the new `DateTable` (e.g., `Year-Month`, `Quarter`, `IsWeekend`).
3. Sort Month names by `Month Number` for correct chronological order.
4. Test filtering your Sales data using this `DateTable`.

🔗 Step 2: Create Relationship in Model View

1. Go to **Model View**.
2. Drag `DateTable[Date]` → drop on `Sales[Date]`.
3. This creates a **1-to-many relationship**, enabling cross-filtering.

✅ Now your custom date table is connected and usable for all visuals and time intelligence operations.

Summary:

Task	Tool/Function
Create date range	`CALENDAR()` + `MINX()`/`MAXX()`

Task	Tool/Function
Add features	`ADDCOLUMNS()` with DAX functions
Connect to fact tables	Drag `Date` to `Sales[Date]`
Reuse	Replace `Sales[Date]` as needed

Practice Challenge:

- Modify the script to use a different date column (e.g., `Orders[OrderDate]`).
- Add a custom column called `"IsCurrentYear"` using:

```
DAX

"IsCurrentYear", YEAR([Date]) = YEAR(TODAY())
```

- Use it to filter visuals for current year analysis.

Section 5: Creating a Key Measures Table and Basic Aggregation Functions in DAX

✅ Why Organize Measures?

- Measures might get created in different tables by default, making them hard to manage.
- Keeping all measures in a single **Key Measures Table** helps maintain cleanliness and structure in the model.
- Especially helpful when you're dealing with **10–50+ measures** in a project.

Steps to Create a Key Measures Table

1. **Go to the Home Ribbon** → Click **"Enter Data"**.
2. **Create a blank table**:
   - No data input needed.
   - Name the table: **`Key Measures Table`**.
3. **Click "Load"** → A new table is created.

📦 Move Existing Measures into the Key Measures Table

1. Select the measure you want to move.
2. In the properties pane:
   - Look for **"Home Table"** option.
   - Change it to **`Key Measures Table`**.
3. Repeat for all other measures you've created.

□ **Clean Up Unused Columns**

- Delete the automatically generated **“Column1”** from the new table:
  - Right-click the column → Select **“Delete”**.

✅ **Result**

- You now have a **dedicated table** that holds all your DAX measures.
- Cleaner model, better organization, easier navigation.

Section 6: Count Functions in DAX

So now that we’ve covered the basic aggregation functions like `AVERAGE`, `MAX`, and `SUM`, let’s dive into another category of essential DAX functions: the **count functions**.

Count functions are used to **count rows or specific values** in a column, and they’re extremely useful when you want to measure activity, quantity of transactions, or even track unique entries in your dataset.

Common Count Functions in DAX

There are a few types of count functions you should be aware of:

1. **COUNT**
   - This function counts the number of **non-blank values** in a column.
   - Example: If you have a sales table and you want to count how many times a product was sold (i.e., number of transactions), you can use:

     ```
     DAX

     Total Transactions = COUNT(Sales[Quantity])
     ```

2. **COUNTA**
   - This function counts the number of **non-blank values** in a column **of any data type**.
   - Typically used on text or mixed-type columns.
3. **COUNTROWS**
   - This function counts the total number of rows in a table. It’s often used when you want to count **how many rows meet a certain condition** or just count total entries.
   - Example:

     ```
     DAX

     Row Count = COUNTROWS(Sales)
     ```

4. **DISTINCTCOUNT**
   - This function returns the **number of unique values** in a column.

    - Very useful when you want to know, for example, how many unique customers placed an order.
    - Example:

      DAX

      ```
      Unique Customers = DISTINCTCOUNT(Sales[CustomerID])
      ```

Let's Create Some Count Measures

To get practical, let's create a couple of these count measures.

1. **Total Transactions (Count of Quantity column):**
    - Navigate to your **Key Measures Table**
    - Click on **New Measure**
    - Write:

      DAX

      ```
      Total Transactions = COUNT(Sales[Quantity])
      ```

2. **Unique Customers:**
    - Create another new measure in the Key Measures Table:

      DAX

      ```
      Unique Customers = DISTINCTCOUNT(Sales[CustomerID])
      ```

3. **Total Sales Records (Using COUNTROWS):**
    - And another measure:

      DAX

      ```
      Total Sales Records = COUNTROWS(Sales)
      ```

Now, go ahead and add these measures to a table visual. You can combine them with dimension attributes like Customer Name or Location to analyze the counts in more detail.

Summary

Count functions are straightforward but very powerful in helping you:

- Understand data volume
- Identify unique entries
- Monitor non-blank activity

Section 7: Understanding Iterative Aggregation Functions (x Functions)

Now that we've covered the basic and extended versions of aggregation functions—like `COUNT`, `COUNTROWS`, `COUNTA`, and `COUNTBLANK`—it's time to go one level deeper and explore an incredibly powerful concept in DAX: the **x functions**.

These include:

- `SUMX`
- `AVERAGEX`
- `COUNTX`
- And other similar iterative functions like `MINX`, `MAXX`, `PRODUCTX`, etc.

So, **what does the x in these functions stand for?**
It stands for **"iterator."** These functions **iterate row by row over a table** and evaluate an expression for each row—then perform the aggregation.

Let's break that down step by step.

🔁 Why Use x Functions?

The standard aggregation functions like `SUM`, `AVERAGE`, `COUNT`, etc., work **directly on a column**. But what if:

- You need to calculate a value that's **not in a single column**, but is the result of a **formula involving multiple columns**?
- You need to evaluate a **custom expression for each row**, then aggregate the results?

That's where `X` functions shine.

□ Understanding `SUMX` with an Example

Let's say you have:

- A `Sales` table
- Each row contains:
  - `Quantity`
  - `PricePerUnit`

You want to calculate the **total revenue**, which is `Quantity * PricePerUnit` for each row.

If you just try `SUM(Sales[Quantity] * Sales[PricePerUnit])`, DAX will throw an error—it doesn't know how to multiply two columns in that context.

Instead, use `SUMX` like this:

DAX

```
Total Revenue = SUMX(
    Sales,
    Sales[Quantity] * Sales[PricePerUnit]
)
```

Here's what's happening:

- `SUMX` iterates over the `Sales` table **row by row**
- For each row, it evaluates `Quantity * PricePerUnit`
- Then it sums up all those results into the final total revenue

---

□ Try It Yourself

Let's create this measure in your **Key Measures Table**:

1. Select the **Key Measures Table**
2. Click **New Measure**
3. Enter this:

```
DAX

Total Revenue = SUMX(
    Sales,
    Sales[Quantity] * Sales[PricePerUnit]
)
```

Now visualize it with a card or table visual, and you'll see the correct revenue calculated based on the row-level formula.

📊 Using `AVERAGEX` – Row-Based Averages

Similarly, you can calculate an average **of a calculated expression** using `AVERAGEX`.

Example:

```
DAX

Average Revenue Per Transaction = AVERAGEX(
    Sales,
    Sales[Quantity] * Sales[PricePerUnit]
)
```

This gives the **average revenue per row** in your `Sales` table, which is different from simply averaging the `Quantity` or `Price`.

✅ Summary – When to Use x Functions

Use `X` functions when:

- You need to **evaluate a custom expression row-by-row**
- You are **multiplying or combining multiple columns** before aggregating
- You want full control over how values are aggregated

Function	Use Case
`SUMX`	Sum of row-based expressions
`AVERAGEX`	Average of calculated expressions
`COUNTX`	Count of values after expression
`MINX`	Minimum of row-based results
`MAXX`	Maximum of row-based results

🔢 Advanced Aggregation Functions in Power BI – The "X" Variants

After understanding the basic aggregation functions (like `SUM`, `AVERAGE`, etc.), it's crucial to dive into their more **advanced and powerful variants**, which end with an **X**, such as `SUMX` or `AVERAGEX`.

❌ The Common Mistake

Many beginners attempt to write a measure like this:

```
DAX

Revenue = SUM(Sales[Quantity]) * SUM(Sales[Price])
```

This calculates the **total quantity** multiplied by the **total price**, leading to incorrect results. For example, instead of the actual revenue (~140,000), it might show something like **10 million**.

This happens because `SUM()` cannot take expressions—it only works on a single column.

✅ The Correct Way: `SUMX`

The right approach uses `SUMX`, which allows row-by-row evaluation:

```
DAX

Revenue = SUMX(Sales, Sales[Quantity] * Sales[Price])
```

How it works:

- `SUMX` takes two parameters:
  1. A table (`Sales`)
  2. An expression to evaluate row-by-row (`Quantity * Price`)
- First, it calculates `Quantity * Price` **for each row**
- Then, it **sums up** all those individual row results

This replicates how a calculated column works (`Quantity * Price`), but inside a measure—**more dynamic and memory-efficient**.

📌 Calculated Column vs Measure

Feature	Calculated Column	Measure
Location	Stored in table	Stored in the model (not tied to a table)
Context Sensitivity	Row-level, static	Fully context-aware, dynamic
Use Case	Needed when value is reused in multiple rows	Preferred for aggregations and visuals

Exercise :

SUMX vs. SUM and AVERAGEX vs AVERAGE

The AVERAGEX function works just like the SUMX function.
Sometimes you have to use the RELATED function as well.

Question 1:
What is the average price of our products?
Would you use the AVERAGE or AVERAGEX function?
Hint: The "amount" of sales is NOT considered, so each product is considered equally.
Answer: 3,88.

Question 2 (more difficult):
What is the average profit of our sold products?
(not considering how many of these are sold, so that in average every product is just counted once)
Would you use the Average or AverageX Function?
Hint 1: Amount of sales should *not* be considered.
Hint 2: Work on the products table
Hint 3: profit = price * profit margin
Answer: 0,78$.

Question 3:
What is the total quantity of our sales?
Answer: 35377.

Question 4 (a bit more difficult):
What is the total profit of our sold products?
Hint 1: SUM or SUMX?
Hint 2: Use the RELATED function.
Answer: 27.612,70$.

Question 5 (also a bit more difficult):
What is the average tax amount we pay in our sales?
Hint: Use the RELATED function for the tax rate and the AVERAGEX function in the sales table.
Answer: 1,42$.

Section 8: 🎯 Understanding Filter Context in DAX

To understand how **measures** are calculated in Power BI, you must grasp the concept of **filter context**.

□ What is Filter Context?

Filter context defines **which subset of the data** is being used in a calculation.

🔍 Two Types of Filtering Affect Filter Context

1. External Filters (from the report visuals)

- Example: You click on a **specific employee** or **product** in a slicer or filter pane.
- This restricts the data used in a visual and directly impacts measure results.

📌 Example:

- Selecting *Employee A* will update your **Revenue** measure to reflect only sales made by Employee A.

2. Visual-Level Filters (Internal Filters)

- These are applied by the **structure of the visual itself**.
- Example: If you're showing `Revenue` by `Customer ID`, then:
  - For each row (each Customer ID), Power BI filters the data to **just that customer** before calculating the measure.

🔁 How Power BI Evaluates a Measure

Step-by-Step:

1. **Step 1: Determine the filter context**
   - Based on selected slicers and visual structure (e.g., Customer ID, Product, Date).

  - Example: Visual is grouped by `Customer ID = 1`.
2. **Step 2: Apply the measure**
  - Power BI now runs the DAX formula **on only that filtered data**.
  - Example: Customer ID 1 has 8 rows → Revenue = 90 (calculated from those 8 rows only).

✅ Key Takeaways

- Always remember: **Filter context comes first**, then the measure is applied.
- Measures are **dynamic** and change based on the filters applied—either by the user or by the visual layout.
- This is why DAX measures are so powerful—they **react automatically** to the context.

💡 Pro Tip

Don't get overwhelmed by context. Instead:

- Be **aware** of it while designing measures.
- **Test** your DAX using small datasets or matrix visuals to observe how filters affect results.

Section 9: Understanding Row Context in DAX

What is Row Context?

- Row context applies when you calculate **expressions row-by-row**.
- Commonly used in **X functions** like `SUMX`, `AVERAGEX`, `MAXX`, etc.
- **Each row** is evaluated **individually**, and the expression result is temporarily stored in memory.

🔍 How Row Context Works: Step-by-Step

1. Power BI evaluates the **expression** (e.g., `Quantity * Price`) for **each row** of the specified table.
2. Each result is stored **temporarily in memory**.
3. Once all rows are evaluated, Power BI **aggregates** them using:
  - `SUMX` → Adds the row-level results.
  - `AVERAGEX` → Calculates the average of those results.

- etc.

📊 Example: Revenue Calculation

DAX

```
Revenue = SUMX(Sales, Sales[Quantity] * Sales[Price])
```

- Here, for each row in the `Sales` table:
  - `Quantity * Price` is computed first.
  - Then all values are **summed** by `SUMX`.

💡 Important Insight

- `SUMX(Sales, Sales[Price])` will return **the same result** as `SUM(Sales[Price])`.
- But `SUM` is **more efficient** than `SUMX` when no row-wise calculation is needed.
- Therefore:

  ✅ Use **basic aggregation** like `SUM`, `AVERAGE`, etc. when you don't need per-row logic.
  ⚠️ Use **X functions** only when you're applying **row-level expressions**.

✅ Key Takeaway

- **Row context = row-by-row evaluation.**
- Use it **only when needed**, as it's more **memory-intensive** and slower than regular aggregations.
- Be deliberate about choosing `SUMX` vs `SUM` based on your calculation need.

Section 10: Understanding the Data Model and Its Impact on Measures

Why the Data Model Matters

- The **data model** defines how tables relate to each other in Power BI.
- It's essential for combining data across multiple tables in your visuals and measures.
- **Filter context** and **row context** both rely on the underlying model to propagate correctly.

🔗 Using Related Tables Together

- Example: Combine `Customer Name` (from Customer table) with `Revenue` (from Sales table) in one visual.
- This is possible **only because of relationships** defined in the model.

📊 Types of Tables in a Model

- **Fact Table**: Main table with transactional data (e.g., Sales).
- **Dimension Table**: Lookup or descriptive data (e.g., Customers, Products).
- These are typically joined using a key (like `Customer ID`).

🔄 How Relationships Work

- Sales table contains `Customer ID`.
- This ID is **matched** with `Customer ID` in the Customers table.
- Power BI **uses the relationship** to pull customer details based on the ID.

❗ Common Issue: No Relationship

- Many problems in DAX or visuals are due to missing relationships.
- Always **check your data model** if your measure or visual isn't working as expected.
- Example: If `Customer Name` doesn't filter `Sales`, check if the relationship exists and is active.

✅ Takeaway

- A well-designed data model is **key** to writing correct DAX and building accurate reports.
- Always **validate your relationships** and **test them** when something seems off.

Section 11: The CALCULATE Function in Power BI

◆ Introduction to CALCULATE

- **CALCULATE** is *the most important function* in DAX.
- It allows you to **evaluate an expression in a modified filter context**.
- Most advanced DAX functions are built on top of `CALCULATE`.

◆ Why Use CALCULATE?

- It changes the **context** in which a measure or expression is calculated.
- Enables complex calculations like time intelligence, conditional filtering, etc.

◆ Basic Syntax

```
DAX

CALCULATE(
    <expression>,       -- Can be SUM, COUNT, or an existing measure
    <filter1>, <filter2>, ...
)
```

🔧 Example 1: Revenue Last Year

✅ Step-by-Step Setup

1. **Prepare Table:**
   - Use the **Date** column in normal format (not hierarchy).
   - Display as a table and format it to remove time.
2. **Use Revenue Measure:**
   - Drag existing **Revenue** measure into the table.
   - Format as **Currency** with **2 decimal places**.
3. **Create New Measure:**

   ```
   DAX

   Revenue LY = CALCULATE(
       [Revenue],
       SAMEPERIODLASTYEAR('Date'[Date])
   )
   ```

   - `SAMEPERIODLASTYEAR` expects a date column.
   - This measure shifts the context **one year back**.
4. **Visualize and Format:**
   - Add `Revenue LY` to the table.
   - Format it similarly.
5. **Test Behavior:**
   - Compare values for April 30, 2020 vs. April 30, 2019.
   - Works also for **month-level** or **year-month** (e.g., May 2018 → May 2017).

🔧 Example 2: Revenue Filtered by State

✅ Setup

1. **Create New Measure:**

   DAX

   ```
   Revenue Utah = CALCULATE(
       [Revenue],
       'Location'[State] = "Utah"
   )
   ```

2. **Behavior:**
   - The context of this measure **always filters to Utah**, regardless of the row in the table.
   - Used in a state-wise table, it **still shows the Utah revenue for all rows**.

💡 Key Learning:

- CALCULATE **overrides** outer context when an internal filter is applied.
- Even if the table is grouped by different states, the measure sticks to "Utah".

✅ Takeaways

- CALCULATE is **used to change filter context**.
- You can pass:
  - Time-intelligence functions like `SAMEPERIODLASTYEAR()`
  - Conditional filters (like `'Location'[State] = "Utah"`)

Section 12: The FILTER Function in CALCULATE – Best Practices and Usage

Now let's look at one of the most commonly used filter functions inside the `CALCULATE` function — the `FILTER` function.

Recap: Previous Filtering Example

Previously, we had a measure that filtered revenue by state using a simple expression like this:

dax

```
CALCULATE([Total Revenue], 'Location'[State] = "Utah")
```

This expression works, and it calculates revenue only for the state of Utah. But this kind of direct filter is not always ideal.

Why Use the `FILTER` Function Instead?

It's best practice to use the `FILTER` function instead of directly filtering inside `CALCULATE`. Here's how the rewritten measure would look:

```
dax

CALCULATE(
    [Total Revenue],
    FILTER(
        'Location',
        'Location'[State] = "Utah"
    )
)
```

Using the `FILTER` function like this makes your DAX code more robust and accurate — especially when working with more complex models.

Understanding the `FILTER` Function

The `FILTER` function is a **table function**, which means it returns a **filtered version of a table**, not just a single value. Its syntax:

```
dax

FILTER(<table>, <filter_expression>)
```

- `<table>`: the table you want to filter (e.g., `Location` or `Products`)
- `<filter_expression>`: a condition that returns TRUE or FALSE (e.g., `Price > 5`)

Let's visualize how this works with a real table.

Creating a Sample Filter Table

You can create a new table using the `FILTER` function to see the result.

```
dax

Sample Filter Table =
FILTER(
    Products,
    Products[Price] > 5
)
```

This new table will:

- Include all the same columns as the `Products` table
- Contain **only the rows** where `Price > 5`

If you change the condition, for example:

```
dax
```

```
Products[Price] > 6
```

…the filtered result will change accordingly.

This example helps you **see** what the `FILTER` function does — it returns a **new table**, filtered according to the condition you specify.

FILTER Inside CALCULATE

Now that we understand `FILTER` is a table function, it becomes clear why we use it inside `CALCULATE`:

```
dax

CALCULATE(
    [Total Revenue],
    FILTER(
        Products,
        Products[Price] > 6
    )
)
```

This tells Power BI:

"Calculate total revenue, but only considering products that have a price above 6."

This is one of the most **common and powerful ways** to apply filters in DAX.

Section 13 Summary: Context Modification Using the `ALL()` Function in DAX

In this section, we explore another way to **modify the filter context** within DAX calculations—using the `ALL()` function. Here's the breakdown:

🔄 What is `ALL()` in DAX?

- `ALL(<table_or_column>)` **removes any filters** that have been applied to the specified **table or column**.
- Commonly used to calculate **totals or percentages** that are **not affected by filters** in visuals.

□ Example Demonstration:

- Assume a table visual that displays **Revenue by State**.

- Normally, each row shows revenue filtered by state.
- But if we use `ALL(State)` inside `CALCULATE`, we **remove the state filter**, and all rows show the **total revenue across all states**.

```
DAX

Total Revenue All States := CALCULATE(SUM(Sales[Revenue]),
ALL(Location[State]))
```

- Result: Each row now shows the **same total** (e.g., ₹139,000), **ignoring state filters**.

📊 Use Case: Calculate % Revenue by State

You can compare filtered revenue to the total revenue:

```
DAX

% Revenue by State :=
DIVIDE(
    SUM(Sales[Revenue]),
    CALCULATE(SUM(Sales[Revenue]), ALL(Location[State]))
)
```

- Format this measure as a **percentage**.
- Each state now shows its **proportional revenue contribution** to the total.

⚠️ What If You Change the Visual Context?

- If you visualize **products** instead of states, your `ALL(State)` doesn't help because **product filters still apply**.
- Result: You get 100% everywhere, because you're dividing a filtered revenue by itself.

✅ **Fix:** Use `ALL(Products)` instead to clear the product filters.

External Filters (e.g., Slicers)

- Suppose a slicer filters by product **price**.
- Even if you used `ALL(Products)`, this slicer can still filter the result.
- Result: Your percentages **won't add up to 100%** anymore.

💡 Goal: Keep Percentages Relative to Current Filter Selections

To maintain 100% totals **based on current selections** (e.g., slicers), use:

```
DAX

ALLSELECTED()
```

➡ This is covered in the **next section**.

✅ Key Takeaways:

- `ALL()` removes filters on specified tables/columns.
- Use it to calculate totals or normalize measures.
- Be mindful of external filters and visuals—they still impact results unless handled with care.
- For dynamic % calculations that adapt to slicers, `ALLSELECTED()` is the next step.

Section 14: Understanding the `ALLSELECTED()` Function

🔄 Why Use `ALLSELECTED()` Instead of `ALL()`?

- `ALL()` removes **all filters** from the specified column/table—**including slicers or external filters**.
- `ALLSELECTED()` removes only the filters **within the visual**, but **preserves slicer or page-level filters** (external context).
- This helps calculate **relative percentages** that always **sum to 100% based on current external selections**.

Scenario: Revenue % by Product (with Filters)

- If you use:

  ```
  DAX

  % Revenue =
  DIVIDE(
      SUM(Sales[Revenue]),
      CALCULATE(SUM(Sales[Revenue]), ALL(Products))
  )
  ```

  - You’ll get 100% **only when no slicers are applied**.

- Applying a **price filter via slicer** keeps the **revenue in numerator filtered**, but **denominator remains full**, leading to **percentages that don't total 100%**.

✅ **Instead, use:**

```
DAX

% Revenue =
DIVIDE(
    SUM(Sales[Revenue]),
    CALCULATE(SUM(Sales[Revenue]), ALLSELECTED(Products))
)
```

➡️ Now, the measure **adapts** to slicers or page filters, and **percentages always total 100%** across the visual.

📊 Key Differences: `ALL()` vs `ALLSELECTED()`

Feature	`ALL()`	`ALLSELECTED()`
Removes Visual Filters	✅ Yes	✅ Yes
Removes Slicer Filters	✅ Yes	❌ No (keeps them)
% Adds to 100% Always	❌ Not when slicers applied	✅ Always within current external filters
Use Case	Total independent of any filter	% of total under current selection

---

✅ When to Use `ALLSELECTED()`

- You want to calculate **percentages** that:
  - Respect user selections from slicers or filters.
  - Still ignore the breakdown filters from inside the visual.
- Example: `% Revenue by Product Category` that totals 100% after applying **price** or **region** slicers.

💡 Summary

- Use `ALL()` to get **absolute totals**.
- Use `ALLSELECTED()` for **dynamic, relative totals** that respect user selections.
- It ensures **visual integrity** (like % = 100%) when filters are active.

✅ Section 15: Using `ALL` and `ALLEXCEPT` Functions to Control Filter Context in Power BI

📌 Use Case Overview

You want to analyze **revenue percentages** and **profit margins** with respect to **tax rates**, and control how percentages are calculated — either **by row** or **by column** — depending on the visual layout.

☐ The Problem

When you show percentages in a matrix visual that includes both **tax rate** and **profit margin**, the values **do not sum to 100%** across either rows or columns. You want to control this behavior:

- Sum to 100% **across rows** (per tax rate)
- Sum to 100% **across columns** (per profit margin)

💡 Solution: `ALLEXCEPT()`

`ALLEXCEPT()` removes all filters **except** the one(s) you specify.

Formula Structure

DAX

```
CALCULATE(
    [Revenue],
    ALLEXCEPT(Products, Products[TaxRate])
)
```

➡ This keeps filtering **by Tax Rate** while ignoring other filters like Product or Profit Margin.

📊 Visual Setup

1. Add **Products**, **Revenue**, and **Tax Rate** into a matrix.
2. Switch **Revenue** to show as percentage.
3. Add **Profit Margin**, ensure it's formatted as decimal and **"Don't summarize"**.
4. Use matrix visual for readability (to analyze multi-dimensional context).
5. Use `ALLEXCEPT()` in your measure to control which dimension (row or column) your 100% total relates to.

🔄 Behavior Based on Filter Context

- If you write:

  DAX

  ```
  CALCULATE([Revenue], ALLEXCEPT(Products, Products[TaxRate]))
  ```

  → 100% sum across **rows** (tax rates remain, other filters cleared).

- If you write:

  DAX

  ```
  CALCULATE([Revenue], ALLEXCEPT(Products, Products[ProfitMargin]))
  ```

  → 100% sum across **columns** (profit margins remain, other filters cleared).

✅ Key Takeaways

- Use `ALL()` to **remove all filters**.
- Use `ALLSELECTED()` to **keep user-selected filters** in slicers.
- Use `ALLEXCEPT()` to **remove all filters except specified ones** — perfect for **row/column percentage analysis** in matrix visuals.

Excersies

FILTER and ALL

Question 1:
What is the average quantity we have sold for products that are more expensive than 3 dollars?
Answer: 2,96.

Question 2:
What is the percentage of products sold (measured by quantity) that are more expensive than 3 dollars?
Answer: Products that are more expensive than 3 dollars make up 66% (rounded) of all products sold (measured by quantity).

Question 3:

How can you create a measure that calculates the percentage of total revenue that was made in a certain year?
For example 2017: x % of our all-time revenue, 2018: y % of our all-time revenue.
What is the percentage of revenue that was made in 2019?
Hint: This measure you want to use in a table or bar chart together with the year's column.
Answer: 57% (rounded).

Section 16: Complex Filtering with Logical Operators in DAX

In this section, we extend basic DAX filtering to more **complex logical conditions** using operators such as **AND (`&&`)**, **OR (`||`)**, and **NOT (`NOT()`)**.

◆ Example 1: Basic Filtering

We begin by filtering the `Products` table:

```
dax

FILTER(Products, Products[Price] < 6)
```

This simple filter gives us all products priced below $6.

◆ Example 2: Adding AND (`&&`) Condition

To make it more restrictive:

```
dax

FILTER(Products, Products[Price] < 6 && Products[Price] > 4)
```

This filter includes only products priced **between $4 and $6**. If no products match (e.g., in the state of Missouri), the result will be blank.

◆ Example 3: Filter by Text using `LEFT()`

We add a condition based on the **first letter** of product names:

```
dax

FILTER(
    Products,
    Products[Price] < 6 &&
```

```
    Products[Price] > 4 &&
    LEFT(Products[ProductName], 1) = "D"
)
```

This returns products priced between $4–$6 **and** starting with "D".

If no products match all three, the table will return no results.

◆ Example 4: Using OR (`||`) Operator

Now let's loosen the criteria:

```dax
FILTER(
    Products,
    (Products[Price] < 6 && Products[Price] > 4) ||
    LEFT(Products[ProductName], 1) = "C"
)
```

This condition includes:

- Products with prices between $4 and $6 **OR**
- Products starting with "C"

⛔ **Precedence Note**: In DAX, **AND (`&&`) takes precedence over OR (`||`)**.

◆ Example 5: Exclude Certain Products

Suppose we want **all products EXCEPT those that start with "C"**, and optionally filter by price too.

Option A – Using <> (Not Equal)

```dax
FILTER(
    Products,
    LEFT(Products[ProductName], 1) <> "C" ||
    (Products[Price] < 6 && Products[Price] > 4)
)
```

Option B – Using `NOT()`

```dax
FILTER(
    Products,
    NOT(LEFT(Products[ProductName], 1) = "C") ||
```

```
    (Products[Price] < 6 && Products[Price] > 4)
)
```

Both expressions exclude products starting with "C" unless they fall into the specified price range.

⚠ Readability and Maintainability Tip

As complexity grows:

- Use **formatting** and **indentation**
- Use **comments** (`// This filters out products that start with "C"`)
- Consider breaking logic into **intermediate variables** using `VAR`

This enhances collaboration and makes measures understandable for others and for your future self.

Section 17: 📊 Power BI: Making Complex Measures Readable

Challenge:

Long and complex DAX measures are hard to read, understand, and maintain—especially weeks later.

✅ Best Practices for Readability

1️⃣ Line Breaks with `Shift + Enter`

- Move to a new line **without executing** the formula.
- Helps organize expressions and functions clearly.

2️⃣ Indenting with `Tab`

- Use indentation inside functions like `FILTER()` or `CALCULATE()` to separate logical blocks.

Example:

```
dax

Revenue Filtered :=
CALCULATE (
```

```
    [Total Revenue],
    FILTER (
        Products,
        Products[Category] = "A"
    ),
    FILTER (
        Locations,
        Locations[State] = "Arizona"
    )
)
```

🗨️ Use Comments (`//`)

- Add context or explanation inline.
- Ignored during calculation.
- Helps others (and your future self) understand logic.

```dax
// This measure filters by product category and Arizona state
Revenue Filtered :=
CALCULATE (
    [Total Revenue],
    FILTER (
        Products,
        Products[Category] = "A"
    ), // Product category filter
    FILTER (
        Locations,
        Locations[State] = "Arizona"
    ) // Location filter
)
```

💡 Tips:

- No fixed way to format — just keep it **clear and logical**.
- Use comments for:
    - Purpose of the measure
    - Special logic applied
    - Unusual business rules

Section 18 Concept Focus: Using `VALUES()` with `AVERAGEX()` to calculate Monthly Average Revenue

---

🔍 What is the `VALUES()` Function?

- **`VALUES(column_name)`** is a **table function** in DAX.
- It returns a **distinct list of values** from the given column.

- Used often to create custom filters or iteration contexts in X-functions (`SUMX`, `AVERAGEX`, etc.).

□ Example 1 – Create a Table of Unique Weekdays:

```
dax

Values Example = VALUES('Date'[Weekday])
```

- Returns a one-column table with **distinct weekdays**: Monday to Sunday.

🎯 Real-world Use Case: Monthly Average Revenue per Customer

We aim to calculate **average revenue per month** across customers.
We **don't have this directly in the data**, so we build it using DAX logic.

🛠 Steps to Build the Measure:

1. **Create a New Measure:**

```
dax

Monthly Average Revenue =
AVERAGEX(
    VALUES('Date'[YearMonth]),  -- Table of unique months
    [Revenue]                   -- Existing measure to sum revenue
)
```

- `VALUES('Date'[YearMonth])` provides a distinct list of months.
- `AVERAGEX()` iterates over each month, calculates `[Revenue]`, and averages it.

2. 💡 **Why use `VALUES()` here?**
   - `AVERAGEX()` requires a **table input**.
   - `VALUES()` provides a **unique set of rows** (e.g., months).
   - That creates the **row context** to compute revenue **month-by-month**.

□ How it Works Internally:

- `VALUES('Date'[YearMonth])` → Table like:

YearMonth
2023-Jan
2023-Feb
...

- For each row:
  - `AVERAGEX` evaluates `[Revenue]` **in that month's context**.
- Then averages all monthly revenue totals.

🔁 Easy Modification – Quarterly Average Revenue:

Just replace `YearMonth` with `YearQuarter`:

```
dax

Quarterly Average Revenue =
AVERAGEX(
    VALUES('Date'[YearQuarter]),
    [Revenue]
)
```

✅ Summary

Function	Purpose
`VALUES()`	Returns distinct values from a column (as table)
`AVERAGEX()`	Averages an expression over a table
Use Together	Create custom contexts to calculate per-month/quarter metrics

🔢 Section 19: VALUES Function with AVERAGEX – Monthly Average Revenue

---

□ Core Concept

- Learn to use the `VALUES()` table function.
- Apply `VALUES()` in combination with `AVERAGEX()` to calculate **Monthly Average Revenue per Customer**.

□ Understanding the VALUES() Function

- **VALUES(<column>)** returns a **distinct list of values** from the given column.
- Useful when you want to **iterate** over unique values like weekdays, months, quarters, etc.
- Example:

  ```
  DAX
  ```

```
VALUES('Date'[Weekday])
```

Returns a one-column table with 7 rows (Monday to Sunday).

🛠️ Use Case: Monthly Average Revenue per Customer

- Goal: Calculate average revenue per month for each customer.

✅ Steps to Implement

1. **Go to "New Measure"**
   Define the following:

   ```
   DAX

   Monthly Avg Revenue :=
   AVERAGEX(
       VALUES('Date'[YearMonth]),  -- a distinct table of months
       [Revenue]                   -- pre-calculated measure
   )
   ```

2. **Explanation**:
   - `VALUES('Date'[YearMonth])`: Creates a table with distinct `Year-Month` combinations.
   - `AVERAGEX`: Iterates over that table.
   - For each month:
     - Evaluates `[Revenue]` (assumed to be sum of revenue per month).
     - Then averages the result across all months.

📈 Visualization

- Place `Monthly Avg Revenue` in a visual (like table/matrix).
- Format it as a currency.
- Compare it to total revenue to gain insights on monthly trends.

🔁 Flexibility: Switch Granularity

- You can swap `YearMonth` with:
  - `QuarterYear` → To get **Quarterly Average**
  - `Weekday` → To get **Average by Day of the Week**

Example:

```
DAX
```

```
Quarterly Avg Revenue :=
AVERAGEX(
    VALUES('Date'[YearQuarter]),
    [Revenue]
)
```

💡 Takeaway

- `VALUES()` is a **dynamic table constructor** for distinct values.
- Combine with `AVERAGEX()` or `SUMX()` to perform **granular aggregations**.

🏆 Section 20: Ranking with `RANKX()` and Context Control

🎯 Objective

Use the `RANKX()` function to **rank customers by their Quarterly Average Revenue**, and learn how to control **filter context** with `ALL()` and `ALLSELECTED()` for different ranking behaviors.

🔢 Step-by-Step: Basic Customer Ranking

✅ Goal: Rank customers based on their quarterly average revenue

1. **Create a new measure:**

```
DAX

Customer Rank :=
RANKX(
    ALL('Customer'),            -- Full customer list regardless of visual
filters
    [Quarterly Avg Revenue]     -- Existing measure for avg revenue
)
```

2. **Why use `ALL('Customer')`?**
   - `RANKX()` ranks **within the current context**.
   - If you're in a table with a single customer per row, the default context = 1 customer → all get rank 1.
   - Wrapping the table in `ALL()` **removes that filter**, enabling correct global ranking.

□ Dynamic Ranking Affected by Filters

- When applying filters (e.g., Year = 2023), the ranking recalculates **only within that filter**.
- This is **expected** behavior and useful in many analyses.

🔁 Freeze the Ranking Context with CALCULATE() + ALL()

□ Goal: Always rank customers based on all years, even when filtered

1. **Modify the ranking to freeze the year context:**

```
DAX

Static Year Rank :=
CALCULATE(
    RANKX(ALL('Customer'), [Quarterly Avg Revenue]),
    ALL('Date'[Year])   -- Removes filter on year
)
```

2. **Effect:**
    - Revenue can change based on year filter.
    - **Ranking stays constant**, always reflecting total-year data.

📊 Contiguous Rankings for Selected Customers Only

□ Goal: Rank only visible customers (e.g., selected in slicer)

1. **Use `ALLSELECTED()` instead of `ALL()`:**

```
DAX

Relative Rank :=
RANKX(
    ALLSELECTED('Customer'),
    [Quarterly Avg Revenue]
)
```

2. **Effect:**
    - Shows ranking **only among filtered/visible customers**.
    - Useful when showing a **subset** (e.g., top N customers).

🔍 Summary of Context Functions for Ranking

Context Function	Description
`ALL()`	Ignores all filters (global ranking)
`ALLSELECTED()`	Respects report/page/slicer filters (ranking among visible rows)
`CALCULATE()`	Changes context for any expression inside it

💡 Key Insights

- `RANKX()` is sensitive to filter context — this can be an advantage or a problem depending on your needs.
- `ALL()` gives **absolute** rankings.
- `ALLSELECTED()` gives **relative** rankings within selected groups.
- `CALCULATE()` can **override context** for targeted control.

▯ Section 21: Top N Filtering Based on Ranking

🎯 Objective

Allow users to dynamically filter the **Top N customers** based on their **Quarterly Average Revenue**, using a slicer with configurable values (e.g., Top 3, Top 10, Top 20, etc.).

🏗️ Step 1: Create a Helper Table

✅ Table Name: `Top N Filter Table`

Create a manual table with the following structure:

Top N Value	Top N Name
3	Top 3
5	Top 5
10	Top 10
20	Top 20
50	Top 50

This table allows users to choose how many top customers to view.

📌 Sort the `Top N Name` column

- Sort `Top N Name` **by** `Top N Value` to avoid incorrect alphabetical ordering in slicers.

□ Step 2: Create the Filtering Measure

✅ Measure Name: `Top N Filtered Revenue`

```
DAX

Top N Filtered Revenue :=
IF (
    [Customer Rank] <= MAX('Top N Filter Table'[Top N Value]),
    [Quarterly Avg Revenue],
    BLANK()
)
```

- ✅ `Customer Rank` — previously created `RANKX()` measure.
- ✅ `MAX()` — ensures proper behavior even if multiple slicer values are selected.
- ❌ Avoid using `VALUES()` here as it causes errors when multiple slicer values are selected.

🖥□ Step 3: Use in Report

- Add the **Top N Name** column to a slicer.
- Display the **Customer Name** and `Top N Filtered Revenue` in a table.
- Optional: Remove the raw ranking or revenue columns for a cleaner layout.

📐 Why Use `MAX()` Instead of `VALUES()`?

Function	Behavior When Multiple Values Are Selected
`VALUES()`	Returns a table → causes errors
`MAX()`	Returns the highest selected value → works ✅

🖼□ Alternative Filtering Option

Instead of a custom helper table, you can also:

- Use a **numeric slicer** with a "Top N Value" directly.
- But the helper table provides **custom labeling** (e.g., "Top 10") and better UX.

💡 Final Tip: Use Variables for Clarity (Coming Next)

As your DAX logic grows:

- Use `VAR` statements to store intermediate values.
- Improves **readability** and **debugging**.

✅ Outcome

You’ve now implemented a dynamic and user-friendly **Top N customer filter** based on a flexible ranking measure — ideal for executive dashboards and sales analytics.

Section 22: Using Variables in DAX

Why Use Variables in DAX?

As DAX formulas grow in complexity, using variables improves:

- **Readability**
- **Maintainability**
- **Reusability**
- **Performance (in some cases)**

Key Benefits

1. **Simplifies complex expressions**
   Long calculations can be replaced with a short, descriptive variable name.
2. **Reusability**
   If an expression is used multiple times, define it once as a variable and reuse it.
3. **Easier updates**
   Modify the expression once and it reflects everywhere it's used via the variable.

Basic Syntax

```
DAX

New Measure :=
VAR <VariableName1> = <Expression1>
VAR <VariableName2> = <Expression2>
RETURN
    <Final Expression using the variables>
```

Example

Let's say we want to calculate the difference between total revenue and a filtered revenue value:

Without Variables

```
DAX

Difference From Filtered Revenue :=
CALCULATE([Total Revenue]) -
CALCULATE([Filtered Revenue])
```

With Variables

```
DAX

Difference From Filtered Revenue :=
VAR FilteredRev = CALCULATE([Filtered Revenue])
RETURN
    [Total Revenue] - FilteredRev
```

With Multiple Variables and Comments

```
DAX

Difference From Filtered Revenue :=
VAR FilteredRev = CALCULATE([Filtered Revenue])
VAR DummyCalc = 1 + 1  // Just an example for a second variable
RETURN
    // Main calculation using variables
    [Total Revenue] - FilteredRev + DummyCalc
```

When to Use Variables

- **When a complex expression is used multiple times**
- **When the logic is too complex to read directly**
- **When making your DAX formula easier to understand for others or your future self**

Tips

- Always define variables at the beginning (right after the `=`).
- Use `RETURN` to specify the final calculation that uses those variables.
- You can also use comments (`//`) to explain sections of your code.

Exercise

IF & RANKX

Let's create your own custom filter in this exercise!

Question 1:
Create a custom filter that enables us to filter our products by their rank in regards to the revenue.
1: Top 10 products
2: Rank 11-20
3: Rank 21-40
4: Residual products

How much revenue is made by the products that *rank between 21 and 40?*
Hint: You can use the calculated column that we have already created ([Ranking by Revenue] column created with the RANKX function).
Answer:
$29,697.75

Question 2:
What is the average revenue we have made per day?
Hint: Use the VALUES function and also use the "Revenue Measure" we have already created earlier in the course!

Answer:
$134.21

Section 23: Time Intelligence Functions in DAX

Overview

Time intelligence functions in DAX allow us to perform powerful and insightful **date-based calculations**. These functions are relatively easy to use and provide access to comparisons like:

- This year vs last year
- Rolling averages
- Month-over-month growth
- Year-to-date totals

These calculations **would be difficult or impossible to do in regular Power BI visuals without DAX.**

Key Time Intelligence Function: `DATEADD`

Purpose

The `DATEADD()` function shifts dates in a column forward or backward by a specific interval (e.g., days, months, years).

Syntax

```
DAX

DATEADD(<dates>, <number_of_intervals>, <interval_type>)
```

- `<dates>`: A date column (e.g., `Dates[Date]`)
- `<number_of_intervals>`: Integer value (positive or negative)
- `<interval_type>`: "DAY", "MONTH", "QUARTER", or "YEAR"

Practical Example

Step-by-step Calculation

```
DAX

DateAdd Example Measure :=
CALCULATE(
    [Revenue],
    DATEADD('Dates'[Date], 2, DAY)
)
```

- **Shifts the date context by 2 days forward**
- Can also use `-2` to go **2 days backward**
- Change `"DAY"` to `"MONTH"`, `"QUARTER"` or `"YEAR"` for broader comparisons

Visualization Insight

You can:

- Add this measure to a **table** with dates to see shifted revenue values
- Create a **line chart** to compare shifted revenue values across time
- Combine this with **hierarchies** like year/month/quarter and it will still work

Flexible Date Shifts

For example:

```
DAX

CALCULATE(
    [Revenue],
    DATEADD('Dates'[Date], -1, YEAR)
```

```
)
```

Shows revenue **from one year ago**, useful for year-over-year comparisons.

Advanced Example: Calculate Year-Over-Year Difference

Using `VAR` and `RETURN`, you can calculate the **difference between current year and previous year**:

```
DAX

YoY Difference :=
VAR RevenueLastYear =
    CALCULATE([Revenue], DATEADD('Dates'[Date], -1, YEAR))
RETURN
    [Revenue] - RevenueLastYear
```

Best Practices

- Use **`DATEADD()`** inside the `CALCULATE()` function to shift context
- Works well with **line charts**, **tables**, and **time hierarchies**
- Combine with **variables (`VAR`)** for clean and readable code
- Easily adaptable to months, quarters, or years

Conclusion

- Time intelligence functions, especially `DATEADD`, are powerful and simple
- Best used **inside `CALCULATE()`** to alter the filter context
- Perfect for **comparative time analysis**
- Essential for any serious Power BI or DAX user

🔢 Section 24: Time Intelligence — Month-to-Date, Quarter-to-Date, and Year-to-Date Measures

🎯 Goal

Learn to use **time intelligence functions** like:

- **Month-to-Date (MTD)**
- **Quarter-to-Date (QTD)**

- **Year-to-Date (YTD)**
  …to **calculate cumulative totals** over time, such as Revenue up to a specific date within a month, quarter, or year.

💡 What These Functions Do

- `DATESMTD()`: Returns dates **from the beginning of the current month up to the selected date**.
- `DATESQTD()`: Returns dates **from the start of the quarter to the selected date**.
- `DATESYTD()`: Returns dates **from the start of the year to the selected date**.

Each of these is typically used with `CALCULATE()` to **alter the date context** of a base measure like Revenue.

✅ Base Measure – Revenue

If not already created, define this measure first:

```
dax

Revenue := SUM(Sales[TotalAmount])
```

📅 Month-to-Date (MTD) Measure

```
dax

Revenue MTD :=
CALCULATE(
    [Revenue],
    DATESMTD('Date'[Date])
)
```

📈 What Happens in a Chart?

- Accumulates day-by-day totals **within the same month**.
- Resets on the **first day of the next month**.
- Helps visualize trends and compare partial months.

📅 Quarter-to-Date (QTD) Measure

```
dax

Revenue QTD :=
CALCULATE(
```

```
    [Revenue],
    DATESQTD('Date'[Date])
)
```

□ **Use Case**

- Shows how the revenue has accumulated **since the beginning of the quarter**.
- Resets when the **quarter changes** (e.g., Q1 to Q2).

📅 Year-to-Date (YTD) Measure

```
dax

Revenue YTD :=
CALCULATE(
    [Revenue],
    DATESYTD('Date'[Date])
)
```

📊 Use Case

- Cumulatively adds revenue from **January 1st up to today**.
- Resets at the **start of a new year**.

📝 Best Practices & Notes

- Always use a **proper Date table** (`'Date'`) and ensure it's **marked as a Date Table** in Power BI.
- These measures are **dynamic** — they'll adjust automatically to date slicers and visuals.
- Use a **line or area chart with Date** on the X-axis (not just month names), so the accumulation behavior is clear.

□ Tip for Understanding

If you look at the values:

- **1st Jan** = Revenue of Jan 1st
- **2nd Jan** = Jan 1st + Jan 2nd
- ...
- **31st Jan** = Total of Jan
- **1st Feb** = Starts over
  And so on — same logic applies to quarters and years.

Section 25: CALCULATE vs TOTALYTD

Objective :

Concept	Description
CALCULATE + DATESYTD	Flexible way to compute Year-to-Date (YTD) using DAX by changing context.
TOTALYTD	A shorthand function that calculates YTD directly, including optional filter and year-end argument.
Key Difference	`CALCULATE` allows **multiple filters** and more complex expressions; `TOTALYTD` is limited in flexibility when adding multiple filters.

🛠️ Practice Exercises

◆ Exercise 1: Create a Measure Using `CALCULATE + DATESYTD`

Goal: Compute YTD revenue.

```dax
YTD Revenue (CALCULATE) =
CALCULATE (
    [Total Revenue],
    DATESYTD ( 'Date'[Date] )
)
```

✅ Add this to a line chart by month and confirm it accumulates.

◆ Exercise 2: Create a Measure Using `TOTALYTD`

Goal: Compute YTD revenue using simpler syntax.

```dax
YTD Revenue (TOTALYTD) =
TOTALYTD (
    [Total Revenue],
    'Date'[Date]
)
```

✅ Add this to the same chart and verify that the result matches Exercise 1.

◆ Exercise 3: Add Single Filter to Both

Goal: Add a filter where `Customer[Customer ID] > 200`.

```dax
YTD Revenue with Filter (CALCULATE) =
CALCULATE (
    [Total Revenue],
    DATESYTD ( 'Date'[Date] ),
    FILTER ( Customer, Customer[Customer ID] > 200 )
)
```

```dax
YTD Revenue with Filter (TOTALYTD) =
TOTALYTD (
    [Total Revenue],
    'Date'[Date],
    FILTER ( Customer, Customer[Customer ID] > 200 )
)
```

✅ Compare both results — they should match.

◆ Exercise 4: Add Multiple Filters — Show Why `CALCULATE` Wins

Goal: Add another filter: `Employee[Employee ID] > 15`

```dax
YTD Revenue with 2 Filters (CALCULATE) =
CALCULATE (
    [Total Revenue],
    DATESYTD ( 'Date'[Date] ),
    FILTER ( Customer, Customer[Customer ID] > 200 ),
    FILTER ( Employee, Employee[Employee ID] > 15 )
)
```

❌ **Try the same with `TOTALYTD`** — you'll realize it can't handle more than one filter directly.

🎯 Key Takeaway

Use `TOTALYTD` for **simple cases**, but prefer `CALCULATE` when you need **multiple filters** or **more flexibility**.

? Section 26: Other Time Intelligence Functions & DATESBETWEEN

✅ What You Learned

Topic	Key Point
CALCULATE + Time Intelligence	Most time functions return a **table** and are meant to be used inside `CALCULATE()` to modify filter context.
Self-explanatory functions	Many time functions like `NEXTDAY`, `PREVIOUSMONTH`, `DATESMTD`, etc., are intuitive — if they return a table, they can be used to filter context.
DATESBETWEEN Function	A versatile function that allows you to define a date range manually or dynamically inside a measure.

🔧 Syntax Refresher: `DATESBETWEEN`

dax

```
DATESBETWEEN(<dates>, <start_date>, <end_date>)
```

Returns a **table** with dates between `start_date` and `end_date` (inclusive).

🛠️ Practice Exercises

◆ Exercise 1: Basic Use of `DATESBETWEEN`

dax

```
Revenue from Specific Period =
CALCULATE (
    [Total Revenue],
    DATESBETWEEN (
        'Date'[Date],
        DATE(2017, 6, 30),
        MAX('Date'[Date])
    )
)
```

✅ What to do:

- Add this measure to a card or table.
- It should only calculate revenue **after June 30, 2017** to the **last available date**.

◆ Exercise 2: Explore with `NEXTDAY`

```dax
Revenue Next Day =
CALCULATE (
    [Total Revenue],
    NEXTDAY('Date'[Date])
)
```

✅ Apply this to a table and observe: It shifts the context to **the next day** for each row.

◆ Exercise 3: Try Out Other Simple Functions

Use any of the following with your `Revenue` measure:

```dax
-- Revenue in Previous Month
Revenue Prev Month =
CALCULATE (
    [Total Revenue],
    PREVIOUSMONTH('Date'[Date])
)

-- Revenue Year-to-Date for previous year
Revenue YTD Last Year =
CALCULATE (
    [Total Revenue],
    DATESYTD('Date'[Date], "31/12", PREVIOUSYEAR('Date'[Date]))
)
```

□ Tips for Using Time Intelligence

- Always confirm whether a function **returns a table** (for use in `CALCULATE`) or a scalar (single value).
- Use `CALCULATE()` to shift the **time context** to a specific period using these functions.
- Keep a **cheat sheet** of DAX Time Intelligence functions handy.
- For unusual date logic (like fiscal years), `DATESBETWEEN` gives full control.

📝 Summary Checklist

✅ Use `CALCULATE + DATESBETWEEN` for **custom date ranges**
✅ Use `NEXTDAY`, `PREVIOUSMONTH`, etc., for **quick filters**
✅ If a function returns a **table**, you can likely use it inside `CALCULATE()`
✅ Combine time functions with **manual dates** or dynamic functions like `TODAY()` or `MAX()`

🔢 Section 27: Practice – Rounding Numbers and Time in DAX

🎯 Goal:

Learn how to round **numbers** and **time values** using DAX functions like `ROUND`, `ROUNDDOWN`, `ROUNDUP`, `FLOOR`, and `CEILING`. You'll create calculated columns to explore each variation.

🛠️ Practice Setup:

📂 Dataset: Ensure your table has the following columns:

- `Numbers` – containing decimal numbers
- `Time` – containing time values (e.g., `09:34:00`, `12:15:30`)

📝 All exercises involve **calculated columns** — not measures.

◆ 1. Basic Rounding

Create a new calculated column:

```dax
Rounded to 1 Digit = ROUND([Numbers], 1)
```

✅ Try different precision levels:

- `0` → round to whole number
- `-1` → round to nearest ten
- `2` → round to hundredths

◆ 2. Force Rounding Up or Down

```dax
Always Round Down = ROUNDDOWN([Numbers], 1)

Always Round Up = ROUNDUP([Numbers], 1)
```

✅ These help when you need **control over rounding direction**.

◆ 3. Round to a Specific Multiple (e.g., $0.05)

```dax
Round to 0.05 = MROUND([Numbers], 0.05)
```

✅ Useful in **finance or currency** calculations (e.g., rounding to nearest nickel).

◆ 4. FLOOR and CEILING for Controlled Rounding

```dax
Floor to 0.1 = FLOOR([Numbers], 0.1)

Ceiling to 0.1 = CEILING([Numbers], 0.1)
```

✅ Use when you **always want to favor the customer** (FLOOR) or **the business** (CEILING).

◆ 5. Time Rounding – Round to the Nearest Hour

```dax
Round Time to 1 Hour =
MROUND([Time], TIME(1,0,0))
```

✅ Change the data type to **Time** if it shows a decimal.

◆ 6. Round Time to Nearest Interval

```dax
Round Time to 5 Min =
MROUND([Time], TIME(0,5,0))

Round Time to 30 Seconds =
MROUND([Time], TIME(0,0,30))
```

✅ Try with `FLOOR()` and `CEILING()` to control direction:

```dax
Floor Time to 5 Min = FLOOR([Time], TIME(0,5,0))
Ceiling Time to 5 Min = CEILING([Time], TIME(0,5,0))
```

🎯 Challenge Task

Create a column that rounds numbers:

- To the nearest `$0.25`
- But always **rounds down**

```
dax

Custom Round Challenge =
FLOOR([Numbers], 0.25)
```

Then do the same but **always rounds up** using `CEILING`.

□ Summary – Key DAX Functions Used

Function	Purpose
`ROUND()`	Rounds to nearest digit
`ROUNDDOWN()` / `ROUNDUP()`	Control direction of rounding
`MROUND()`	Round to a specific multiple
`FLOOR()` / `CEILING()`	Always round down or up to a multiple
`TIME()`	Used with rounding time (hour, minute, second)

🎯 Section 28: Custom Formatting with the `FORMAT()` Function in Power BI

📌 Purpose:

To display numbers, dates, and times in customized and readable formats using the `FORMAT()` function.

□ The `FORMAT()` Function

Syntax:

```
dax

FORMAT(<value>, "<format_string>")
```

- `<value>`: The number, date, or time you want to format.

- <format_string>: Predefined or custom formatting instruction.

✅ Two Ways to Format:

1️⃣ Predefined Formats

Format Type	Example	Description
"General Number"	1234.56 → 1234.56	Default decimal display
"Currency"	-1234 → ($1,234.00)	Adds currency symbol, commas, and brackets for negatives
"Percent"	0.85 → 85.00%	Multiplies by 100 and adds %
"Scientific"	123456 → 1.23E+05	Scientific notation
"True/False"	0 → FALSE, 1 → TRUE	Boolean style

📌 Usage Example:

```dax
Formatted Currency = FORMAT([Amount], "Currency")
```

2️⃣ Custom Formats

(Previewed, explained in next section)

- More control over structure (e.g., "#,##0.00").
- Supports special characters, conditionals, placeholders.

Dynamic Formatting with Logic

```dax
FormattedValue =
FORMAT(
    IF([Amount] < 0, 0, [Amount]),
    "Currency"
)
```

- Replaces negative values with 0, then formats.
- Demonstrates combining IF() + FORMAT() for business logic.

🔍 How to Explore Formats:

- Use Microsoft documentation: [FORMAT Function]
- Explore **predefined numeric**, **date**, and **time** formats.

💡 Key Takeaways:

- Use `FORMAT()` to improve readability and presentation.
- Predefined formats cover most needs: currency, percent, boolean, scientific, etc.
- Logic can be nested to create **conditional formatting**.
- Custom formats give full control (explored in the next section).

29. Custom Numeric Formatting in Power BI

We've already explored predefined numeric formatting in Power BI. However, **custom numeric formats** give you the flexibility to define exactly how numbers are displayed — tailored to your unique needs.

Understanding the Basics

Custom formatting relies on **special characters** to control how numbers appear:

- **0 (zero):** Digit placeholder — displays a digit or 0 if none exists.
- **# (hash):** Digit placeholder — displays a digit or nothing if none exists.
- **. (dot):** Decimal point placeholder.
- **% (percent):** Multiplies the number by 100 and displays the % sign.
- **, (comma):** Thousand separator.
- **; (semicolon):** Separates formatting for positive and negative numbers.
- \ **(backslash):** Escapes special characters, showing them literally (e.g., `%`, `#`).

Examples in Power BI

Let's look at how these symbols behave:

- **Using # and 0:**
  - Format: `#.##` — will show digits up to two decimal places if they exist, but hide zeros.
  - Format: `0.00` — forces display of two decimal places, even if they're zeros.
- **Thousand Separator:**
  - Format: `#,###` — adds commas for numbers in the thousands.
  - Example: `123456` → `123,456`.
- **Percentages:**
  - Format: `#%` — multiplies the number by 100 and adds a `%`.

        - Example: `0.25` → `25%`.
    - To **avoid multiplication** but still display `%`, escape it using `\%`:
        - Format: `#\%` → `0.25` → `0%`.
- **Appending Text:**
    - You can add text like `"K"` or `"ABC"` to the format:
        - Example: `#,##0.0,"K"` → `1500` → `1.5K`.

Escaping Characters

If you want to **display a character literally** that usually has special meaning:

- Use **backslash (\)**:
    - Format: `#\#` → Shows an actual hash (`#`) after the number.
    - Format: `0\%` → Shows `%` without multiplying the number.

Formatting Positive and Negative Numbers Differently

Use **semicolons (`;`)** to define separate formats for:

1. **Positive numbers**
2. **Negative numbers**

- Example: `#,##0.00;(#,##0.00)`
    - Positive → `1,234.56`
    - Negative → `(1,234.56)` (no minus sign, wrapped in parentheses)

To retain the minus sign:

- Example: `#,##0.00;-#,##0.00` → Negative → `-1,234.56`

Key Takeaways

- Custom formats let you **fully control how numbers appear**.
- Use placeholders and special symbols for **fine-grained control**.
- Escape characters using `\` to **safely display literal symbols**.
- Use `;` to **differently format** positive and negative values.

With some practice, **custom number formatting** becomes a powerful tool to enhance your Power BI visuals.

30. Custom Time Formatting in Power BI

After learning about custom formatting for numbers and phone numbers, let's now explore **custom time formatting** in Power BI using the `FORMAT()` function.

Custom time formatting allows you to present **hours, minutes, seconds, and even AM/PM** indicators in a way that fits your report's style or user expectations.

□ Key Symbols for Time Formatting

Symbol	Meaning	Example Format	Output Example
`h`	Hour (no leading zero)	`h`	`7`
`hh`	Hour (with leading zero)	`hh`	`07`
`n`	Minute (no leading zero)	`n`	`5`
`nn`	Minute (with leading zero)	`nn`	`05`
`s`	Second (no leading zero)	`s`	`9`
`ss`	Second (with leading zero)	`ss`	`09`
`tt`	AM/PM indicator (locale-based)	`tt`	`AM`, `PM`
`T`	Full time string (locale-based)	`T`	`7:45:20 AM`

⚠□ Use **`n` for minutes**, not `m`.
`m` is used for **months**.

🛠□ Example: Formatting Time Using DAX

Create a new calculated column:

```
dax

Formatted Time := FORMAT('Table'[Time], "hh")
```

This displays hours with leading zeros (e.g., `07`).

More formatting examples:

```
dax

-- Hours and minutes
FORMAT('Table'[Time], "hh:nn")          -- 07:05

-- Hours, minutes, and seconds
FORMAT('Table'[Time], "hh:nn:ss")       -- 07:05:30
```

🔤 Add Custom Text in Format

You can insert static text using **single quotes** `' '`:

dax

```
FORMAT('Table'[Time], "hh 'hours' nn 'minutes'")
```

Output: `07 hours 30 minutes`

◐ Working with AM/PM (12-hour format)

To include AM or PM:

dax

```
FORMAT('Table'[Time], "hh:nn tt")
```

Examples:

- `08:00 AM`
- `10:45 PM`

✅ `tt` works based on locale. You can also use `"AM/PM"` or `"am/pm"`.

□ Full Time Format with `"T"`

Using `"T"` gives a **complete time string**:

dax

```
FORMAT('Table'[Time], "T")
```

Output varies by region:

- `07:45:20 AM` (12-hour format)
- `19:45:20` (24-hour format)

🌐 Locale settings in Power BI affect how `"T"` and `"tt"` behave.

🗓️ Predefined Time Formats in Power BI

Instead of building custom strings, you can use Power BI's **predefined formats**:

dax

```
FORMAT('Table'[Time], "Short Time")      -- e.g., 14:35
FORMAT('Table'[Time], "Medium Time")     -- e.g., 2:35:20 PM
FORMAT('Table'[Time], "Long Time")       -- e.g., 2:35:20 PM (more detailed
in some locales)
```

🔚 Summary

- Use `FORMAT()` for **custom time formats** in Power BI.
- Use:
  - `h`/`hh` for hours
  - `n`/`nn` for minutes
  - `s`/`ss` for seconds
- Use `tt` or `"AM/PM"` for time indicators.
- Use `' '` to insert **text**.
- Use `"Short Time"`, `"Medium Time"`, `"Long Time"` for **locale-sensitive** defaults.
- Locale affects 12-hour vs 24-hour format.

Section 31

1. **Extracting Unique Dates**
   - Use `VALUES()` function to create a table with **distinct dates** from a column.
   - Example:

```
DAX

DatesExample = VALUES(Sales[OrderDate])
```

2. **Formatting Dates Using FORMAT()**
   - Create a **new column** in the `DatesExample` table.
   - Use `FORMAT()` to change how the date is displayed.
   - Examples:

```
DAX

LongFormat = FORMAT(DatesExample[OrderDate], "Long Date")
ShortFormat = FORMAT(DatesExample[OrderDate], "Short Date")
CustomFormat = FORMAT(DatesExample[OrderDate], "ddd, dd-mmm-
yyyy")
```

3. **Common Formatting Codes**

Code	Description	Example
`d`	Day number without leading zero	3
`dd`	Day with leading zero	03
`ddd`	Abbreviated weekday name	Mon
`dddd`	Full weekday name	Monday
`M`	Month number without leading zero	6
`MM`	Month with leading zero	06
`MMM`	Abbreviated month name	Jun

Code	Description	Example
`MMMM`	Full month name	June
`yy`	Last two digits of year	24
`yyyy`	Full year	2024
`ww`	Week of the year	27

□ Practice Exercise in Power BI

Step 1: Create a Date Table

Go to:
Modeling > New Table and use:

```
DAX

DatesExample = VALUES(Sales[OrderDate])
```

Replace `Sales[OrderDate]` with your actual date column.

Step 2: Add Custom Formatted Columns

Go to **Modeling > New Column**, and create the following:

1. **Long Date Format**

   ```
   DAX

   LongDate = FORMAT(DatesExample[OrderDate], "Long Date")
   ```

2. **Short Date Format**

   ```
   DAX

   ShortDate = FORMAT(DatesExample[OrderDate], "Short Date")
   ```

3. **Custom Format – 01-Jan-2024**

   ```
   DAX

   CustomDate1 = FORMAT(DatesExample[OrderDate], "dd-mmm-yyyy")
   ```

4. **Day Name and Month**

   ```
   DAX

   CustomDate2 = FORMAT(DatesExample[OrderDate], "dddd, MMMM")
   ```

5. **Week of Year**

```
DAX

WeekOfYear = WEEKNUM(DatesExample[OrderDate])
```

✅ Your Task

Create the following columns in your Power BI date table:

Column Name	DAX Formula
`DayName`	`FORMAT(DatesExample[OrderDate], "dddd")`
`MonthName`	`FORMAT(DatesExample[OrderDate], "MMMM")`
`ShortYear`	`FORMAT(DatesExample[OrderDate], "yy")`
`MonthYear`	`FORMAT(DatesExample[OrderDate], "mmm-yyyy")`
`CustomReadable`	`FORMAT(DatesExample[OrderDate], "dddd, dd mmmm yyyy")`

📝 Notes

- Always format dates for **user-friendly visuals**.
- Use **custom formats** for reports needing localization or business-specific formatting.
- You can use this approach to build a **complete date dimension table** with fiscal calendars too.

Section 32.A – Introduction to Quick Measures in Power BI

As we move forward in our Power BI journey, it's time to introduce a helpful feature: **Quick Measures**.

What are Quick Measures?

Quick Measures allow you to **generate predefined DAX calculations** without needing to write the full formula yourself. You'll find them under the **Home** ribbon in Power BI. This is particularly useful for beginners or when you need a quick calculation template.

🛠 How to Create a Quick Measure

1. Go to the **Home** tab.
2. Click on **Quick Measure**.
3. Choose a calculation type, such as:
   - Aggregation
   - Filters
   - Time intelligence (e.g., Year-to-Date)
   - Running totals, etc.
4. Provide the necessary input fields, for example:
   - Select the **base measure** (e.g., Revenue).

- Choose a **Date field** (typically from your Dates table).

✅ That's it! Power BI automatically generates a measure and adds it to your **Fields pane**.

Learning from the Generated DAX

A powerful benefit of Quick Measures is that **you can inspect the generated DAX formula**:

- Understand Microsoft's DAX structure.
- Edit the formula to customize it.
- Use it as a **learning tool or inspiration** to build your own DAX knowledge.

⚖️ Pros and Cons of Quick Measures

Advantage	Disadvantage
Great for learning DAX syntax	The auto-generated formulas can be more complex than needed
Fast setup for common calculations	May not be optimal or tailored for your specific use case
Can be customized and extended	Over-reliance may hinder DAX skill development

When to Use Quick Measures

- ✅ **Yes**: Use them to **learn, experiment**, or as a **starting point**.
- ❌ **No**: Don't rely on them blindly without understanding the logic behind the measure.

If you're already comfortable with DAX, you'll likely find it **faster and more precise to write your own measures**. However, for those still learning, Quick Measures offer a structured and safe environment to explore DAX capabilities.

Section 32.B – Using Quick Measures: Filters

After understanding how Quick Measures work, let's now explore one of the simplest and most practical types: **Quick Measures using Filters**.

Creating a Filter-Based Quick Measure

Here's how to create a filter-based quick measure step-by-step:

1. **Click** on the **Quick Measure** option in the **Home** ribbon.
2. **Choose** a calculation type under **Filters**.
3. **Provide input values:**
    - **Base value**: This can be a measure (e.g., `Revenue`) or an aggregated column (e.g., `SUM(Sales)`).
    - **Filter field**: Drag a field (like `Location[State]`) into the filter area.

    - **Filter values**: Select one or multiple values (e.g., "California", "Florida").
        - To select multiple values, **hold Ctrl** while clicking.
4. **Click OK** to generate the measure.
5. The new measure will appear in the **Fields pane**.

📈 Understanding the Generated Measure

Once the Quick Measure is created, Power BI generates a DAX formula using:

```
dax

CALCULATE (
    [Revenue],
    'Location'[State] IN { "California", "Florida" }
)
```

- This expression calculates **Revenue** only for the selected states.
- When placed in a visual, all states will show the **combined value** unless filtered further.

You may also encounter:

```
dax

CALCULATE (
    [Revenue],
    FILTER (
        'Location',
        'Location'[State] IN { "California", "Florida" }
    )
)
```

- This alternative uses the `FILTER()` function explicitly, which gives you more flexibility in complex scenarios.

➖ Difference from Filtered Value

You can also create a **"Difference from Filtered Value"** quick measure. Here's how:

1. Choose **"Difference from filtered value"** from the Quick Measures panel.
2. Select your **base measure** (e.g., `Revenue`).
3. Choose your **filter field** and filter values again (e.g., California and Florida).
4. Decide how to handle blanks:
    - ✅ **Treat as blank** – shows no value where there's no base measure data.
    - ✅ **Treat as zero** – calculates difference assuming 0 for missing values.

□ What Does the Measure Do?

Power BI uses DAX with variables like this:

```
dax

VAR BaseValue = CALCULATE ([Revenue], 'Location'[State] IN { "California",
"Florida" })
VAR MeasureValue = [Revenue]
RETURN
IF (
    ISBLANK(MeasureValue),
    BLANK(),
    MeasureValue - BaseValue
)
```

- `BaseValue`: Revenue from California and Florida combined.
- `MeasureValue`: Revenue from each state (or total, depending on context).
- The `IF` logic handles missing values gracefully.

If you choose **"Treat blanks as zero"**, the formula skips the `IF()` check and simply subtracts the `BaseValue` directly—resulting in constant negative values where no data is available.

□ Best Practices

- ✅ Use **Quick Measures** to prototype and **learn the DAX logic**.
- 🔁 **Edit** the generated formulas to suit your business needs.
- □ Understand the difference between **calculated context** (with or without filters).
- ✅ Use the **"Home Table" setting** to organize your measures correctly if they end up in the wrong table.

🔄 Summary

Quick Measures with filters:

- Are a great entry point for understanding `CALCULATE`, `FILTER`, and `IN`.
- Help you explore DAX patterns by **reverse-engineering the generated formulas**.
- Should be used **intelligently**—don't rely on them blindly.
- Are useful for creating comparison metrics like **"difference from filtered value"**.

Section 32.C – Time Intelligence Quick Measures

Time Intelligence is one of the most powerful capabilities in Power BI—and Quick Measures offer a fast way to apply these calculations without writing DAX manually. In this section, we'll explore how to use Quick Measures for **Year-to-Date (YTD)**, **Year-over-Year (YoY)** comparisons, and discuss the **limitations** and **best practices** around them.

🕒 1. Creating a Year-to-Date (YTD) Measure

To calculate a Year-to-Date total using a Quick Measure:

1. Open the **Quick Measure** dialog.
2. Choose **"Year-to-date total"** as the calculation type.
3. Select a **base measure** (e.g., `[Revenue]`).
4. Select a **date column** from your **Date Table**.
5. Click **OK**.

The measure will be created (by default) in the currently selected table. If it's the wrong table (e.g., `Customers`), move it to the correct one by updating the **Home Table**.

📝 **Note**: Power BI automatically appends numbers to duplicate measure names (e.g., "Year-to-date", "Year-to-date 2").

⚠️ 2. Limitation: Must Use Date Hierarchy

When using a Time Intelligence Quick Measure, you may encounter this error:

"Time intelligence quick measures can only be grouped or filtered by the Power BI-provided date hierarchy."

This means the default Quick Measure **only works** with the default **date hierarchy** (Year → Quarter → Month → Day). If you try to use individual date fields (e.g., `Date` column only), the visual may break.

□ 3. Cleaning the Auto-Generated DAX

Let's look at how to simplify and improve the auto-generated DAX.

Instead of Power BI's complex output (which checks if the hierarchy is active), a simpler version like this works in both hierarchy and flat views:

```
dax

YTD Revenue =
TOTALYTD(
    [Revenue],
    'Date'[Date]
)
```

- ✅ Works with or without the hierarchy.
- ✅ Easier to read and maintain.
- ❌ Power BI’s default version includes unnecessary complexity (e.g., `IF` statements checking for hierarchy membership).

Best Practice: Simplify the generated code for clarity and flexibility.

🔁 4. Shortcut: Create Quick Measure from an Existing Measure

Instead of opening the Quick Measure dialog from the ribbon, right-click any measure and choose **New Quick Measure**. This auto-fills the base measure field, saving time.

Example: Right-click `[Revenue]` → New Quick Measure → Choose "Year-over-year change" → Continue setup.

📉 5. Year-over-Year (YoY) Change

To calculate YoY change:

1. Choose **“Year-over-year change”** from Quick Measures.
2. Set the base measure (e.g., `[Revenue]`).
3. Provide the **date column** from your date table.
4. Choose the number of periods (e.g., 1 year back).

📊 This measure returns **percentage growth**:

- 100% = doubled from last year
- 0% = same as last year
- -50% = half of last year

Power BI uses DAX like this:

```
dax

VAR PreviousYear =
    CALCULATE(
        [Revenue],
        SAMEPERIODLASTYEAR('Date'[Date])
    )
VAR Current = [Revenue]
RETURN
DIVIDE(Current - PreviousYear, PreviousYear)
```

💡 **Why variables?**
Using `VAR` improves performance and readability—it's a recommended practice.

□ You can **clean** the generated measure by removing:

- Redundant `IF` checks for hierarchy
- Dot notation for hierarchy levels (e.g., `Date Hierarchy.Year`)

🔄 6. YoY Change Works at All Levels

YoY comparisons adapt based on context:

- If viewing **years**, it compares entire years.
- If viewing **months**, it compares month-to-month YoY.
- If viewing **days**, it compares exact days across years.

This flexibility makes it an excellent tool for both executives and analysts.

✅ 7. Summary: Best Practices for Time Intelligence Quick Measures

Best Practice	Description
✅ Use a proper **Date Table**	Ensure it's marked as a Date Table.
✅ Use **TotalYTD**, **SAMEPERIODLASTYEAR**	These are core DAX functions for time intelligence.
⚠□ Simplify generated code	Remove unnecessary IFs and hierarchy checks.
⚠□ Watch for hierarchy dependency	Avoid errors by testing visuals with both hierarchy and flat date fields.
💡 Prefer variables (`VAR`)	Makes code clearer and reusable.

Section 32 D: Understanding Rolling Average in Power BI

In this section, we dive into the mechanics behind **Rolling Averages** in Power BI — a useful but slightly complex calculation that enhances report insights by smoothing out data fluctuations over time.

Objective

To **understand how Power BI calculates rolling averages** and how to recreate or modify such calculations for custom reporting.

Step-by-Step Breakdown

1. **Base Values and Time Periods**:
   - The rolling average is computed over a set time frame (e.g., **last 5 or 10 days**, or **last 2 months**).
   - The base metric for averaging is **Revenue**.
   - The rolling window can be customized to include days **before** and **after** the current context date.
2. **Visualization**:
   - Start by adding the **rolling average measure** to your visual (e.g., line chart or table).
   - Compare the rolling average with the actual revenue to observe smoothing effects.
3. **Understanding the Measure**:
   - The formula includes familiar DAX functions like:
     - `AVERAGEX`
     - `DATESBETWEEN`
     - `DATEADD`
     - `CALCULATE`
     - `LASTDATE` (new for many users)
4. **Key Function - LASTDATE**:
   - Returns the **last non-blank date** in the current filter context.
   - Essential for defining the **end date** of the rolling window.
5. **How AVERAGEX is Used**:
   - The **table** parameter: `DATESBETWEEN(...)` that generates a list of dates for the rolling window.
   - The **expression**: `Revenue` values corresponding to those dates.
   - For example, if the context is **11th May**, the table includes the **last 5 days** up to the 11th May, and `AVERAGEX` calculates the average of those revenue entries.
6. **Customization Possibilities**:
   - You can change the **time granularity** (from days to months).
   - Adjust the **rolling period** (e.g., from 5 to 10 days).
   - Include **future periods** by modifying the `DATESBETWEEN` range.

Final Notes

- The rolling average helps in **trend analysis**, especially for smoothing erratic daily revenue data.
- Once understood, this measure gives you **full control** to modify and adapt based on your reporting needs.
- Being able to construct such a measure manually is a valuable skill for **advanced Power BI development**.

Section 32 E: Aggregate Per Category in Power BI

In this section, we explore how **aggregate per category** works in Power BI using quick measures like **average**, **maximum**, **minimum**, and **variance**.

Objective

To understand how to calculate and properly use **aggregated values (e.g., average revenue)** grouped by a specific **category**, such as **Customer**, **State**, or **Year**.

Step-by-Step Explanation

1. **Selecting the Quick Measure:**
    - Navigate to **Quick Measures**.
    - Choose **"Average per Category"** (or another aggregate like max/min/variance).
    - Select the base value (e.g., **Revenue**).
    - Then choose the **category to group by**, e.g., **Customer**.
2. **Initial Misuse Case – Wrong Context:**
    - If you use **"Average per Customer"** and then visualize this **alongside individual customers**, the result seems redundant:
        - Each row shows the revenue for that **single customer**.
        - Since it's one customer per row, the "average" is just that customer's revenue — no real aggregation happens.
        - Thus, both "Revenue" and "Average per Customer" values are **identical**.
3. **Correct Usage – Different Context:**
    - To see meaningful insights, use a **different category** in the visual than the one you calculated the average by:
        - For example, show the **"Average Revenue per Customer"** by **Year** or **State**.
        - This shows:
            - In 2017: average revenue **per customer** for that year.
            - In 2018: likewise, average per customer for 2018.
        - Or:
            - In **Arizona**: average revenue per customer.
            - In **California**: same logic.
        - Totals across categories now show insightful averages.
4. **Takeaway**:
    - **Do NOT use the same category you are aggregating by in the visual**.
        - If you calculate "average per customer", don't display customer in the same visual — that defeats the purpose.
    - Use **another context** (e.g., state, year) to observe **true aggregation behavior**.

Section 32F Summary – **Star Rating Quick Measure in Power BI using DAX**

◆ Objective

To create a dynamic **star rating** visualization based on a numerical metric (e.g., revenue), using **Quick Measures** in Power BI.

◆ Steps Explained

1. Creating a Star Rating

- Use **Quick Measures** in Power BI.
- Choose a **base value**, e.g., `Revenue`.
- Set parameters:
  - **Max stars**: Default is 5.
  - **Minimum value**: Revenue below this → 0 stars.
  - **Maximum value**: Revenue above this → 5 stars.

2. How It Works Internally (DAX Logic Breakdown)

The generated DAX has 5 parts:

- **Variables**
  - `Number of stars` → usually 5
  - `Min value` → revenue below which 0 stars
  - `Max value` → revenue above which 5 stars
  - `Base value` → metric being rated (e.g., revenue)
  - `Normalized value` = (Basevalue–Min)/(Max–Min)(Base value – Min) / (Max – Min)(Basevalue–Min)/(Max–Min)
    Clamped between 0 and 1
- **Star Calculation**

  ```
  dax

  Rounded Stars = ROUND(Normalized_Value * Max_Stars, 0)
  ```

- **Unicode Character for Stars**
  - ⭑ (filled): `UNICHAR(9733)`
  - ☆ (empty): `UNICHAR(9734)`
- **Building the Star String**

  ```
  dax

  REPT(⭑, Rounded Stars) & REPT(☆, Max Stars - Rounded Stars)
  ```

◆ Why Understanding DAX Matters

If you don't understand DAX:

- You may assume `Revenue = Price * Quantity`, but might incorrectly write:

  ```dax
  SUM(Sales[Price]) * SUM(Sales[Quantity])
  ```

  → This **overestimates** revenue.

- Correct approach is:

  ```dax
  SUMX(Sales, Sales[Price] * Sales[Quantity])
  ```

◆ Takeaway

- Quick measures are **great starting points**, but to **customize or debug**, understanding DAX is essential.
- Star ratings can be modified: number of stars, different Unicode symbols, and logic.
- DAX gives full control for precise and visual metrics.

Project :

Mini Power BI Project Steps

1. **Build a Key Measures Table:**
   - Create a new table called `Key Measures` to logically group important calculated measures (like Net Revenue, Profit, etc.).
2. **Create Relationships:**
   - Connect your `Sales` table to the `Product` table using `ProductID`, and similarly link other relevant dimension tables (like Date, Region).
3. **Add Basic Measures using DAX:**
   - `Gross Revenue = SUM(Sales[Amount])`
   - `Net Revenue = SUMX(Sales, Sales[Quantity] * RELATED(Product[Net Price]))`
   - `Profit = [Net Revenue] - SUMX(Sales, Sales[Cost])`
4. **Create Date Range Filter Table:**
   - Use `Enter Data` to manually add a new table like:
     - `Time Filter Table` with columns: `Time Filter` and `Time Value`
     - Examples:

Time Filter	**Time Value**
3 Days	3
7 Days	7
30 Days	30
1 Year	365
Total	3000

5. **Create Revenue Type Filter Table:**
   - Another table using `Enter Data` with values like `Gross`, `Net`, `Profit`
   - Add a sort column if needed:

Revenue Type	**Sort Order**
Gross	1
Net	2
Profit	3

6. **Sort Columns Accordingly:**
   - Sort `Revenue Type` by `Sort Order`
   - Sort `Time Filter` by `Time Value` if needed
7. **Final Measure Incorporating Filter Selection:**

```
Final Revenue =
VAR SelectedRevenue = MAX('Filter Revenue Table'[Revenue Type])
 VAR SelectedTime = MAX('Time Filter Table'[Time Value])
 VAR DateFilter = MAX('Sales'[Date]) >= TODAY() - SelectedTime

 RETURN
 IF(
     SelectedRevenue = "Gross",
     CALCULATE([Gross Revenue], DateFilter),
     IF(
         SelectedRevenue = "Net",
         CALCULATE([Net Revenue], DateFilter),
         CALCULATE([Profit], DateFilter)
     )
 )
```

23. **Use Slicers:**
    - Add slicers for Revenue Type and Time Range in your Power BI report
24. **Add Visuals:**
    - Add cards or charts to display the final revenue metric based on filters
25. **Format the Final Measure:**
    - Move it to `Key Measures` table
    - Apply currency formatting

Power BI Report Creation Walkthrough:

11. **Customize Filters and Layout:**
    - Change slicer orientation to *Horizontal* for better layout.

- Duplicate the filter and add another field (like Revenue Filter) to it.

12. **Create First Visual:**
    - Use the `Final Revenue` measure.
    - Visualize customers in a bar chart.
    - Change default color under *Data colors* (e.g., dark blue to light gradient).
    - Turn off axis titles for cleaner visuals.
    - Apply basic formatting and alignment.
13. **Reuse Visuals Efficiently:**
    - Copy and paste the visual, adjust the Axis and Title as needed.
    - Maintain clean alignment for consistency and readability.
14. **Create Top-N Visual (e.g., Products):**
    - Add another visual for products.
    - Apply *Top N Filter* (Top 5) based on `Final Revenue` measure.
    - Reduce text size for space.
15. **Map Visualization by State:**
    - Add a map visual to show revenue by state.
    - Use slicer to filter only "California" initially.
    - Copy and paste the visuals and adjust the state filter (e.g., "Florida", "Arizona").
16. **Visual Grouping and Background:**
    - Use gray rectangle backgrounds behind grouped visuals (e.g., for each state).
    - Turn off background of individual visuals so the rectangle is visible.
17. **Fine Tune Alignment:**
    - Use the *Format > General* tab to align visuals precisely.
    - Adjust titles, font styles, and sizes for readability and aesthetics.
18. **Wrap-Up:**
    - Review overall layout.
    - Ensure slicers work correctly with visuals.
    - Polish formatting to create a clean and interactive report page.