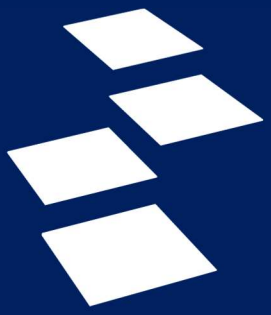
DAX Basics

Sherpa Workbook



JOURNEYTEAM

Guide to Workbook

This workbook contains exercises for the JourneyTEAM DAX Basics Sherpa.

The associated Power BI Desktop (.pbix) and data files with this workbook can be downloaded at https://bit.ly/DAXBasicsSherpa.

Once you download the data you will need to change the data source in Power BI to your local machine. To accomplish this, follow the steps below:

1. In Power BI Desktop, go to File
2. Click *Options and settings*
3. Click *Data source settings*
4. In the new window, select the data source and click *Change Source*
5. Click *Browse*
6. Navigate to where you downloaded the data file *Adventure Works LT Data Set*
7. Click *Open*
8. Click *OK*
9. Close the window
10. Refresh your data

Each section in this workbook contains step-by-step instruction followed by a solutions subsection called *Check Your Code*. To get the most out of this workbook, wait to look at the *Check Your Code* section until you have tried to write the DAX on your own.

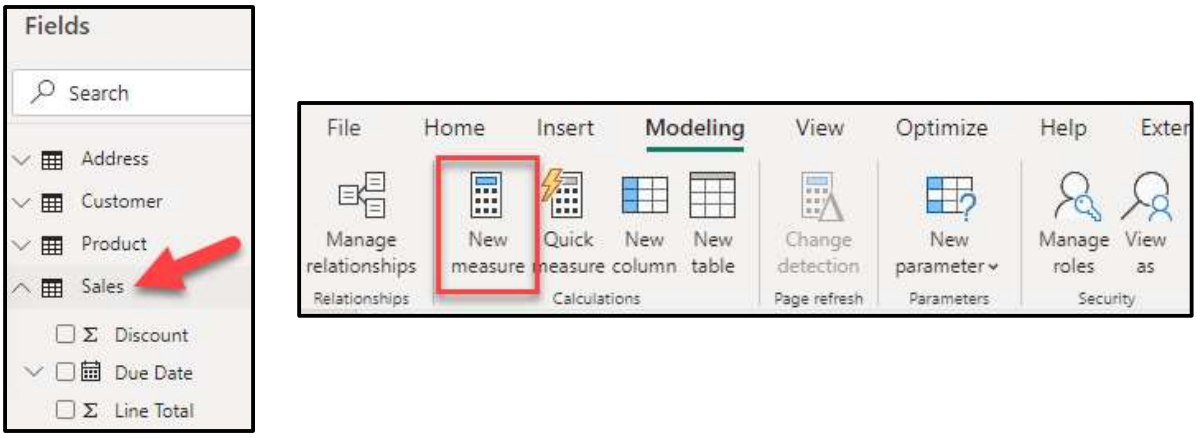
Enjoy!

Measures

Measures are the best way to calculate numbers with your data because they are calculated *as needed* rather that at data refresh, therefore measures take up less space in your model. And smaller data models equal better performance most of the time.

Measures are similar to formulas in Excel with an important difference; measures require context.

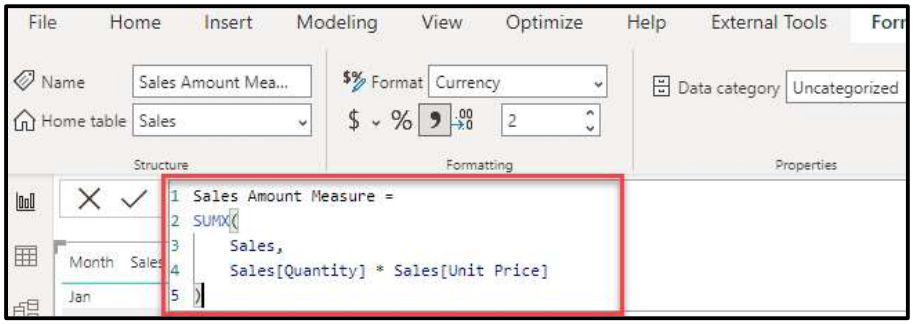
To begin, select the table in the fields pane you wish to add the column to, in this case the *Sales* table, go to the *Modeling* tab, and select *New Measure*.



1. Then type **`Sales Amount Measure =`** Sales[Quantity] * Sales[Unit Price], quantity multiplied by the unit price.

Note an error is returned. That is because when referring to every row in a table like we are in this example, Measures need aggregator functions such as **`SUMX`** or **`AVERAGEX`**. We will cover functions like these a little later.

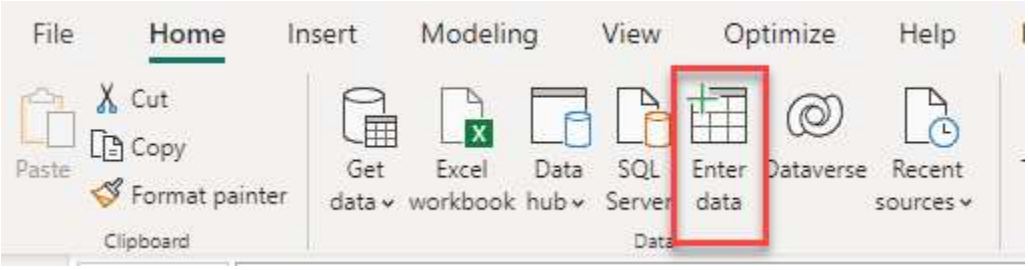
2. Include a **`SUMX`** before the code, add the table reference to *Sales*, and voila, you have a measure.



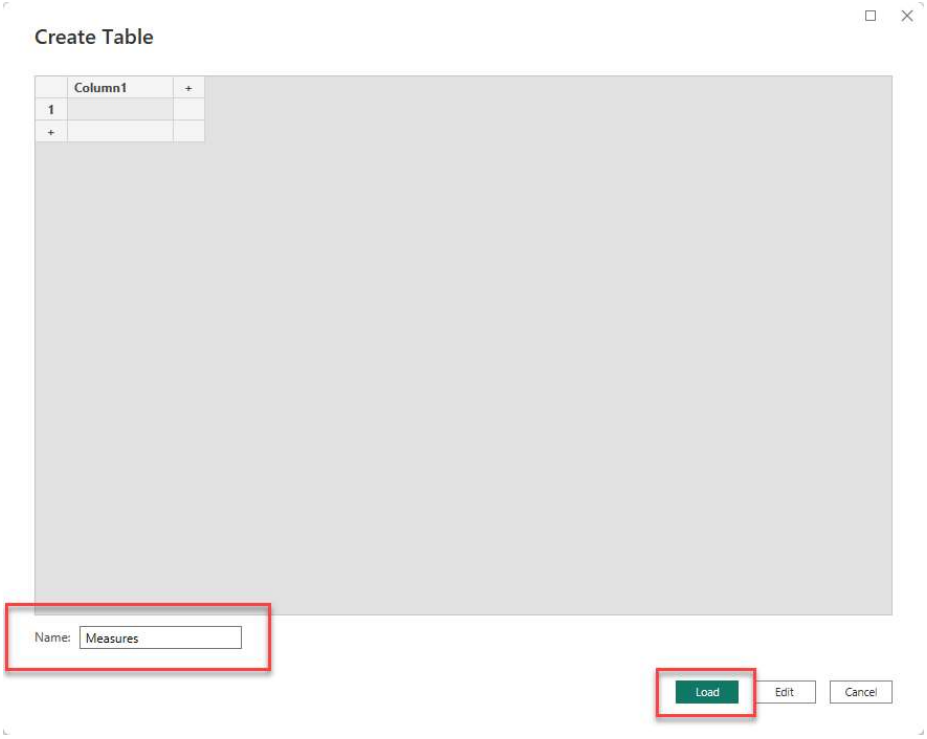
Measures are powerful, but that doesn't mean they should be used in every instance. Measures are NOT good for:

- Categorizing data
- Filtering data
- Making up for a poor data model

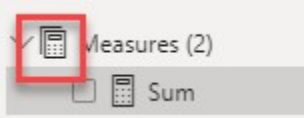
Organizing measures is best done in a measure table. To create a measure table, click on the Enter data button in the ribbon.



Then give the table a name and do NOT enter any data. Click the Load button after naming the table.



Add a measure to the table, hide the only column and the table becomes a special Measure Table indicated by this icon:



Measures can also be organized into folders.

*If possible, **measures** should always be used when computing numbers or performing equations (as opposed to calculated columns). Only the formula is stored in memory saving space and refreshing faster*

Check Your Code

```
Step 3

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

Functions: Aggregators

Aggregators in DAX are as close to Excel functions as you can get. Each aggregator function takes a column as an input and returns a single value within the given filters. Let's create some measures to see how aggregators work in Power BI.

1. First, create a new measure in the *Sales* table called *Sum of Line Total*. See if you can type the DAX code on your own, and then check your code.

Let's do the same with the **`AVERAGE`** and **`MIN`** functions.

2. Create a measure in the *Product* table called *Min Date* that finds the minimum start date for each product.

Note that there are a couple ways of adding a new line in the DAX formula editor:

Shift+Enter will move your cursor, and any content after your cursor on the same line, to a new line, properly indented with the code above it.

Alt+Enter will do the same thing as Shift+Enter, but does *not* add the proper indentation

Ctrl+Enter moves your cursor to the next line, keeping the appropriate indentation, but does not move any content which follows the cursor.

3. Then create another measure in the *Sales* table called *Average of Line Total* that computes the average of the Line Total column.

Aggregators are most similar to Excel functions, but remember that DAX is a tabular language so you cannot reference a specific cell or range of cells.

Check Your Code

Step 1

```
Sum of Line Total = SUM( Sales[Line Total] )
```

Step 2

```
Min Date = MIN( 'Product'[Start Date] )
```

Step 3

```
Average of Line Total = AVERAGE( Sales[Line Total] )
```

Functions: Iterators

Iterators are very similar to aggregators but include even greater functionality. Aggregators can only take a column as an input while iterators can take expressions as an input. An expression is DAX code that contains a logical or mathematical evaluation rather than a single column reference. For example 2 + 2 is an expression. The expression is then iterated row by row in the given filter context (more on filter context later).

To see the benefit of iterators, recall the *Sales Amount Measure* we created earlier. The code allows us to multiply sales quantity by unit price; something the **SUM** function does not allow, because it only takes as it's input a single column (instead of an expression, e.g.).

1. For further practice, create a new measure in the *Product* table called *Product Total Sales* that sums the Line Total for each product.

Aggregator functions have sister iterator functions. SUM and SUMX. AVERAGE and AVERAGEX. Using an iterator, we can find the first sale date of each product.

2. Create a new measure in the *Product* table called *First Sale Date* that finds the first time a product was ordered.

Remember, iterator functions evaluate an expression row by row. These functions always include "X" at the end - **SUMX**, **AVERAGEX**, *and so on.*

Check Your Code

Step 1

```
Product Total Sales =
SUMX(
    Sales,
    Sales[Line Total]
)
```

Step 2

```
First Sale Date =
MINX(
    Sales,
    Sales[Order Date]
)
```

Remember that "if it's not formatted, it's not DAX"

Functions: Counters

Count functions are also very similar to excel functions, with a few additional abilities. Let's see how all the different count functions work.

Here is a list of all the counting functions

`COUNT`	`COUNTBLANK`	`DISTINCTCOUNT`
`COUNTA`	`COUNTROWS`	`DISTINCTCOUNTNOBLANK`
`COUNTAX`	`COUNTX`	`DISTINCTCOUNT`

First, say we want to find out how many product models we have.

1. Create a measure in the *Products* table called *# of Product Models* that counts the number of product models using one of the counting functions above.

Let's also create a few other measures so we can compare them.

2. In the *Products* table create a measure called *# Products* that counts the number of products and another measure called *# of Current Products* which counts the number of products currently for sale.

Notice that the *# Products* and *# Product Models* match, which is not correct. What is wrong with the DAX code? The issue is the **`COUNT`** function just looks for non-blank values and doesn't take into account that the same value may be represented more than once (ex. A Product Model may be related to multiple Product, so we would be over-counting Products if we used the **`COUNT`** function).

3. Instead, try to use the **`DISTINCTCOUNT`** function in a new measure called *# Distinct Product Models*. Note though, **`DISTINCTCOUNT`** also counts includes blanks values as a value.

Always double check your counter functions to make sure you are getting the value you expect. Some counter functions include blanks. Also, ***`COUNT`*** *and* ***`COUNTA`*** *are identical in DAX for all the data types except Boolean.* ***`COUNTA`*** *can operate on a Boolean data type, whereas* ***`COUNT`*** *cannot do.*

Check Your Code

Step 1

```
# Product Models =
COUNT( 'Product'[Product Model] )
```

Step 2

```
# Products =
COUNT('Product'[Product] )

# Current Products =
COUNTBLANK( 'Product'[End Date] )
```

Step 3

```
# Distinct Product Models =
DISTINCTCOUNT( 'Product'[Product Model] )
```

Functions: Error Management

Errors will happen as you write in DAX, but DAX has several helpful ways for you to account for and manage errors that result from calculations.

Say, we want to know what the cost per ounce (oz) is for each Product

1. Create a calculated column in the *Product* table so we can see the result (a measure is best practice for this). Name the column *Cost per oz* and divide the *Product[Cost]* by the *Product[Weight]*.

Notice that the result is infinity for some products. Why is this? The *Product[Weight]* column does not have a value for all products so the calculation is dividing by 0, hence the "infinite" result.

DAX has built in functions to identify when a calculation results in an error. One of these functions is `ISERROR` which returns `TRUE` if the calculation results in an error and `FALSE` otherwise.

2. Create a calculated column in the *Product* table called *Error* and use the `ISERROR` function using *Cost per oz* as the input column.

Notice how for every infinite result in the *Cost per oz* calculated column the *Product[Error]* columns is `TRUE`.

We can also use the `IFERROR` function to return a blank when the calculation result is an error.

3. Create a calculated column in the *Product* table called *Cost per oz 2* using the `IFERROR` function.

The `DIVIDE` function automatically returns a blank (or another selected value) when the denominator is zero.

4. Use the `DIVIDE` function to create another calculated column called *Cost per oz with Divide*.

Always use the DIVIDE function to avoid unnecessary errors when your denominator is blank or zero.

Check Your Code

```
Step 1

Cost per oz = 'Product'[Cost] / 'Product'[Weight]


Step 2

Error = ISERROR( 'Product'[Cost per oz] )


Step 3

Cost per oz 2 =
IFERROR(
    'Product'[Cost] / 'Product'[Weight],
    BLANK()
)

Step 4

Cost per oz with Divide = DIVIDE( 'Product'[Cost],
'Product'[Weight] )
```

Functions: Conditional

Conditional functions allow filtering within a formula that replace or add to the filters from outside the formula. For this example, say we want to only see the sales orders that had more than 2 items and were over $1000.

1. Create a measure called *Large Sales 1* that calculates the revenue for only orders with 2 or more items AND were over $1000 (hint: use **`SUMX`** and **`FILTER`**).

Alternatively, we may want to see all sales that had more than 2 items or were over $1000. We would need to use the OR function in this scenario. Compare the results.

2. Create a measure called *Large Sales 2* that calculates the revenue for only orders with 2 or more items OR were over $1000.

Note that you can also replace the **`AND`** and **`OR`** functions with **`&&`** and **`||`** respectively.

The **`IN`** function is a cleaner way to get the same results as the **`OR`** function if you are only filtering from one column.

3. Create a measure called *Large Sales 1* that calculates the revenue for only orders with 1 OR 2 items as their quantity.

Use the operators && and || in place of the AND and OR functions.
Otherwise you will need to use nested functions, and that gets messy.

Check Your Code

Step 1

```
Large Sales 1 =
SUMX(
    FILTER(
        Sales,
        AND( Sales[Quantity] > 2, Sales[Line Total] > 1000 )
    ),
    Sales[Quantity] * Sales[Unit Price]
)
```

Step 2

```
Large Sales 2 =
SUMX(
    FILTER(
        Sales,
        OR( Sales[Quantity] > 2, Sales[Line Total] > 1000 )
    ),
    Sales[Quantity] * Sales[Unit Price]
)
```

Step 3

```
Large Sales 3 =
SUMX(
    FILTER(
        Sales,
        Sales[Quantity] IN { 1, 2 }
    ),
    Sales[Quantity] * Sales[Unit Price]
)
```

Functions: Date

Date functions are one of the most useful and powerful functions in DAX. We will only be scratching the surface in this course, but other courses will explore the full functionality of the Time Intelligence features and functions in Power BI.

For now, we will introduce the basic date functions: **YEAR**, **MONTH**, **WEEKDAY**, and **DATE**. As we will see later, these functions are particularly useful for creating a date table.

1. Create three calculated columns in the *Sales* table that parse out the year, month, and weekday of the *Sales[Order Date]*.
2. Use the **DATE** function to get the first day of the order date month (you can also use the **STARTOFMONTH** function).

Each of these date functions returns a number. To return the text value of a month or weekday, use the FORMAT function.

Check Your Code

Step 1

```
Year = YEAR( Sales[Order Date] )

Month = MONTH( Sales[Order Date] )

Weekday = WEEKDAY( Sales[Order Date] )
```

Step 2

```
First Day of Month =
DATE( YEAR( Sales[Order Date] ), MONTH( Sales[Order Date] ), 1 )
```

Functions: Table Functions

Up to now we have mostly been learning about functions in DAX that take columns as inputs. A few functions have taken tables as inputs which is why we call them "table functions". Table functions are useful because they allow you to modify the filtering that a formula uses to evaluate its expressions. Using table functions allows you to customize your data to show what is most important to you despite or in addition to the filters displayed on the page.

We first will learn about the **FILTER** function.

1. Create a measure called *US & Canada Sales* that calculates the sales only for the US and Canada using the **FILTER** function. Hint: You will need to use **RELATED** as well.

We then need to see the total sales regardless of the filters that are applied on a report page. To ignore all filters, we use the **ALL** function.

2. Create a measure in the *Sales* table called *All Sales* that shows the total sales despite what filters are selected.

That is useful to know, but what if we want certain filters to change the value displayed by our *All Sales* measure? We need to use the **ALLSELECTED** function. This function ignores filters within a formula's context but does apply filters from outside the formula's context.

3. Create a new measure in the *Sales* table called *All Selected Sales* that changes based on the selected filters on the report page.

The next two functions perform produce almost the same result. **VALUES** returns the total number of distinct values from a column, including a blank to account for unmatched values between tables. The **DISTINCT** function ignores unmatched values. In our data, not all customers in the *Customer* table have an order.

4. Create two measures in the *Sales* table that count the number of customers using the **VALUES** function in one measure and the **DISTINCT** function in the other. When you create a matrix with the measures and *Product[Product Category]*, notice that the **DISTINCT** counts one less than the **VALUES** calculation.

Compare your code with the code on the next page.

Check Your Code

Step 1

```
US & Canada Sales =
SUMX(
    FILTER(
        Sales,
        RELATED( 'Address'[Country/Region] ) IN { "United States",
"Canada" }
    ),
    Sales[Quantity] * Sales[Unit Price]
)
```

Step 2

```
All Sales =
SUMX(
    ALL( Sales ),
    Sales[Quantity] * Sales[Unit Price]
)
```

Step 3

```
All Selected Sales =
SUMX(
    ALLSELECTED( Sales ),
    Sales[Quantity] * Sales[Unit Price]
)
```

Step 4

```
# of Customers (Values) =
COUNTROWS(
    VALUES( Customer[Customer ID] )
)

# of Customers (Distinct) =
COUNTROWS(
    DISTINCT( Customer[Customer ID] )
)
```

Statements and Variables

Statements are useful for cleaning up DAX code to make it easier to digest. Let's see how using the statements **VAR** and **RETURN** make it easier to read DAX. First, add a slicer to your report that uses the *Discount[Value]* column as is values.

1. Create a measure called *Discounted Sales 1* that calculates the *Sales Amount* given the selected discount.
2. Make a copy of the *Discounted Sales 1* measure in the *Sales* table and name it *Discounted Sales 2*.
3. Use the **VAR** statement to store the expression **1 – SELECTEDVALUE** as a variable called *DiscountPerc*. Then after the **RETURN** statement we can include the expression with the variable.

Each DAX expression can include as many variables as you wish with only one **RETURN** statement.

4. Create a copy of *Discounted Sales 2* and include a variable for 1, the discount percentage, and the calculation.

It is important to remember that variables exist to make your code easier to read and provide additional functionality as we will see further in the course.

You can even create variables within variables. Inception??

Check Your Code

```
Step 1

Discounted Sales 1 =
[Sales Amount Measure] * ( 1 - SELECTEDVALUE( Discount[Value] )

Step 2

Discounted Sales 2 =
VAR DiscountPerc =
    1 - SELECTEDVALUE( Discount[Value] )
RETURN
    [Sales Amount Measure] * DiscountPerc

Step 4

Discounted Sales 3 =
VAR One = 1
VAR Disc = SELECTEDVALUE( Discount[Value] )
VAR Result = One - Disc
RETURN
    [Sales Amount Measure] * Result
```

Evaluation Contexts: Filter Context

To see how we can modify the filter context with DAX we need to create a few measures.

1. Create a measure in the *Sales* table called *Total US Sales* that calculates the total sales in the US regardless of time or product model.
2. Create a measure called *Total Sales* that calculates the total sales regardless of time or product model.
3. Create a measure called *US Sales* that calculates the total sales regardless of time.

Notice that when the measure are all put as values into a matrix visualization with *Product[Product Model]* in the rows field, the numbers show up differently.

4. Create a slicer with *Date[Year]* and see which numbers change and which don't. That is because the filter context is different for each measure.
5. Create a measure that returns the total US sales regardless of product model but that changes when a different year is selected.

Think about the filter context, then compare your DAX with the solution in the Check Your Code section.

Filter context defines the subset of the data that is used to calculate measures. When troubleshooting, think about your filter context.

Check Your Code

Step 1

```
Total US Sales =
SUMX(
    FILTER(
        ALL('Sales'),
        RELATED( 'Address'[Country/Region] ) = "United States"
    ),
    [Sales Amount Measure]
)
```

Step 2

```
Total Sales = SUMX( ALL( Sales ), [Sales Amount Measure] )
```

Step 3

```
US Sales =
SUMX(
    FILTER(
    'Sales',
        RELATED( 'Address'[Country/Region] ) = "United States"
    ),
    [Sales Amount Measure]
)
```

Step 4

```
Total Yearly US Sales =
SUMX(
    ALL( 'Product'[Product Model] ),
    SUMX(
        FILTER(
            'Address',
            'Address'[Country/Region] = "United States"
        ),
        [Sales Amount Measure]
    )
)
```

Evaluation Contexts: Row Context

Row context is a subset of filter context because the filter context determines which rows are in the row context. A DAX formula is then evaluated for each row in the row context. Calculated columns have an implicit row context because they create a new column for every row in a table. Measures allow for some flexibility with the row context.

1. Create a measure in the *Sales* table called *Line Cost*. We want a measure that gives us the total cost for each order.

The **RELATED** function allows us to access the row context from another table and calculate the line cost.

Remember, **RELATED** is used to access a table on the one side of a one-to-many relationship. **RELATEDTABLE** is used to access a table on the many side of a one-to-many relationship.

2. Create a measure in the *Product* table that returns the line total for each product. Remember how we access a table on the many side of a one-to-many relationship.

Now that we know how to access the row context of other tables, let's learn how to access the outer context within a single table.

3. Create a calculated column in the *Product* table called *Rank* that returns the rank of each product based on price with the highest priced product being first. Hint: Use the **COUNTROWS** function.

Variables allow us to access the outer row context and compare the current row to itself within a function.

Calculated columns have implicit row context. Measures do not. Keep this in mind when writing measures.

Check Your Code

Step 1

```
Line Cost =
SUMX(
    Sales,
    Sales[Quantity] * RELATED( 'Product'[Cost] )
)
```

Step 2

```
Product Line Total =
SUMX(
    RELATEDTABLE( Sales ),
    Sales[Line Total]
)
```

Step 3

```
Rank =
VAR ProductPrice = 'Product'[Price]
RETURN
COUNTROWS(
     FILTER(
          VALUES( 'Product'[Price] ),
          'Product'[Price] > ProductPrice
     )
) + 1
```

CALCULATE Function

The **CALCULATE** function is the most powerful function in DAX and should be used often. We can replace many of the iterator functions with a **CALCULATE** function.

1. Create a new measure in the *Sales* table called *Total Yearly US Sales 2* to calculate the same values as the *Total Yearly US Sales* we created earlier with a SUMX function.

The benefits of using **CALCULATE** include a faster model and cleaner DAX.

Now let's see how we can use the **KEEPFILTERS** function, which is specific to the **CALCULATE** function. **KEEPFILTERS** retains the outer filter context rather than replacing it with the filter context defined in the **CALCULATE** function.

2. Create a new measure in the *Sales* table called *Bike Sales* that calculates the total sales in the bike group regardless of product model.

This gives us the total, but it shows up in every row.

3. Modify the *Bikes Sales* measure so the total only shows up in the Bikes row. Hint: Use the **KEEPFILTERS** function instead of **FILTER**.

It is best practice to include measures in the expression argument of the CALCULATE function.

Check Your Code

Step 1

```
Total Yearly US Sales 2 =
CALCULATE(
    [Sales Amount Measure],
    ALL( 'Product'[Product Model] ),
    'Address'[Country/Region] = "United States"
)
```

Step 2

```
Bike Sales =
CALCULATE(
    [Sales Amount Measure],
    FILTER(
        ALL( 'Product' ),
        'Product'[Product Category] = "Bikes"
    )
)
```

Step 3

```
Only Bike Sales =
CALCULATE(
    [Sales Amount Measure],
    KEEPFILTERS( 'Product'[Product Category] = "Bikes" )
)
```

Time Intelligence

DAX simplifies building measures for time-based calculations with time intelligence functions. These functions allow you to quickly calculate a month-to-date version of a measure without needing to specify the filter logic for month-to-date calculations. Other functions include SAMEPERIODLASTYEAR() which is extremely useful for calculating year-over-year metrics.

Your data model must meet a few requirements for the time intelligence functions to work properly. You can see these requirements in Microsoft's documentation.

1. You must have a date table that is marked as a date table
2. Your date table must contain full years of data – Jan 1 to Dec 31 for each year if using a standard year. If using a fiscal year starting in July then all dates between Jul 1 and Jun 30 must be included in the date table
3. The date column must have unique values formatted as a date, not datetime

The model we have includes a date table that meets all of these requirements for a standard year. Let's explore a few of the most common time intelligence functions and how to implement them in your data model.

1. Create a measure that calculates the month-to-date value of the Sales Amount Measure using the CALCULATE function
2. Create a measure that calculates the quarter-to-date value of the Sales Amount Measure not using the CALCULATE function
3. Create a measure that calculates the year-to-date value of the Sales Amount Measure not using the CALCULATE function

You will notice that there is more than one way to write each of these measures. Time intelligence functions allow you the flexibility to customize the measure while also leveraging their simplicity.

Now let's write a more complex measure. Often time intelligence functions may not meet the exact needs of a specific metric, but can help get there.

4. Create a measure that takes the sum of the last three complete months from the current month. For example, if the month is April, sum months January thru March. Remember, it needs to be dynamic so as the month we are looking at shifts, the 3-month sum shifts as well.
   Hint: use the DATESBETWEEN function.

It is best practice to include measures in the expression argument of the CALCULATE function.

Check Your Code

Step 1

```
MTD Sales Amount =
CALCULATE ( [Sales Amount Measure], DATESMTD ( 'Date'[Date] ) )
```

Step 2

```
QTD Sales Amount =
TOTALQTD ( [Sales Amount Measure], 'Date'[Date] )
```

Step 3

```
YTD Sales Amount =
TOTALYTD ( [Sales Amount Measure], 'Date'[Date] )
```

Step 4

```
3-Month Sum =
var lastmonth = EOMONTH( MAX( 'Date'[Date] ), -1 )
var threemonthsago = EOMONTH( MAX( 'Date'[Date] ), -4 ) + 1
var result =
    CALCULATE(
        [Sales Amount Measure],
        DATESBETWEEN( 'Date'[Date], threemonthsago, lastmonth )
    )
RETURN
    result
```


JOURNEYTEAM