Fundamentals of DAX

Course Workbook

JOURNEYTEAM

Guide to Workbook

This workbook contains exercises for the JourneyTEAM Fundamentals of DAX course.

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

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

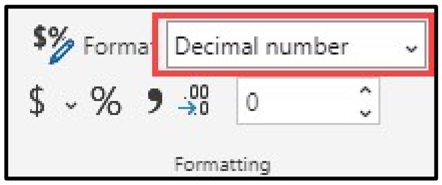
Each section in this workbook contains step-by-step instruction followed by a solutions subsection called *Check Your Code*. To get the most out of this workbook, wait to look at the *Check Your Code* section until you have tried to write the DAX on your own.

Enjoy!

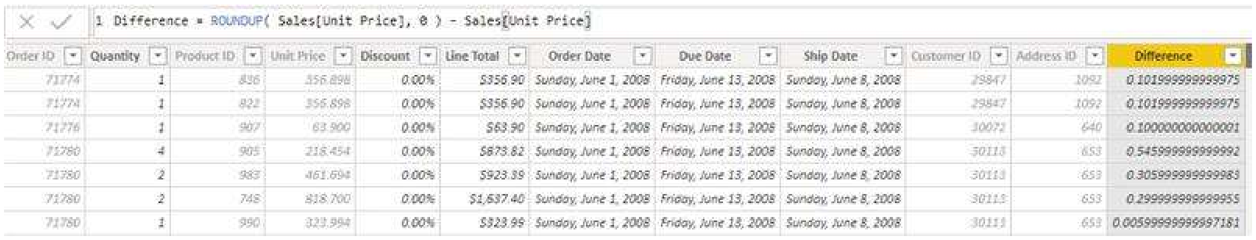
© Copyright 2019

Data Types

Go to the data view on the left-hand side of Power BI Desktop and open the *Sales* table. Click on the *Unit Price* column and then the *Modeling* tab in the ribbon. The *Unit Price* column is hidden in the report view so it will be slightly faded in the data view. Here you will see the *Data Type* set as a decimal number.



1. Create a calculated column called *Differences* that subtracts the Unit Price from the rounded-up Unit Price (check your code on the next page). Notice the results contain decimals.

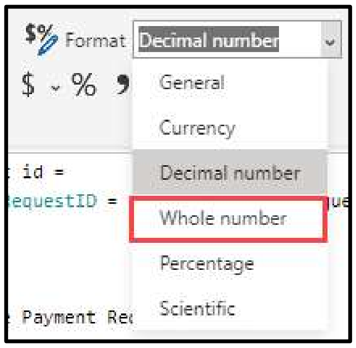


```
1 Difference = ROUNDUP( Sales[Unit Price], 0 ) - Sales[Unit Price]
```

Order ID	Quantity	Product ID	Unit Price	Discount	Line Total	Order Date	Due Date	Ship Date	Customer ID	Address ID	Difference
71774	1	836	356.898	0.00%	$356.90	Sunday, June 1, 2008	Friday, June 13, 2008	Sunday, June 8, 2008	29847	1092	0.101999999999975
71774	1	822	356.898	0.00%	$356.90	Sunday, June 1, 2008	Friday, June 13, 2008	Sunday, June 8, 2008	29847	1092	0.101999999999975
71776	1	907	63.900	0.00%	$63.90	Sunday, June 1, 2008	Friday, June 13, 2008	Sunday, June 8, 2008	30072	640	0.100000000000001
71780	4	905	218.454	0.00%	$873.82	Sunday, June 1, 2008	Friday, June 13, 2008	Sunday, June 8, 2008	30113	653	0.545999999999992
71780	2	983	461.694	0.00%	$923.39	Sunday, June 1, 2008	Friday, June 13, 2008	Sunday, June 8, 2008	30113	653	0.305999999999983
71780	2	748	818.700	0.00%	$1,637.40	Sunday, June 1, 2008	Friday, June 13, 2008	Sunday, June 8, 2008	30113	653	0.299999999999955
71780	1	990	323.994	0.00%	$323.99	Sunday, June 1, 2008	Friday, June 13, 2008	Sunday, June 8, 2008	30113	653	0.00599999999997181

Let's see what happens when we **change the data type** to whole number.

2. Change *Unit Price* to a whole number by clicking on the drop down next to *Data type* and selecting whole number.

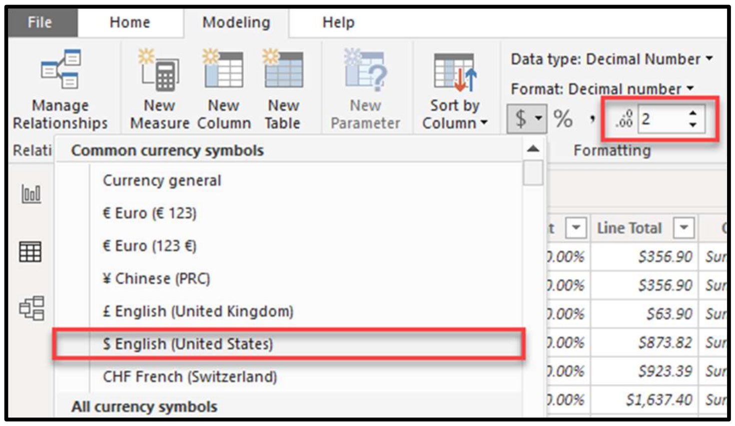


See how the *Difference* calculated column has changed to 0. The reason is because the underlying data has been modified in Power Query and as such the result of the calculation is different.

3. Now change the data type back to decimal number and refresh the data (the data will not change until you refresh). Calculated columns are only updated at data refresh; this is also true of DAX generated tables, but we'll get to those later.

Once the data is refreshed you will see the *Difference* calculated column now contains decimal numbers again.

What if you want the *Unit Price* to only show up to 2 decimals? 0 decimals? 8 decimals? No problem. Formatting the data doesn't change the underlying data type and value in Power Query, only the way the value is displayed. Remember, formatting is strictly cosmetic and doesn't change your data model. In the *Sales* table select *Unit Price* and then go to the



Data type: Decimal Number
Format: Decimal number
2

Modeling tab.

4. Let's format to 2 decimals and make it currency as well. Notice the value of *Difference* doesn't change when you format the *Unit Price* column.

Some actions in Power BI Desktop will change your M code in Power Query. Changing the format of the column is often the better choice because the underlying data types and values remain the same.

Check Your Code

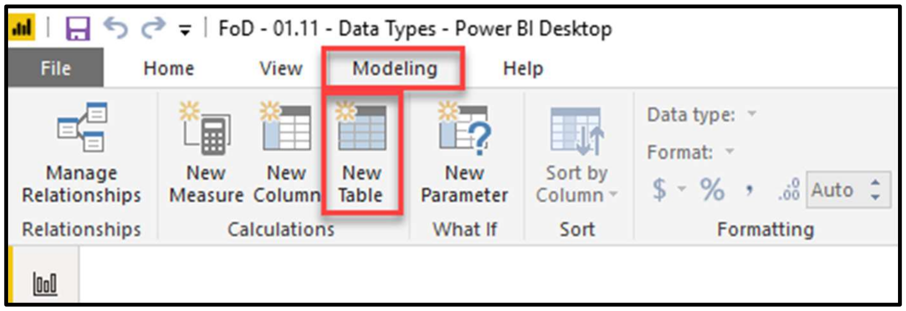
```
Step 1

Differences = ROUNDUP( Sales[Unit Price], 0 ) - Sales[Unit Price]
```

Calculated Tables

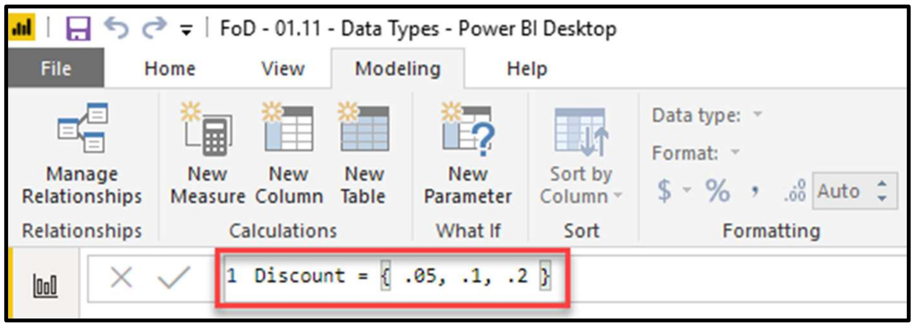
Tables are the core of any Power BI file and they can come from a database, excel file, or many other sources. However, sometimes you need a data table that contains customizable data like a date table, which is covered later, or a goal table with static goals. In these cases we will create a table completely with DAX.

1. Create a table that contains three discounts, 5%, 10%, and 20%. Start by going to the *Modeling* tab and clicking *New Table*.



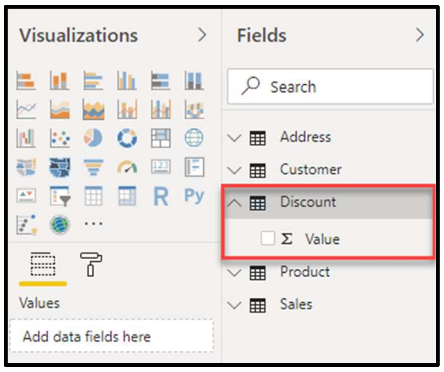
This will open the DAX code dialog box.

2. Type **`Discount = { 0.05, 0.1, 0.2 }`** and click enter (more complicated DAX is in the Check Your Code section).



```
Discount = { .05, .1, .2 }
```

A new table will appear in the *Fields* menu on the right of your screen with a column single column called, "*Value*". In this column are the values you just typed in the formula bar above.



Check Your Code

This code is the more complete method for creating a DAX table. The `DATATABLE` function allows you to name columns and specify data types and thus is more visually appealing.

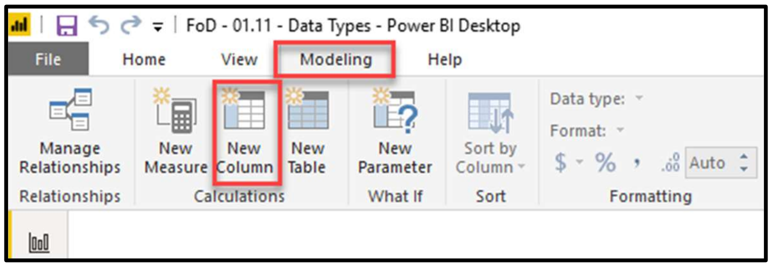
```
Step 2

Discount (Advanced) =
DATATABLE (
    "Row", INTEGER,
    "Discount", CURRENCY,
    {
        { 1, 0.05 },
        { 2, 0.1 },
        { 3, 0.2 }
    }
)
```

Calculated Columns

Best practice dictates calculated columns should be used primarily to categorize your data, but they can also be used to do calculations. We are going to, strictly for educational purposes, use calculated columns for calculations in this section despite it going against best practice. We are going to create a new column labeled *Sales Amount* that calculates the gross dollar amount for each sale.

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



2. Then type **`Sales Amount =`**

Let's think about how we would get the gross dollar amount for each sale. We should take the quantity and multiply it by the unit price so your code should look like this: **`Sales Amount = Sales[Quantity] * Sales[Unit Price]`**.

3. Go ahead and finish your new calculated column and hit enter to save it.

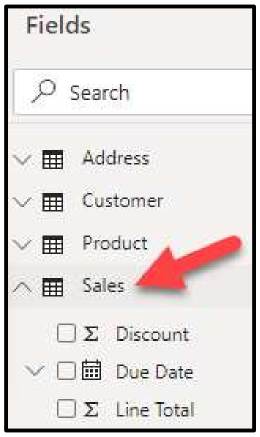
You now can see the gross dollar amount of each sale. Next, we will show that while calculations can be done in calculated columns, measures are best practice when working with numbers.

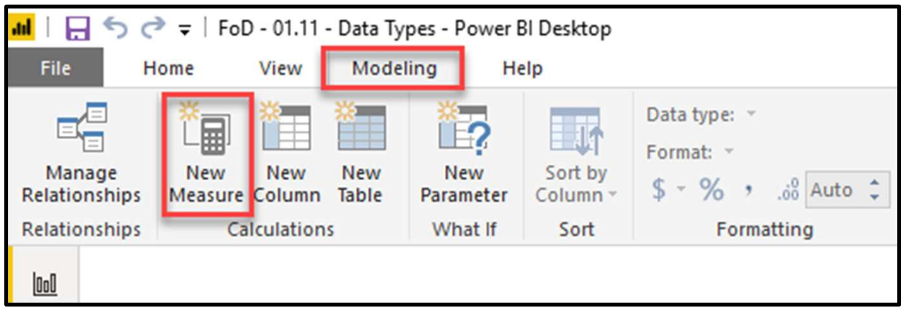
Calculated columns are best used for categorizing data such as grouping your customers into age groups.

Measures

In the previous section we used a calculated column to find the gross dollar amount per sale (which we called, *Sales Amount*). While that works, it's not best practice. Measures are the best way to calculate numbers with your data because they are calculated *as needed* rather that at data refresh time, therefore taking up space in your model. Let's see how we get the same result as a calculated column, but while using a measure instead.

1. 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*.

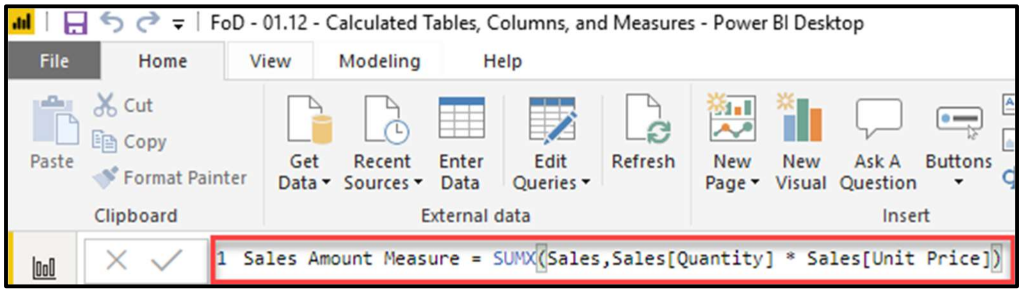




2. Then type **`Sales Amount Measure =`** and let's type the same DAX code we did in step 3 in the Calculated Columns section, quantity multiplied by the unit price.

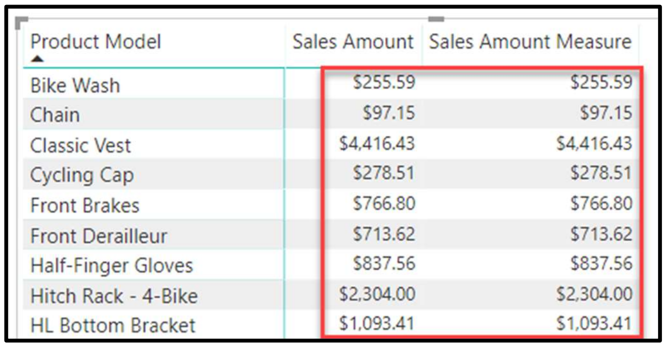
Note an error is returned. That is because when referring to every row in a table like we are in this example Measures need aggregator functions such as **`SUMX`** or **`AVERAGEX`**. We will cover functions like these a little later.

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



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

4. To verify that the calculated column and measure have the same result, go to the *Report* tab on the left and create a matrix visualization.
5. Drag *Product[Product Model]* to the rows field and *Sales[Sales Amount]* and *Sales Amount Measure* to the values field. The result is the same.



Product Model	Sales Amount	Sales Amount Measure
Bike Wash	$255.59	$255.59
Chain	$97.15	$97.15
Classic Vest	$4,416.43	$4,416.43
Cycling Cap	$278.51	$278.51
Front Brakes	$766.80	$766.80
Front Derailleur	$713.62	$713.62
Half-Finger Gloves	$837.56	$837.56
Hitch Rack - 4-Bike	$2,304.00	$2,304.00
HL Bottom Bracket	$1,093.41	$1,093.41

*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 to 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: Mathematical

Mathematical functions work much like Excel functions do. Note that while these functions should almost always be used in measures, these exercises will be done in calculated columns so we can see the results in the data view.

We are going to create several columns in the *Sales* table that modify the *Sales[Line Total]* values using different mathematical functions. Take a moment to try the functions on your own and then compare your code with the code below.

1. Create three calculated columns named *Int Example*, *Floor Example*, and *Mod example* using `Sales[Line Total]` as the input. *Floor Example* should be rounded to the hundreds place. Use 100 as the divisor in *Mod Example*.

   `INT` returns the integer of the input.
   `FLOOR` rounds the input down to the nearest multiple of significance.
   `MOD` returns the remainder after an input is divided by a divisor.

Mathematical functions are well documented by Microsoft and at dax.guide.

Check Your Code

```
Step 1

Int Example = INT( Sales[Line Total] )

Floor Example = FLOOR( Sales[Line Total], 100 )

Mod Example = MOD( Sales[Line Total], 100 )
```

Functions: Text

While you may not use these functions much, DAX is able to perform actions on text values using text functions.

1. Create a calculated column in the *Product* table called *Description Length* to calculate the length of the *Product[Description]*.

It should be noted that this will not work as a measure because it lacks an aggregator. Measures can return text values and are appropriate to use as such given the circumstance. One example of this would be using the `SELECTEDVALUE` function to return different text based on the selection in a slicer.

Then look at the *Customer* table and notice that we have no field for the customer's full name.

2. Use the `CONCATENATE` function to combine the first and last names into a calculated column called *Full Name*.

Notice that there is no space between the first and last names. `CONCATENATE` can only accept two inputs, so another method is needed.

3. Create another column called *Full Name 2* and use operators instead. The "&" is used to concatenate multiple text fields together into one. And while the ampersand is used to concatenate, it also can be used as text within double quotes.

Even text functions have iterators. Use CONCATENATEX to string the results of an expression together. The function even allows you to choose the separator between each value.

Check Your Code

```
Step 1

Description Length = LEN( 'Product'[Description] )

Step 2

Full Name = CONCATENATE( [First Name], Customer[Last Name] )

Step 3

Full Name 2 = Customer[First Name] & " " & Customer[Last Name]
```

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: Relational

When creating calculated tables, columns, and measures, referencing columns within the same table where the calculation lives is easy. Referencing columns in other tables requires additional DAX.

The `RELATED` function is used for referencing columns on the one side of a one-to-many relationship. `RELATEDTABLE` is used for referencing the table on the many side of a one-to-many relationship. Let’s see them in practice.

1. Create a new calculated column in the *Sales* table called *Country*. See how the country from the *Address* table is populated for each row.
2. Now create a calculated column in the *Product* table called *Total Sales Quantity* that computes the total number of units ordered for each product. See if you can do it on your own before looking at the solution (on the next page).

Measures typically do not need relational functions, but calculated columns often do. This is because measures don't really live in a table. They're represented as being assigned to a specific table, but they don't necessarily live in that table.

Check Your Code

```
Step 1

Country = RELATED( 'Address'[Country/Region] )

Step 2

Total Sales Quantity =
SUMX(
    RELATEDTABLE( Sales ),
    Sales[Quantity]
)
```

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
```

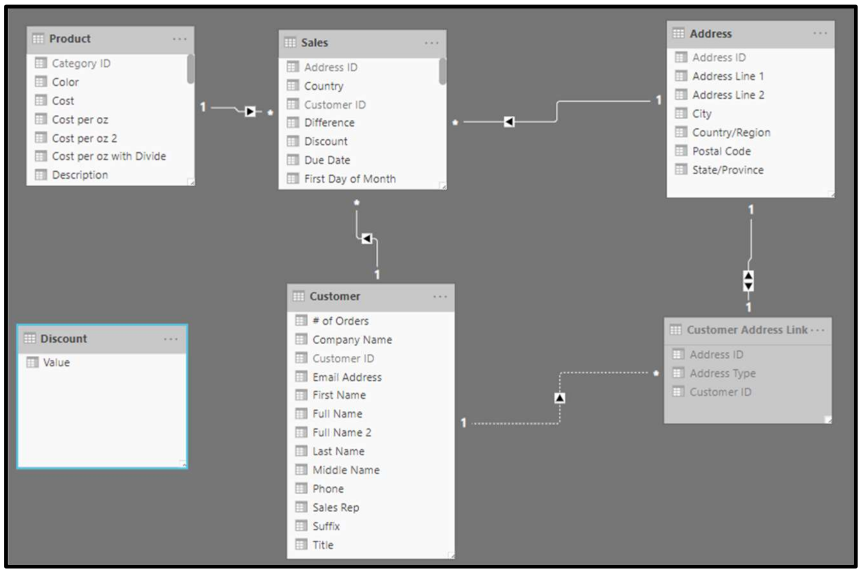
Modeling

While this is a course about DAX we need to take a moment to talk about modeling. Modeling is how you design and connect your data tables, and a model consists of the data tables and the relationships between those tables.

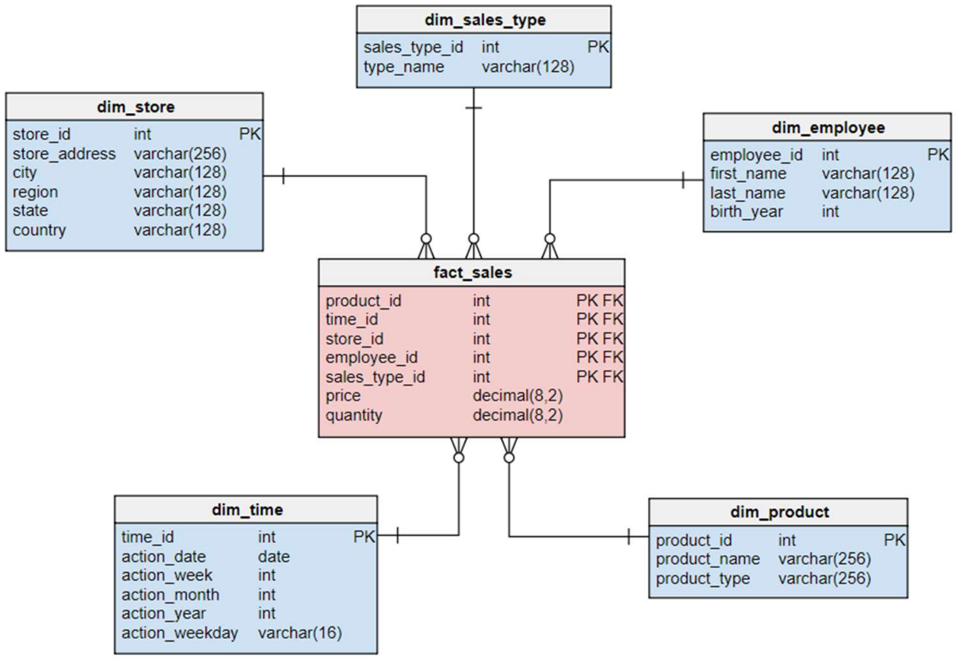
Data tables come in two forms, fact and dimension tables. Fact tables contain the core data of your model. Tables like sales transactions, ticket tracking, time entries are good examples of fact tables. Dimension tables contain data that describes the fact table. Customer address, product category, salesperson, etc.

Typically. a data model should have one or two fact tables with two to eight, maybe more but not likely, dimension tables arranged in what is called a star schema.

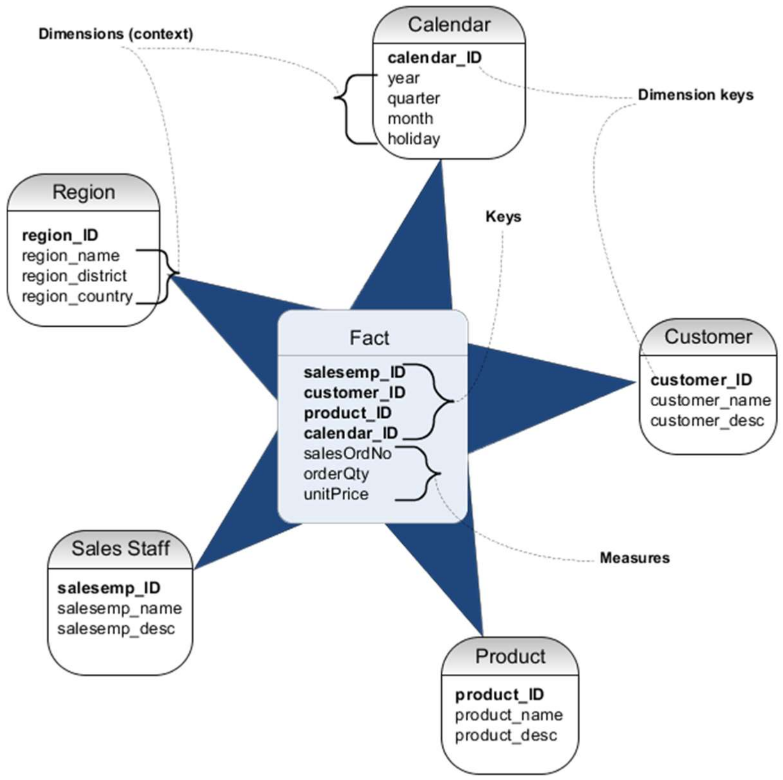
This is what the data model should look like for this data set. Strive to always have your data model resemble a star schema!



Here are another couple examples of what a star schema might look like:



This is a good example of why we call it a "Star" schema. It… kinda looks like a star 😊



Creating a Date Table

Creating a date table is one of the first things you should do when building a report. A date table will speed up your model, unify your time filters, and provide the ability to customize date hierarchies and formatting.

DAX contains two functions for creating a date table, `CALENDARAUTO` and `CALENDAR`. Let's look at both.

`CALENDARAUTO` automatically scans all the date columns in your data model and finds the minimum and maximum date and creates a one column table with dates starting on January 1st of the minimum year and ending on December 31st of the maximum year.

1. Create a new table called Date using the `CALENDARAUTO` function and see the results (You can specify the fiscal end month by entering in the month number in the function).

`CALENDARAUTO` may contain more dates than you need, perhaps because you have a customer that was born in 1920! We can create a more customized date table by using the `CALENDAR` function. This function allows you to specify a start and end date for your date table.

2. Create a date table that starts on January 1st, 2016 to December 31st, 2019.

The dates can also be dynamic rather than being hard coded. A better option would be to take the year of the minimum date in the *Sales* table for the start date and the year of the maximum date of the *Sales* table for the end date.

3. Replace the *year* value with a `MIN` and `MAX` to make this function dynamic.

These date tables are great, but a single column of dates is not a good date table. The `ADDCOLUMNS` function in DAX is ideal for adding metadata to the date table such as month, quarter, day or week, and more.

4. In the DAX code you just created in step 3, add a `VAR/RETURN` pair with the variable *Dates* being the calendar code you just created. Then use the `ADDCOLUMNS` function to add the *Year*, *Quarter*, *Month*, *Month Number*, *Day of Week*, and *Day of Week*

Number to your date table. Hint: You will need to use the **`FORMAT`** function to get the month names and day of week names.

Fiscal calendars are supported by the CALENDAR and CALENDARAUTO functions. Include the last month of the fiscal year in the optional argument.

Check Your Code

Step 1

```
Date = CALENDARAUTO()
```

Step 2

```
Date =
CALENDAR(
    DATE( 2016, 1, 1 ),
    DATE( 2019, 12, 31 )
)
```

Step 3

```
Date =
CALENDAR(
    DATE( YEAR( MIN( Sales[Order Date] ) ), 1, 1 ),
    DATE( YEAR( MAX( Sales[Due Date] ) ), 12, 31 )
)
```

Step 4

```
Date =
VAR Dates =
CALENDAR (
     DATE ( YEAR ( MIN ( Sales[Order Date] ) ), 1, 1 ),
     DATE ( YEAR ( MAX ( Sales[Due Date] ) ), 12, 31 )
)
RETURN
ADDCOLUMNS(
     Dates,
     "Year", YEAR ( [Date] ),
     "Quarter", QUARTER( [Date] ),
     "Month", FORMAT ( [Date], "mmm" ),
     "Month Number", MONTH ( [Date] ),
     "Day of Week", FORMAT ( [Date], "dddd" ),
     "Day of Week Number", WEEKDAY ( [Date] )
)
```

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 contact 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" )
)
```

Other Classes Taught by JourneyTEAM

Power BI

- Fundamentals of DAX
- Intermediate DAX
- Advanced DAX
- Dashboard in a Day
- Advanced Dashboard in a Day
- Admin in a Day
- Advanced Visualizations

Power Automate

- Flow in a Day
- Advanced Flow in a Day

Power Apps

- Power Apps in a Day
- Advanced Power Apps in a Day

Power Virtual Agent

- Virtual Agent in a Day

JOURNEYTEAM