Data Analysis Made Easy with Microsoft Power Tools
05 DAME Video: M Code in Power Query

This video teaches the fundamentals of the M Code functional language in Power Query.

This video is not as in-depth as MECS video #10, listed here:

https://www.youtube.com/watch?v=3ZkIwKBVkVE&list=PLrRPvpgDmw0nre_bTeBfJWjrnixKoyNtW&index=16

This video will not be as in-depth as my new book that will be out Oct, 2024:

https://www.amazon.com/Transformative-Magic-Power-Query-Excel/dp/1615470832

checking out because it has every minute and tiny detail about M Code.

> https://learn.microsoft.com/en-us/powerquery-m/power-query-m-language-specification

Table of Contents

History 3
Power Query and M Code 3
M Code as Compared To Other Three Functional Languages 3
User Interface & M Code 4
Three places to edit M Code 6
Expression 6
M Code Values 7
Type values and data types 8
Operators, values, and data types 9
Identifiers, Keywords 11
Let expression 12
Null Value 13
Logical values 13
Text Value 13
Number values 14
Date, Time, Datetime, Datetimezone and Duration 15
  Duration value 15
  Date value 16
  Time value 18
  Datetime value 19
  Datetimezone value 20
Tables, records, and lists 21
List value 21
Record value 22

Table value 22
Binary value 23
M Code Lookup Formulas 24
Rules for row index lookup 26
Rules for key match lookup: 26
Drill Down, lookup formulas and primary keys 28
Approximate match lookup 42
Table.Buffer function 42
Join operations used by Merge feature 29
Custom function value 31
Each and underscore to create custom functions 32
If expression 34
Table.AddColumn function 34
Excel.CurrentWorkbook function 35
Csv.Document function 36
File.Contents function 39
On-premises file and folder paths 40
Data source settings 41
Table.Group function 44
  Table.Group fifth argument: Comparer 45
  Comparer functions and where to use them 45

History

Power Query was first introduced in Excel in 2013, exactly 20 years after the PivotTable was invented. For 20 years the PivotTable was the preferred method of taking a table of data and converting into a useful summary reports and chart visuals. The PivotTable was easy to use if you had a correctly structured table with column names and records. But often the data came from many different sources and did not have the correct table structure to create the reports and chart visuals. To get the correct structure for the PivotTable tool, we would use VBA code or manually convert the data into correct structure. When Power Query arrived, it dramatically improved our ability to take data from different data sources and then clean, transform and load the data to the best location in the Excel workbook file. Then in 2015, Power BI Desktop was invented to easily report and visualize data. Microsoft embedded Power Query into Power BI Desktop to make connecting to and structuring data easy. Finally, in about 2023, Microsoft finally gave us the full Power Query tool in the Power BI Online Service and they called it Dataflow. Now, with whatever tool you may be using, Power Query allows you to connect to almost any data source and transform the data into just the structure you need to create your data analysis and analytic solutions.

Power Query and M Code

Power Query is a Microsoft tool that allows you to create queries that can connect to, import, clean, transform and load data. The amazing Power Query tool is in the apps: Excel, Power BI Desktop, and the Power BI Online Service (called Dataflow). All Power Query queries are created with the programming language called M Code. Microsoft gave it the name M Code because they described what the tool could do as "Mashup Data". The M in Mashup is the M in M Code. **M Code** is officially defined as a case-sensitive, function-based language that can deliver M Code values, such as tables, lists, numbers and 12 other data related values. When you create a Power Query query, you can use the user interface to write the M code, you can type your own M Code, or use a combination of both. The function-based M Code language is a complement to, and works closely with, the other three Microsoft function-based languages in Excel and Power BI: worksheet formulas, Standard PivotTables and Data Model DAX formulas. All four function-based languages help you perform calculations, analytics and data analysis. To help place Power Query M Code into the correct context, I will describe the main characteristics of each functional language.

M Code as Compared To Other Three Functional Languages

The hallmarks of worksheet formulas are that you can point your formula to any cell, range, column or table in the Excel worksheet. The other three languages cannot do that: they are all confined to structured data such as tables and columns. With worksheet formulas you have freedom to incorporate any part of the worksheet, such as tables, columns, ranges or cells. You are not limited to a strict column and table data structure like with Standard PivotTables, DAX formulas or M Code. There are also many convenient worksheet functions such as XLOOKUP, NETWORKDAYS.INTL and BINOM.DIST.RANGE that are not available in the other functional languages. Finally, worksheet formulas allow a solution to instantly update when the source data changes. The other three languages do not allow instant update when the source data changes. The hallmark of standard PivotTable calculations is that for summing, counting and calculating percentages, there is no other calculation tool that is as fast and easy to use as the standard PivotTable. When it comes to the DAX functional language, the hallmarks are that you can work

with big data more quickly than any of the other languages and you can create tables at various grains inside DAX formulas to help reduce the complexity of the calculation process. The DAX language has powerful functions that the other languages do not have such as CALCULATE, AVERAGEX and RELATEDTABLE. As for M Code, the hallmark is that it allows you to work with data, tables, data sources such as SQL databases, columns of tables, columns of files and other structures that contain data in ways that none of the other languages can. M Code is specifically programmed to work with data and allows you to shape and form the data into efficient structures that can be loaded to the worksheet, the PivotTable cache, or the Data Model in either Power Pivot or Power BI. In addition, M Code is the only language that allows you to write most of your formulas using the user interface. Each functional language has its own power and magic to help you make calculations and perform data analysis, but for this book it is Power Query M Code's unique ability to work with and shape data that we will study.

User Interface & M Code

Even when you become a master of M Code, you will often use the user interface to write your M Code. Sometimes the user interface generated M Code will be perfect, other times you will edit the generated M Code to better meet your needs. I usually use the user interface to create most of my initial M Code, and then edit and hack the code to get just what I want. Throughout this book I will show you good hacks to get the user interface to write as much of the M Code as possible. To get started, the following three pictures (Figures 01 to 03) on the next page review the user interface in Excel, Power BI Desktop and Dataflow.

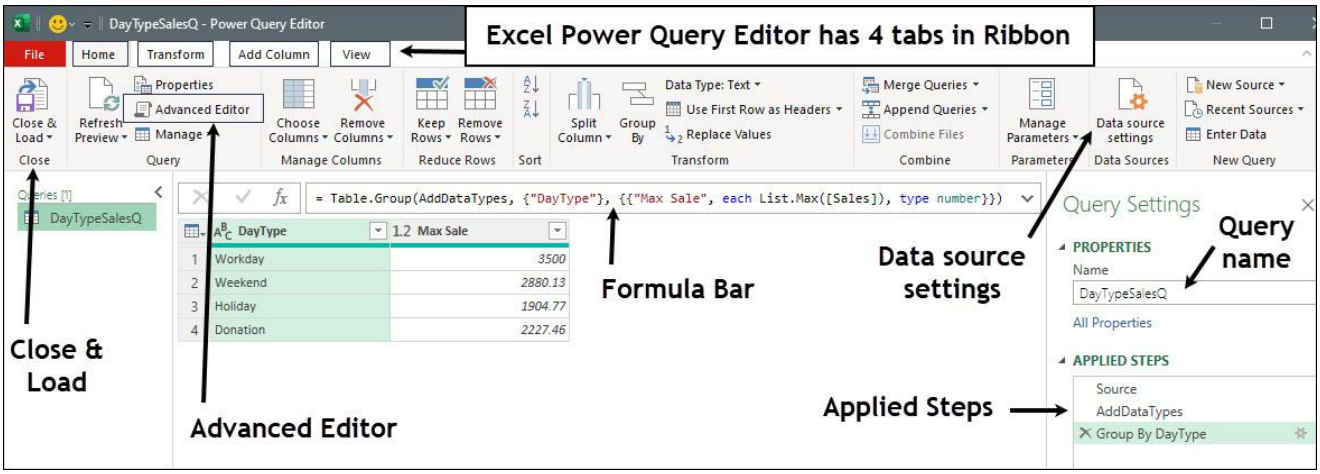


Figure 01: Power Query Editor in Excel.

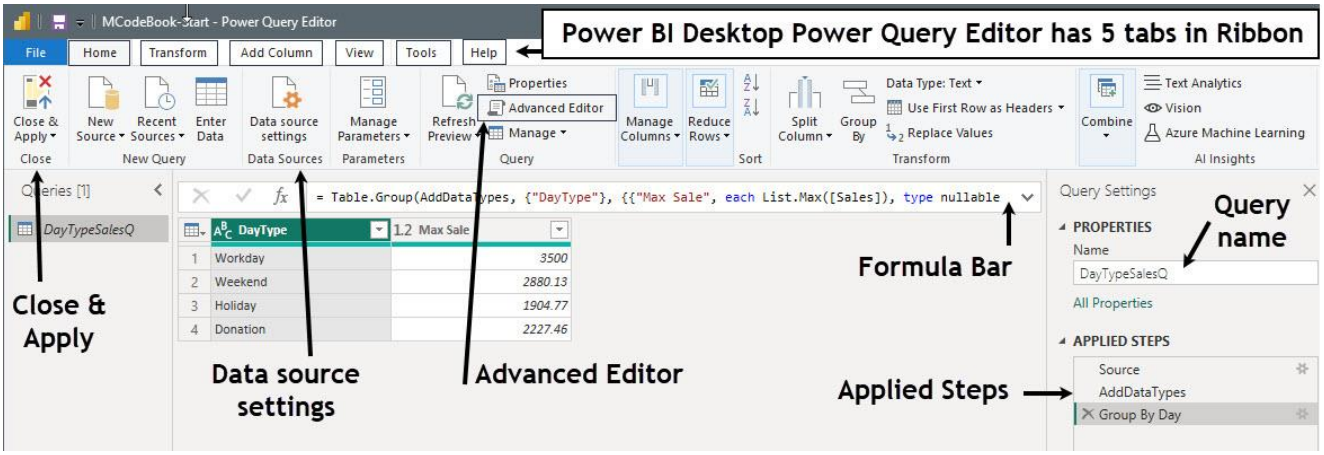


Figure 02: Power Query Editor in Power BI Desktop.

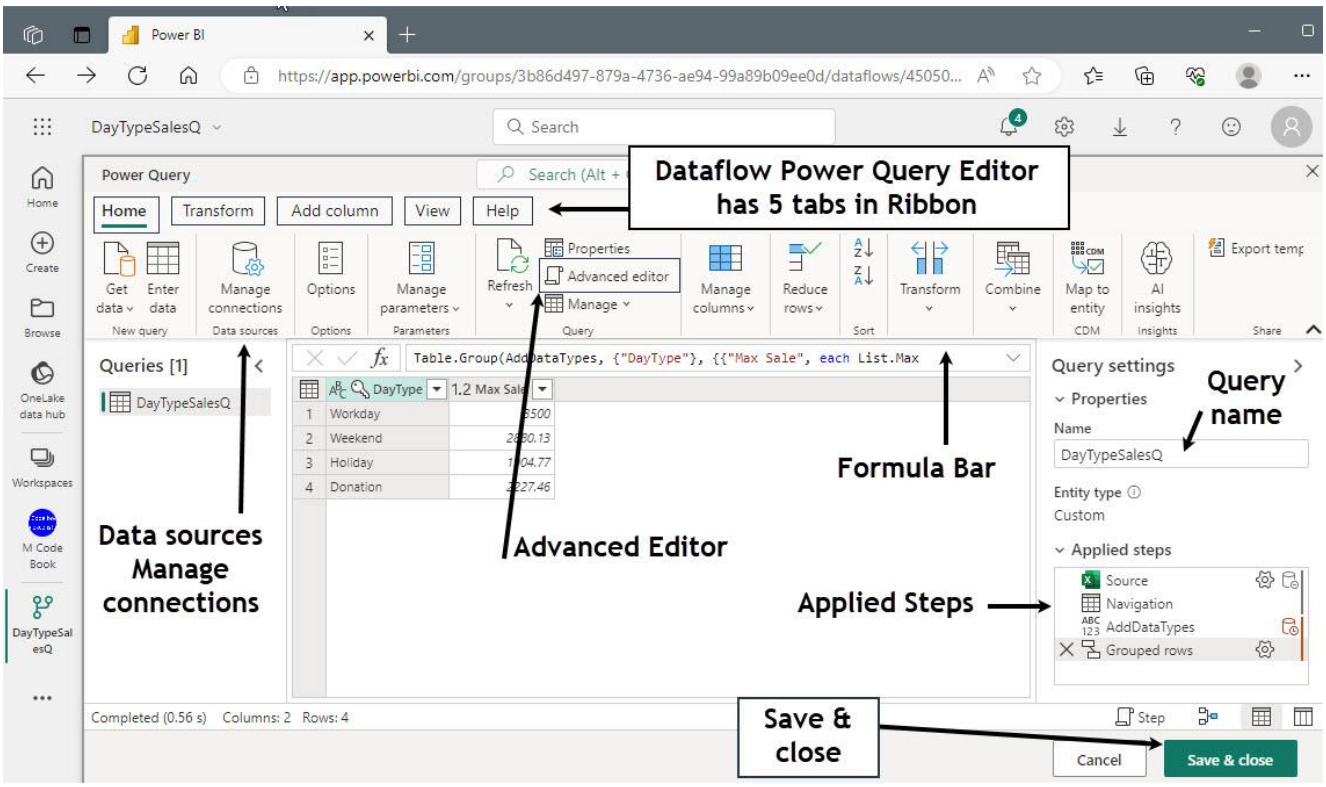


Figure 03: The Dataflow Power Query Editor is similar to the other two editors.

I will work almost exclusively in Excel's Power Query and take screen shots from within that app. When there are differences in the user interfaces, I will indicate the differences.

Now that we have reviewed the user interface, let's get started by creating a query using the user interface and then dive into M Code from there.

Three places to edit M Code

The three locations that you can edit M Code are listed below (in Power BI Desktop and Dataflows the Advanced Editor is in a different location as shown on previous page).

- Applied Steps
- Formula Bar
- Advanced Editor

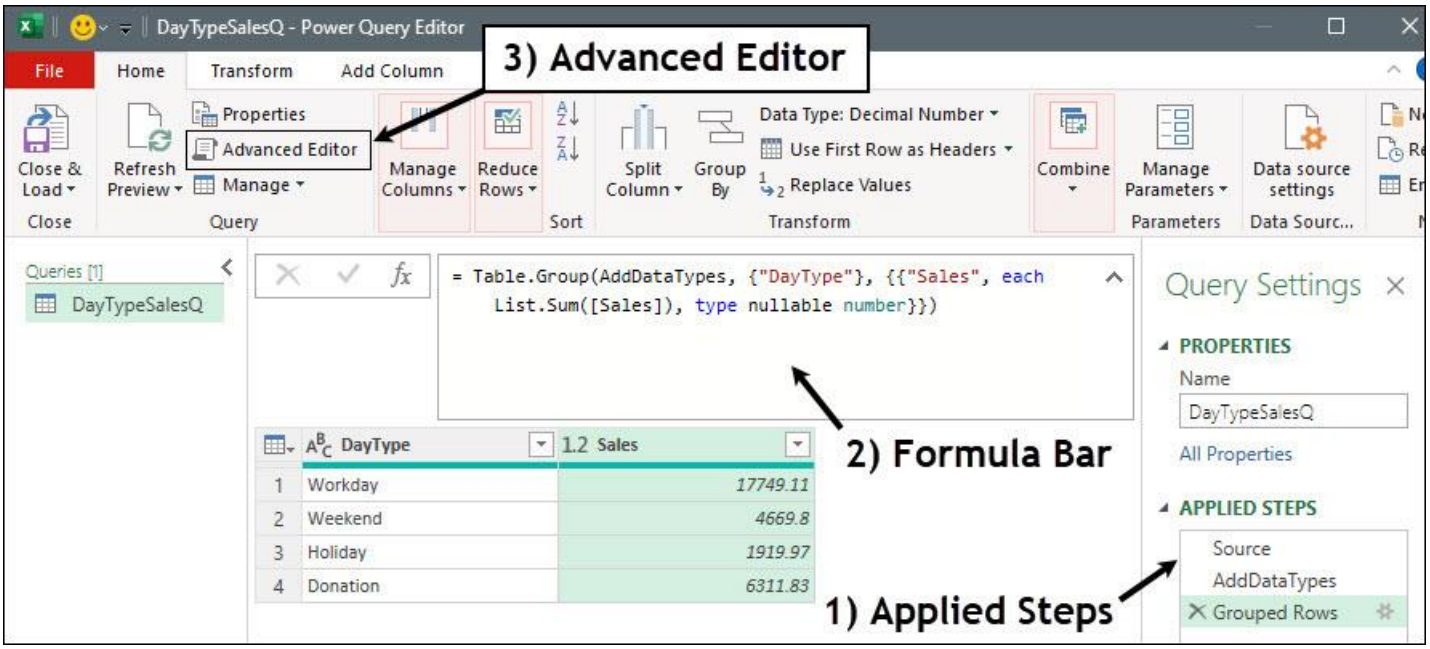


Figure-04: The three locations to edit M Code.

Expression

An expression is any M Code that results in a value.

Some examples of expressions:

- A function like Table.ColumnNames(TableName).
- A list.
- A number.
- A query step within a let expression.
- A full let expression.

M Code Values

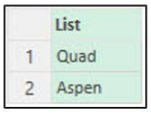
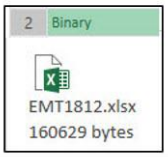
	Value	Description	M Code to Hard Code Value
1	Null	Absence of data.	Example: null
2	Logical	Boolean: true or false.	Example: true or false
3	Text	A string of Unicode characters.	Examples: "Quad", "A430G"
4	Number	Used for numeric and arithmetic operations.	Examples: -43 , 0 , 43.46 , 9.9e-5
5	Time	Time for 24 hour day as decimal between 0 and 1.	#time(hour, minute, seconds) Example: #time(11,10,57) = 11:10:57
6	Date	Date as a serial number (Common Era Gregorian calendar). Dates span from 01/01/0001 to 12/31/9999.	#date(year, month, day) Example: #date(1598, 01, 15)= 1/15/1598
7	DateTime	Datetime value contains both date and time.	#datetime(year, month, day, hour, minutes, seconds) Example: #datetime(2021,10,01,11,10,57)) = 10/01/21 11:10:57 AM
8	DateTime Zone	Represents a UTC (Universal Coordinated Time) date/time with a time-zone offset (last two arguments: hours & minutes).	#datetimezone(year, month, day, hour, minute, second, offset-hours, offset-minutes) Example: #datetimezone(2021,10,01,11,10,57,09,00) = 10/01/21 11:10:57 AM + 9 hours
9	Duration	An amount of time in units of days, hours, minutes, and seconds. Not supported in Dataflow.	#duration(days, hours, minutes, seconds) Example: #duration(01,01,10,0) = 1 day and 1 hour 10 minutes
10	Table	Table with field names and records.	Example: #table({"Boom Product", "Sales"}, {{"Quad",43},{"Aspen",35}}) =
11	Record	An ordered sequence of fields.	Example: [Boom Product = "Quad", Sales = 43] =
12	List	A sequence of M Code values.	Example: {“Quad”,”Aspen”} = 
13	Binary	Represents a sequence of bytes.	Example: Excel file = 
14	Function	Custom function that defines the variables and the mapping for those variables to deliver a value.	Example: Function calculates the effective rate = (APR, Periods) => Number.Power(1+APR/Periods, Periods)-1
15	Type	Classifies values with 1 of 16 data types.	Example: "type number" for defining the Decimal data type

Figure-05: M Code values.

An **M Code value** is what is produced by evaluating an expression. Figure-05 shows a list of all 15 values, a short description for each, and the M Code to hard code the value.

Type values and data types

	Data Type	Icon	Short Description	M Code
1	Decimal number	1.2	Number up to 15 decimals	type number
2	Currency (Fixed decimal number)	$	Number up to 4 decimals	Currency.Type
3	Whole number	123	Number with no digit to right of decimal. It removes decimals using bankers rounding.	Int64.Type
4	Percentage	%	Number up to 15 decimals with % Number Format	Percentage.Type
5	Date/Time		Serial number date and time together	type datetime
6	Date		Serial number date	type date
7	Time		Serial number time	type time
8	Date/Time/Timezone		Represents a UTC date/time with a time-zone offset	type datetimezone
9	Duration		Serial Number Length of Date and Time (not supported in Dataflow)	type duration
10	Text	ABC	Text	type text
11	True/False		Boolean	type logical
12	Binary		File like Excel file or Text file	type binary
13	Any	ABC 123	Sets numbers such as dates and decimals according to regional settings	type any
14	nullable		You can add the keyword nullable to data types like number so that the column can have the number or a null.	type nullable number
15	anynonnull		Any non-null value (all values excluding null).	type anynonnull
16	none		No values are classified	type none

Figure-06: Data types that classify M Code values.

A **type value** is a value that classifies other values with one of the 16 data types as shown in Figure-06. **Data types** are safeguards placed on columns to produce consistent data and accuracy in calculations and can change the underlying value to a new value if necessary. A value that is classified by a data type is said to conform to that data type. For example, if you apply the Whole number data type to a source column of number values with digits to the right of the decimal, new number values are created that are rounded to the ones position so that the new numbers conform to the definition of Whole number data type. In addition, if you load the whole number data to the worksheet or Data Model, all your calculations are based on the new whole numbers, not the original source number values. Further, if you change your mind about applying a data type, you are allowed to revert to the original source values before the data type was applied. This is possible because access to the source data values is stored in a query step that occurs before the query step that applies the data type. When you revert to the original source data, or you change the data type from one type to another, the query, the loaded data, and any analysis based on the loaded data, all are updated to match the new data type. A common example of using data types to change the source data to benefit analysis is when you get ISO text date values (YYYYMMDD). These values will not work as true dates in the worksheet or Data Model. But if you apply a date data type to the date text values, the text date values will be converted to date values. The date values would then work as true dates for your analysis. Next, we want to introduce how M Code values and operators interact.

Operators, values, and data types

Operators are applied to **operands** to form expressions. For example, in the expression 43 + 7 the number values 43 and 7 are operands and the operator is the addition operator (+). Figure-07 lists the M Code operators with a description and an example for each. In the figure, the list of operators presents the operator categories in order of **expression evaluation precedence (order of operations)**, from highest to lowest. Operators in the same category have equal precedence. Many of the operators in the list are the same operators that we use in the worksheet and DAX. However, the application of operators works much differently in M Code than in the worksheet and DAX. For example, in the worksheet and DAX if you want to add the number value 10 days, to the date value 11/01/2023, the formula is: 11/01/2023 + 10 = 11/11/2023. This type of formula does not work in M Code. In M Code, for most **operators**, both operands must have equivalent value types. In M Code you can subtract a date from a date because both are date values, but you cannot directly add a number to a date because a number value is not equivalent to a date value. Calculations like adding a number to a date are easy to perform in M Code, it is just works differently than in most other systems. If you do need to add a number to a date, you use the easy-to-interpret function Date.AddDays, or you add a duration value to the date, which is one of the exceptions to our rule. In addition, each M Code value has a different defined set of operators. For example, the only math operators defined for time and date values are addition and subtraction, whereas for number values the operators, multiplication, division, addition, subtraction and unitary plus and minus are defined, but not exponentiation operator (^). Exponentiation operations require the use of the Number.Power built-in function. Still further, when you need to add a time to a date in the worksheet and DAX, you use the addition operator. But in M Code you use the join operator. This means that it is crucial to learn the designated set of operators for each M Code value type. The key to applying operators to operands is to make sure that both operands have the same value type (exceptions are duration, null and date-time join values) and always check the list of defined operators for the given M Code value.

Also, don’t confuse data types with values when applying operators. For example, if you apply the Any data type to a column of number values and to a column of text values, you cannot apply an operator between the two columns because, although the data type is the same for each column of values, the underlying values are different. As another example, when you apply the data type Decimal, or Currency (Fixed in Power BI), or Whole number or Percentage to a column of number values, the underlying numbers are all still number values, so you are allowed to mix and match those data types with operators. This comes from the definition of an M Code number value: a number value is used for numeric and arithmetic operations. So regardless of whether a column of number values has the Decimal, Currency (Fixed), Whole number or Percentage data type applied, the number values are all still numbers that can be used in arithmetic operations. It is the M Code value that determines the type of operation, not the data type. Next, we want to consider the null value.

Category in order of precedence	Operators	Description	Example
	i	Identifier	Source
	@i	Recursive identifier	let Repeat = {0, @Repeat} in Repeat
	( )	Parenthesized expression	2 * (2+3) = 10
	Table[ColumnName]	Column Lookup	#table({"A", "B"}, {{2, 4}, {6, 1}})[B] = {4, 1}
	Table{RowNumber}	Row Lookup	#table({"A", "B"}, {{2, 4}, {6, 1}}){0} = [A = 2, B = 4]
	Function(...)	Function invocation	List.Sum({4, 1}) = 5
	{x, y,...}	List	{4, 1}
Primary	[i = x, ...]	Record	[A = 2, B = 4]
	+x	Identity	+3 = 3
	-x	Negation	-3 = -3
Unary	not	NOT Logical Test	not (-3 = -3) = false
Metadata	meta	Record of metadata	3 meta [Sales = "Low"] = 3
	*	Multiplication	2 + 3 / 3 * 1 = 3
Multiplicative	/	Division	2 + 3 * 3 / 1 = 11
	+	Addition	2 + 5 - 1 = 6
Additive	-	Subtraction	2 - 5 + 1 = -2
		Join text values	"You are " & "Rad!" = You are Rad!
		Join list values	{43} & {2, 17} = {43, 2, 17}
		Merge record values	[x = 3] & [y = 2] = [x = 3, y =2]
		Merge time & date values	#date(2024, 6, 1) & #time(8, 0, 0) = 6/1/2024 8 AM
Join/Merge/Append	&	Append table values	= #table({"B"},{{43}}) & #table({"B"},{{37}}) = #table({"B"},{{43},{37}})
	>	Greater than	2 > 3 = false
	>=	Greater than or equal	2 >= 2 = true
Comparison	<	Less than	"a"<"b" = true
(Logical)	<=	Less than or equal	10 <= 10 = true
Equality	=	Equal	"Quad" = "Quad" = true
(Logical)	<>	Not equal	"Quad" <> "Quad" = false
Type assertion	as	Value assertion	43 as number = 43. 43 as text = error.
Type conformance	is	Check value type	43 is number = true. 43 is text = false.
AND Logical Test	and	AND Logical Test	("Quad" = "Quad" and 100>5) = true
OR Logical Test	or	OR Logical Test	("Quad" = "Quad" or 100>5) = true
		Returns the value on the left	Example 1: null ?? 43 = 43
		if it is not null, otherwise it	Example 2: 28 ?? 43 = 28
		evaluates the value on the	These two formulas deliver same result:
Coalesce	??	right and returns it.	value ?? 43 = 43 and if value <> null then value else 43

Figure-07: M Code operators and order of precedence.

Identifiers, Keywords

Looking at Figure-08 (I added some hard returns and tabs to make the picture easy to see), the first thing you want to notice are the keywords: *let*, in, *each* and *type*.

Advanced Editor

DayTypeSalesQ

```
let
    Source = Excel.CurrentWorkbook(){[Name="DayTypeSales"]}[Content],
    AddDataTypes = Table.TransformColumnTypes(Source,{{"Date", type date},
        {"DayType", type text}, {"Sales", type number}}),
    #"Group By DayType" = Table.Group(AddDataTypes, {"DayType"},
        {{"Max Sale",  each List.Max([Sales]), type nullable number}})
in
    #"Group By DayType"
```

Figure-08: Query let expression named DayTypeSaloesQ.

Keywords are reserved words that have a predefined use, such as the word ***let*** which is reserved as the keyword to indicate the start of a let expression, the keyword ***in*** which ends the query steps and precedes the query output, the keyword ***each*** which allows a formula to calculate in each row of a table or list, and the keyword ***type*** which assigns data types. In M Code, keywords appear in blue. Figure-09 shows a list of some of the possible keywords.

and, as, each, else, error, FALSE, if, in, is, let, meta, not, otherwise, or, section, shared, then, TRUE, try, type

#binary, #date, #datetime, #datetimezone, #duration, #infinity, #nan, #sections, #shared, #table, #time

Figure-09: A list of reserved keywords in M Code.

The next thing you want to notice in Figure Ch01-08 are the identifiers such as DayTypeSalesQ, Source, AddDataTypes and #"Group By DayType". An **Identifier** is a name used to refer to an expression that delivers an M Code value (such as a number, a list, or a table).

Note: If Identifiers use spaces, you must house the identifier in quotes and place a # sign at the beginning. This is to distinguish an identifier from text, which requires quotes but not a # sign. To make your M Code easier to read, avoid spaces when creating identifiers. For example, use the identifier GroupByDayType rather than #"Group By DayType".

Note: Generalized Identifier are identifiers that allow spaces without the # sign and quotes. Generalized Identifiers are allowed in either:

1) The name of a column in a record literal like: [Boom Product = "Quad", Sales = 43]

2) The name of a column in a field access operator like: [Boom Product].

3) The name of a column in the lookup operator like: {[Boom Product = "Quad"]}

We'll learn much more about Generalized Identifier later in this book, but I wanted to list the full identifier details here is this section of the book.

Let expression

The **let expression** allows you to define **variables**, also known as **query steps**, and use them throughout the let expression to deliver a final query value (query output). let expressions can be used in any M Code to define variables, but most often the let expression is used to define a new query and deliver a query output. Every time you start a new query and select commands from the user interface, Power Query automatically records the commands you select in a let expression. The syntactical rules for a let expression are listed on the next page:

The syntactical rules for the let expression are listed below in three different forms: bulleted list, query let expression example (Figures-10) and cheat sheet with notes (Figure-11).

- Start the let expression with lower case *let*.
- Each variable, or query step, starts with an identifier, followed by an equal sign, then the expression. The identifier represents the expression throughout the query.
- Variables are usually used in the first argument of the function in the following step to modify the previous step, but can be used anywhere throughout the let expression. Variables cannot be used outside the let expression.
- Each variable is followed by a comma. The comma allows the let statement to deliver an intermediate value, such as a table that can then be acted on further or used in other locations in the query.
- The last variable does not end with a comma. By putting no comma, the let expression expects the keyword *in* to come next, followed by the query output.
- End the let expression with lower-case keyword *in* followed by the identifier for the final value to be delivered by the query. The output of the query is almost always the identifier of the last query step; however, the output can be any of the variables defined in the let expression, any other query in the file, or any expression.
- You can add non-executable comments to your M Code by add two forward slashes, like //, for single-line comments, or multi-line comments that begin with /* and end with */.

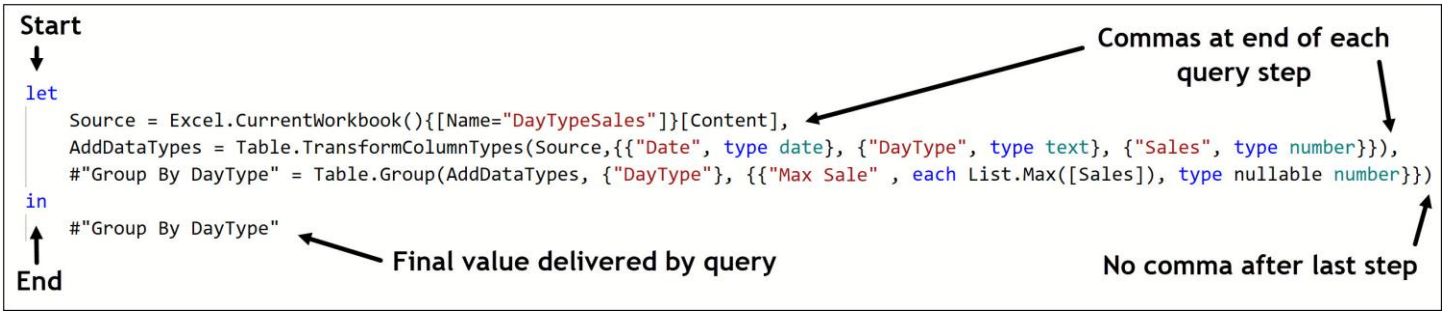


Figure-10: Rules for let expression.

```
let
   VariableName1 = Expression,
    #" Variable Name 2" = Expression,
    VariableName3 = Expression
in
    Output
      (Output is usually the last variable name, but can
      be any variable, query name or expression)

// one-line comment
/* multiple line comments */
```

Figure-11: Cheat sheet for the syntax of the let expression.

Null Value

A **null value** represents the absence of an M Code value. The literal syntax for hard coding a null value into M Code is *null*, all lowercase. When using a null, an operator and any of the other value types in an expression the result is a null value. For example, 5 + null = null, "Quad" & null = null, 43 >= null = null. As I mentioned in the previous section, the null value is an exception to the rule that operators must work on equivalent value types, but the result will always be null.

Logical values

A **logical value** is one of two possible Boolean values: true or false. The literal syntax for hard coding logical values is *true* or *false*, all lowercase. Figure-12 shows the operators defined for logical values. To type the literal syntax for a true value and build two formulas using logical values, follow

Description	Operator
Greater than	>
Greater than or equal	>=
Less than	<
Less than or equal	<=
Equal	=
Not equal	<>
Logical tests	and, or, not
Coalesce	??

Figure-12: Operators that are defined to work between two logical values.

Text Value

A **text value** is a string of Unicode characters. The literal syntax for hard coding a text value into M Code is to put the characters in quotes, like "Quad" for the word Quad. Figure-13 shows the operators defined for text values.

Description	Operator
Equal	=
Not equal	<>
Greater than or equal	>=
Greater than	>
Less than	<
Less than or equal	<=
Concatenation	&
Coalesce	??

Figure-13: Operators that are defined to work between two text values.

Number values

A **number value** is used for numeric and arithmetic operations. The literal syntax for some possible number values is shown here:

- 1,210
- -43
- 20.3524
- 12.5874441875555
- 0.025 or 2.5%
- 0.00157 = 1.57E-03
- 2560000 = 2.56E+06
- #infinity = like from the operation 1/0
- -#infinity = like from the operation -1/0
- #nan = like from the operation 0/0

A number is represented with at least the precision of a Double (but may retain more precision). The Double representation is congruent with the IEEE 64-bit double precision standard for binary floating-point arithmetic defined in [IEEE 754-2008]. The Double representation can be zero or can range from -1.797693134862315E+308 to -2.225073858507201E-308, or from 2.225073858507201E-308 to 1.797693134862315E+308.

Figures 14 and 15 show the operators defined for number values.

Description	Operator
Greater than	>
Greater than or equal	>=
Less than	<
Less than or equal	<=
Equal	=
Not equal	<>
Sum	+
Difference	-
Product	*
Quotient	/
Coalesce	??
Unary plus	+x
Negation	-x

Figure-14: Operators that are defined to work between two number values.

Operator	Left Operand	Right Operand	Meaning
x * y	duration	number	N times a duration
x * y	number	duration	N times a duration
x / y	duration	number	Fraction of a duration

Figure-15: Number values can perform three operations with duration values.

Whereas Figure-14 shows operators defined for number values, Figure-15 shows that for three operations, the duration value can be used with number values. As I mentioned in the operator section of this chapter, the duration value is an exception to the rule that operators must work on equivalent value types. We have not studied the duration value yet, but I wanted to list the duration related operations in this section so that when you need to look up the operators for numbers, they are all in one place. Briefly, a duration value is an amount of time in units of days, hours, minutes, and seconds. So, the fact that you can multiply a number value by a duration or divide a duration by a number makes sense. For example, 2 times the duration of two days would yield 4 days. We will see how to make this type of duration and number value calculation in the upcoming duration value section.

Date, Time, Datetime, Datetimezone and Duration

The time and date related values in M Code are: date, time, datetime, datetimezone and duration. When you hard code each of these values you use an intrinsic functions to deliver the value. A description of these values, an example of each intrinsic function, and several expression examples follows in the next five sections. Because the duration value interacts with the other date and time related values, we will look at the duration value first.

Duration value

A **duration value** is encoded as a serial number that represents an amount of time in units of days, hours, minutes, and seconds. A duration can be negative or positive. There is no literal syntax for durations, but you can use an intrinsic function to hard code a duration value, as shown here:

```
Intrinsic function: =#duration(days, hours, minutes, seconds)
```

```
Example: =#duration(1,0,10,0) = 1:00:10:00 = 1 day and 10 minutes
```

Duration values are important in M Code because they can interact as an operand with the date, time, datetime and datetimezone and number values. For example, if you want to add 10 days to a date, you cannot use the formula 1/1/2023 + 10 because the value types for the date and number are not equivalent. But you can use the formula 1/1/2023 + #duration(10,0,0,0) and the result will be 1/11/2023. As a second example, if you want to add 2 hours to a datetime value, you cannot use the formula 1/1/2023 8 AM + 2:00 because the value types for the datetime and time are not equivalent. But you can use the formula 1/1/2023 8 AM + #duration(0,2,0,0) and the result will be 1/1/2023 10 AM. Still further, if you subtract an earlier date from a later date to get number of days, the result will be a duration. For example, 1/11/2023 - 1/1/2023 = #duration(10,0,0,0), or 10 days. Finally, if you have an amount of time as a duration and you want to double that time, because multiplication is defined between numbers and durations, you can simply multiply the time duration by two. If you are working in the app Dataflow, Microsoft's documentation says that duration values are not supported in Dataflow. I have found that I can use a duration values embedded in formulas if the formula does not deliver a duration value and the formula does define a duration data type. Duration values do not exist in the worksheet or Data Model. If you load them to the worksheet or Data Model, they are converted to datetime values. Figure-16 shows the operators defined for duration values. The lists of defined operators that work between duration values and other values are shown later in this chapter.

Description	Operator
Equal	=
Not equal	<>
Greater than or equal	>=
Greater than	>
Less than	<
Less than or equal	<=
Coalesce	??
Sum	+
Difference	-

Figure 16: Operators that are defined to work between two duration values.

Date value

The format for dates depends on the regional settings for your computer. For example, in the US a date is shown as mm/dd/yyyy and in France a date is shown as dd/mm/yyyy. I will use the US date format for all examples in this book. However, the Using Locale feature that allows us to work with dates from other locals and other formats. M Code date values are different than dates in the Excel worksheet. In the beginning of spreadsheet history, Lotus 1-2-3, and then Excel incorrectly assumed that the year 1900 was a leap year and that the day 2/29/1900 existed. This error persists in worksheets today. In addition, the earliest possible date in the Excel worksheet is 1/1/1900. M Code date values remedies both deficiencies. Date values in M Code do not list 2/29/1900 as a valid date. In addition, M Code date values span from 1/1/0001 to 12/31/9999 (Common Era on the Gregorian calendar).

Note: Although you can perform date math calculations in Power Query on all Gregorian calendar dates, if you start with whole numbers and apply the date data type, the smallest number is - 657,434 which corresponds to the date 1/1/0100.

If you are using dates before 1/1/1900 in a query and you load the query results to the Data Model in Power Pivot or Power BI, the dates will work as valid dates; however, if the same dates are loaded to the Excel worksheet, the dates before 1/1/1900 will be loaded as text dates rather than valid date values. As in the Excel worksheet, M Code date values are encoded with serial numbers that help formulas perform date math. However, M Code date serial numbers are different than the date serial numbers in the Excel worksheet.

As shown in Figure-17, serial numbers in the worksheet and in M Code are the same for dates on 3/1/1900 or after, but the dates between 1/1/1900 and 2/28/1900 are off by one. In the worksheet, dates entered before 1/1/1900 are considered text values, with a left alignment, and therefore do not have serial numbers, as shown in Figure Ch02-29. In M Code, date serial numbers are negative starting with the date 12/29/1899 and increment by -1 until the first possible date. Notice that the worksheet serial number for 2/29/1900 is 60, but the M Code serial number for 2/28/1900 is 60. This bit of Microsoft cleverness allows you to import the date 2/29/1900 and it will be interpreted as 2/28/1900, or import 2/28/1900 with a serial number 59 and it will be interpreted as 2/27/1900 because its serial number is 59. Lastly, in M Code, the negative and positive date serial numbers allow you to make accurate date formulas, such as:

```
End Date – Start Date = Number of Days Between Two Dates.
```

For example, 12/29/1899 – 12/28/1899 = -1 - -2 = 1 day. As another date calculation example, if you create an expression to calculate 12/28/1899 + 2 days , you get -2 + 2 = 0, which is the correct date of 12/30/1899.

Power Query Serial Numbers	Power Query Dates
2958465	12/31/9999
45114	7/7/2023
366	12/31/1900
365	12/30/1900
61	3/1/1900
60	2/28/1900
59	2/27/1900
3	1/2/1900
2	1/1/1900
1	12/31/1899
0	12/30/1899
-1	12/29/1899
-2	12/28/1899
-102589	2/12/1619
-255899	5/15/1199
-644000	10/13/0136

Excel Serial Numbers	Excel Dates
2958465	12/31/9999
45114	7/7/2023
366	12/31/1900
365	12/30/1900
61	3/1/1900
60	2/29/1900
59	2/28/1900
3	1/3/1900
2	1/2/1900
1	1/1/1900
Text	12/31/1899
Text	12/30/1899
Text	12/29/1899
Text	12/28/1899
Text	2/12/1619
Text	5/15/1199
Text	10/13/0136

Figure 17: Some of the possible dates and serial numbers in Power Query and the worksheet.

The definition of a date value is: a **date value** is encoded as a serial number that represents the number of days since epoch, starting from January 1, 0001 Common Era on the Gregorian calendar. The maximum number of days since epoch is 3652058, corresponding to December 31, 9999. There is no literal syntax for date values, but you can use an intrinsic function to hard code date values, as shown here:

```
Intrinsic function: =#date(year, month, day)
```

```
Example: =#date(1598, 1, 15)= 1/15/1598
```

Figures-18 and 19 show the operators defined for date values.

Description	Operator
Greater than	>
Greater than or equal	>=
Less than	<
Less than or equal	<=
Equal	=
Not equal	<>
Coalesce	??

Figure-18: Operators that are defined to work between two date values.

Operator	Left Operand	Right Operand	Meaning
x + y	date	duration	Date offset by day duration
x + y	duration	date	Date offset by day duration
x - y	date	duration	Date offset by negated day duration
x - y	date	date	Duration between dates
x & y	date	time	Merged into datetime

Figure-19: These operators permit one or both of their operands to be a date value.

Whereas Figure-18 shows operators defined for two date values, Figure-19 shows the operators allowed between duration and date values, two date values, and date and time values. In the above table, the first two rows show that you can add a day duration value to a date value to get a new date value. For example, this formula would work:

```
= #date(2023,1,1) + #duration(8,0,0,0) = 1/9/2023
```

The third row shows that you can subtract a day duration value from a date value to get a new date value. However, subtracting a date value from a duration value is not defined. For example, this formula would work:

```
= #date(2023,1,1) - #duration(8,0,0,0) = 12/24/2022
```

The fourth row shows that when you subtract a date from a date you will get a day duration, where the result can be positive or negative. For example, this formula would work:

```
= #date(2022,12,24) - #date(2023,1,1) = -8:00:00:00
```

The last row shows that you can merge a date value and a time value to create a datetime value using the join operator. In the worksheet and DAX, date and time values are merged using the addition operator, rather than the join operator. Also, notice that the order of operands matters. Whereas in the worksheet and DAX you can add a time to a date or a date to a time. In M Code you can only merge a date value with a time value with the formula: date & time.

Time value

As with time values in the worksheet and DAX, an M Code **time value** is encoded as s serial number between 0 and 1, which represents the proportion of a 24 hour day, such as 8AM = 8/24 = 1/3 =0.3333. There is no literal syntax for time values, but you can use an intrinsic function to hard code time values, as shown here:

```
Intrinsic function: =#time(hour, minute, seconds)

Example: =#time(11,10,57) = 11:10:57
```

Figures 20 and 21 show the operators defined for time values. Time values have the same set of defined operators as date values. For example, you can offset a time value with a time duration value to get a new time value using addition and subtraction, you can calculate the difference between to time values to get a duration value and you can merge a date and time using the join operator.

Description	Operator
Greater than	>
Greater than or equal	>=
Less than	<
Less than or equal	<=
Equal	=
Not equal	<>
Coalesce	??

Figure-20: Operators that are defined to work between two time values.

Operator	Left Operand	Right Operand	Meaning
x + y	time	duration	Time offset by time duration
x + y	duration	time	Time offset by time duration
x - y	time	duration	Time offset by negated time duration
x - y	time	time	Duration between times
x & y	date	time	Merged into datetime

Figure-21: These operators permit one or both of their operands to be a time value.

Datetime value

In M Code, a **datetime value** is encoded as a serial number that represents the number of days since 1/1/0001 plus any decimal amount that represents the proportion of a 24 hour day. There is no literal syntax for datetime values, but you can use an intrinsic function to hard code datetime values, as shown here:

```
Intrinsic function: =#datetime(year, month, day, hour, minutes,
seconds)

Example: =#datetime(2021,10,1,11,10,57) = 10/01/21 11:10:57 AM
```

Figure-22 and Figure-23 show the operators defined for datetime values. The operators defined for datetime values are the same as the ones defined for date and time values except that you cannot use the join operator.

Description	Operator
Greater than	>
Greater than or equal	>=
Less than	<
Less than or equal	<=
Equal	=
Not equal	<>
Coalesce	??

Figure-22: Operators that are defined to work between two datetime values.

Operator	Left Operand	Right Operand	Meaning
x + y	datetime	duration	Datetime offset by duration
x + y	duration	datetime	Datetime offset by duration
x - y	datetime	duration	Datetime offset by negated duration
x - y	datetime	datetime	Duration between datetimes

Figure-23: These operators permit one or both of their operands to be a datetime value.

Datetimezone value

A **datetimezone value** is encoded as a serial number that represents a UTC (Universal Coordinated Time) datetime value with a time-zone offset. The time-zone offset is set in the last two arguments, offset-hours and offset-minutes. The offsets can be negative or positive. Datetimezone values do not exist in the worksheet or Data Model. If you load them to the worksheet or Data Model, they are converted to datetime values. There is no literal syntax for datetimezone values, but you can use an intrinsic function to hard code datetimezone values, as shown here:

```
Intrinsic function: =#datetimezone(year, month, day, hour, minute,
second, offset-hours, offset-minutes)

Example: =#datetimezone(2021,10,1,11,10,57,09,0) = 10/01/21 11:10:57
AM + 9 hours
```

Figure-24 and Figure-25 show the operators defined for datetimezone values. The operators defined for datetimezone values are the same as the ones defined for date, time and datetime values except that you cannot use the join operator.

Description	Operator
Greater than	>
Greater than or equal	>=
Less than	<
Less than or equal	<=
Equal	=
Not equal	<>
Coalesce	??

Figure-24: Operators that are defined to work between two datetime values.

Operator	Left Operand	Right Operand	Meaning
x + y	datetimezone	duration	Datetimezone offset by duration
x + y	duration	datetimezone	Datetimezone offset by duration
x - y	datetimezone	duration	Datetimezone offset by negated duration
x - y	datetimezone	datetimezone	Duration between datetimezones

Figure-25: These operators permit one or both of their operands to be a datetime value.

So far in this chapter we have studied these nine M Code values: null, logical, text, number, date, time, datetime, datetimezone and duration. These nine values are called **primitive values** because they are single part values. Primitive values are what make up most of the data that we use to perform data analysis. Next, we want to look at M Code values that have more than one part: tables, records, and lists.

Tables, records, and lists

Tables, records, and lists are M Code values that can hold more than one value. Most of the time, tables, records, and lists hold and store the primitive values, or data, that we use to perform our data analysis. However, they can also hold any of the other M Code values such as functions and binary (files), and they can even hold values such as tables, records, and lists. Yes, that is correct, a table can hold other tables, a list can hold other lists, and records can even hold other records! In addition, there is an important relationship between these three M Code values. If you start with a table value and lookup (extract, or select) a full row, the result is a record value. If you start with a table value and lookup (extract, or project) a full column, the result is a list value. The table itself does not contain record values and list values; the table contains rows and columns. This is an important distinction because if you try to add a table column of numbers, you will get an error; but if you lookup the table column and return a list of numbers, then you can add the list of numbers without an error with a function like List.Sum. What you want to understand here is that there is a relationship between the three values and that extracting a table column yields a list value and extracting a table row yields a record value. Next, we want to take a closer look at each of the three values: list, record, and table.

List value

A **List value** is a sequence of M Code values. List values can hold any of the 15 M Code Values, including lists. A list can contain one or more values. A list is not limited to a single value type. For example, you can have a list that contains number values, text values and date values. You can think of a list as a column in a table, but technically it is not a list value until the column is extracted from the table by doing column lookup. There is no intrinsic function to create a hard coded list. To create a hard coded list value, you use curly brackets to house the list, known as **initialization syntax**, and then separate the elements in the list with commas. A hard coded list is known as a **list literal**. An example of a hard coded list with two text values is shown here:

```
={"Quad","Aspen"}
```

As shown in the below figure, A list within a list is often used to allow you to enter multiple items into one location. For example, in the Table.Group function in the third argument uses a list with a list to enter the three paerts for each aggregate calculation: column name, aggregate formula and data type.

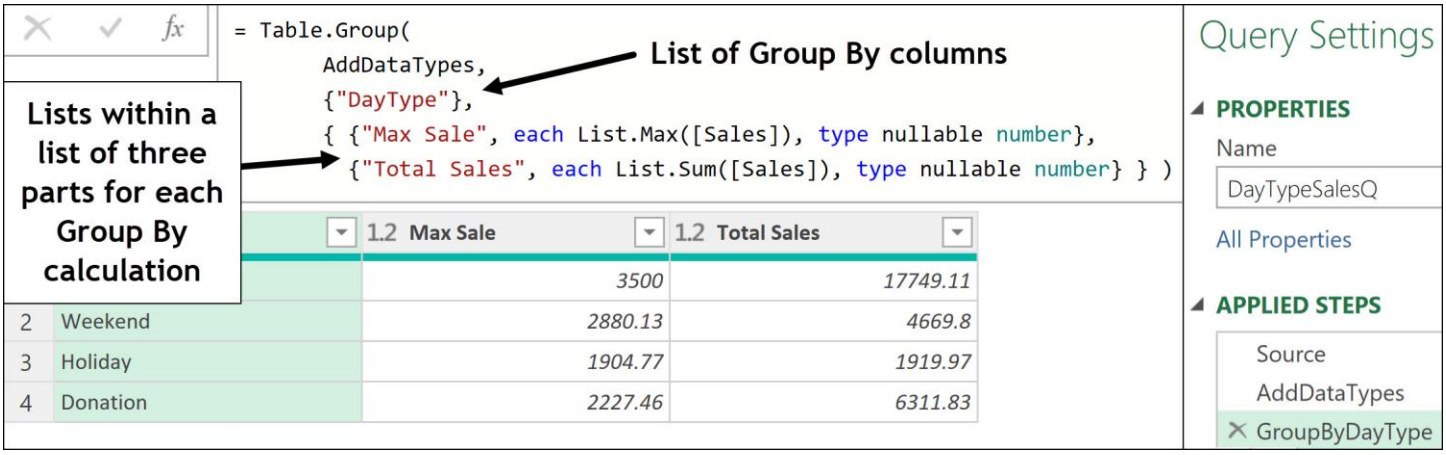


Figure-26: The second and third arguments in the Table.Group function use lists.

In addition, when you extract a column from a table, a list of values is returned. For example, when a column of numbers is extracted from a table, it is returned as a list of numbers. This is why the aggregate functions all have the word List before the name of the function, like: List.Sum, List.Max, List.Count, List.StandardDeviation.

Still further, if you use the dot-dot operator, you can create a sequence of numbers or latters as shown in the below figure.

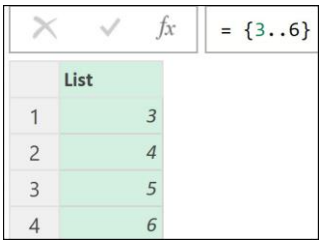


Figure-27: The dot-dot operator allows you to create a list sequence of numbers.

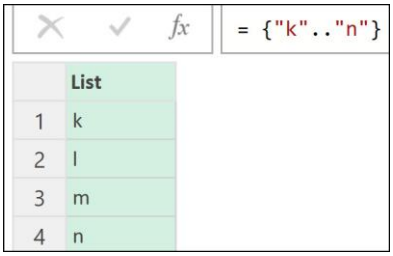


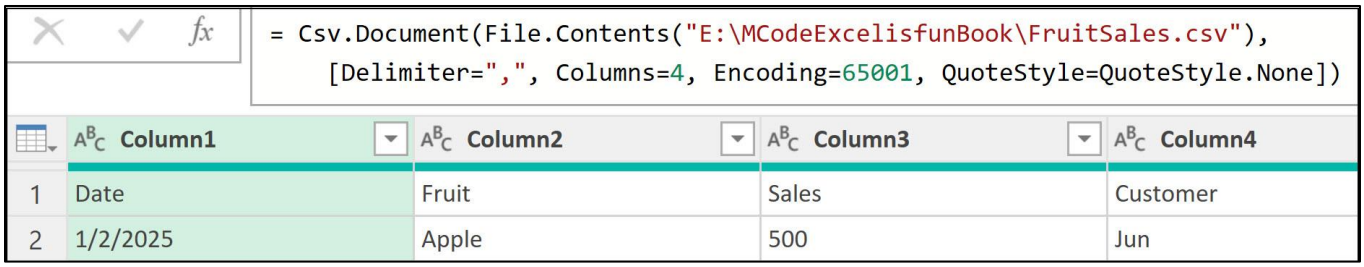
Figure-28: The dot-dot operator allows you to create a list sequence of letters.

Record value

A **Record value** is an ordered sequence of columns with M Code values entered in each column. Records can hold any of the 15 M Code Values, including records. You can think of a record as a row in a table, but technically it is not a record value until the row is extracted from the table by doing row lookup. There is no intrinsic function to create a hard coded record. To create a hard coded record value, you use square brackets to house the record, known as initialization syntax, and separate each pair of columns and values with commas. A hard coded record is known as a **record literal**. The identifier for the column name does not need quotes or the # sign. The column name is followed by an equal sign and then the value. An example of a hard coded record with two columns and two values is shown here:

```
=[Boom Product = "Quad", Sales = 43].
```

Records are often used inside functions to allow you yo enter argumenst, as shown below in the Csv.Document function:



Ch06-03: A record with parameters can be specified in the columns argument.

Table value

A **Table value** has column names in the first row and records of values in subsequent rows. As with any data related tool, tables are the heart of any data analysis process because they hold all the data that we analyze. Tables can hold any of the 15 M Code Values, including tables. You can think of tables as being made up of record values in rows and lists values in the columns. However, technically a row in a table is not a record value until you extract it from the table, and a column in a table is not a list value until you extract it from the table. Extracting rows and columns from tables is a very common task and we will learn about later when we learn M Code lookup.

The table intrinsic function can have as many columns or records as you would like. In the first argument of the table intrinsic function, you list the column names and in the second argument you list the records. There are two different syntactical structures for the table intrinsic function. Here is an example of both:

```
=#table(ColumnNames , Records)
```

Table with data types:

```
= #table(
    type table [Boom Product = text, Sales = number],
    {{"Quad",43}, {"Aspen",35}})
```

Table with no data types:

```
=#table(
    {"Boom Product", "Sales"},
    {{"Quad",43},{"Aspen",35}})
```

Binary value

M Code binary values represent a sequence of bytes, such as Excel files, text files, xml files, images, audio files, and other data sources. The value type is important to have in M Code so that we can accomplish tasks such as importing multiple Excel or text files and extracting the data from within those files. You will learn how to work with binary values in chapters 6 and 7. As a preview of those chapters, Figure-29 shows the Content column that contains binary values in each row and a binary data type. This column contains Excel files. In row #2 the Excel file is named GrassValley.xlsx.

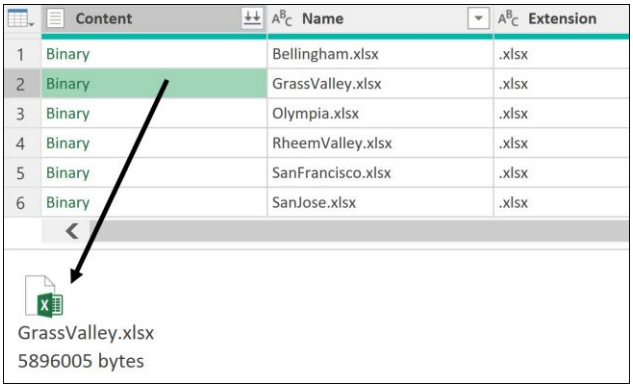


Figure-29: The binary value type allows you to import Excel files that contain data.

M Code Lookup Formulas

Now that we have studied the fundamental building blocks: expressions, let expressions, queries, built-in functions, custom functions, M Code values and data types, the last building block we need to study is M Code lookup. M Code lookup allows you to lookup a value, a row as a record, a column as a list or one or more columns as a table.

> **Note**: From relational algebra, database theory and the M Code Specifications Guide:
>
> Projection (π, pi)= pick a column (attribute) from a table (relation)
>
> Selection (σ, sigma) = select a row (tuple) from a table (relation)

For example, common lookups are:

- Two-way lookup to get a table from an SQL database or an Excel file.
- Two-way lookup to get a sales discount rate.
- Column lookup to get a list of values for an aggregate calculation.
- Row lookup to get record with the attributes for a file.

We will see the above examples and many more throughout the rest of the book.

The two types of lookup that we will learn about are exact match and approximate match lookup. The definitions are listed in the next two paragraphs.

Exact match lookup involves taking a **lookup value** and searching for an exact match in a **match column** that contains a unique list, and when a match is made a value is retrieved from the same corresponding position in the a **return column**. For example, the lookup value "Quad" would match "Quad", but not with "Quad " (extra spaces at the end).

Approximate match lookup or **Exact match or next smaller lookup** involves taking a lookup value and searching for an exact match or next smaller value in a **match column,** and when a match is made a value is retrieved from the same corresponding position in the return column. For example, in a match column of {0, 500, 1000, 2500} the lookup value 774 would match 500 because there is no exact match, and the next smaller value is 500.

In M Code, there is a built-in method to perform the common task of exact match lookup, but no built-in method for approximate match lookup. Both M Code and DAX lack a built-in method for approximate match lookup. However, in the Excel worksheet there are many functions that can do both. Later in this chapter I will teach you how to build a custom function for approximate match lookup for tasks such as looking up a tax or commission rate.

Finally, because M Code is base zero, which means that row one is 0, row two is 1, row three is 2 and so on, we must be aware of this when creating lookup formulas. Figure-30 illustrates base zero and base one.

M Code is base zero		**Worksheet & DAX are base one**	
	Products		**Products**
Row 0	Quad	Row 1	Quad
Row 1	Aspen	Row 2	Aspen
Row 2	Carlota	Row 3	Carlota

Figure-30: In M Code row one in tables and lists is represented by the number zero (0).

For those of you who know how to perform two-way exact match lookup with the worksheet INDEX function, you will understand M Code lookup easily because the two methods are very similar. Both methods start with a table that has rows and columns, then you provide a row number and a column number, and the intersecting value is retrieved. Figure-31 shows the two-way exact match lookup formulas for both M Code and for the INDEX function. For example, as shown in Figure-32, if you start with the Product table and you need to lookup the retail price for the Aspen product, you start by finding the Aspen product in the Product column, this is row #2, then you find the Retail Price column, this is column #3, and the intersecting price of 26.95 is returned. As shown in Figure-31, for the INDEX function you enter the table name, the row number, and the column number into the function, and it retrieves the intersecting price of 26.95. Still looking at Figure-31, in M Code, you start with a table, then you put the row number in curly brackets, called the **row positional index operator** or **row index operator**, and then you put the column name in square brackets, called the **field access operator**, and it retrieves the intersecting price of 26.95.

M Code Lookup:	**Table { RowNum } [Column Name]**
	Product { 1 } [Retail Price] = 26.95
Excel INDEX:	**INDEX(Table , RowNum , ColNum)**
	INDEX(Product , 2 , 3) = 26.95

Figure-31: M Code lookup and the INDEX function work similarly.

Products	Supplier	Retail Price
Quad	Gel Booms	43.69
Aspen	Colorado Inc.	26.95
Carlota	Gel Booms	27.95
Sunset	Colorado Inc.	32.95

Figure-32: Product lookup table.

There are two types of M Code exact match lookup: row index lookup and key math lookup. The lists below and Figure-33 present the rules for M Code exact match lookup.

Rules for row index lookup

- Lookup Syntax: **Table { Row Index Number }[Field Name]**
- **Row index lookup** performs an exact match two-way lookup based on a hard coded row number in the positional index operator, and a column name in the field access operator, to retrieve a value from a table at the intersection of the row number and designated column name. The hard coded row number does not change when the sort or content of the column changes. The lookup is static because the row number does not change when the data changes.
- Examples of row index lookup:
  - Source{0}[Content] gets the first value from the Source table Content column.
  - Source{0} gets the first row from the Source table as a record value.
  - Source[Content] gets the Content column from the Source table as a list value.
  - Source[[Content]] gets the Content column from the Source table as a table.
  - {43,86}{0} gets the first item from the list, 43.
  - {43,86}{2} yields an error because there is no third item.
  - {43,86}{2}? delivers a null instead of an error (optional operator: ?).
  - {43,86}{2}?? 5 delivers 5, instead of an error. (coalesce operator: ??).

Rules for key match lookup:

- Lookup Syntax: **Table{[Field Name = LookupValue]}[Field Name]**
- **Key match lookup** performs an exact match two-way lookup based on an equality logical test between a lookup value and a match column (key column) that contains a unique list of values. The equality test must be inside square brackets, called the **lookup operator**. The lookup operator must be inside the positional index operator. The logical test will dynamically determine the row position when the column is sorted or the data changes. The field name in the field access operator determines the column position. Then the intersecting value is retrieved from the table.
- If there are duplicate values in the key column, an error is produced.
- Examples of key match lookup:
  - Source{[File=”DD”]}[Content] gets the value from the Source table Content column that corresponds to the row position of DD in the File column.
  - Source{[File=”DD”,Col=”G”]}[Content] gets the value from the Source table Content column that corresponds to the row position of DD in the File column and G in the Col column.
  - Source{[File=”DD”]} gets the record from the Source table that corresponds to the row position of DD in the File column.

- Source{[POP="Z"]} yields an error because there was more than one Z in the POP column. The error reads: "The key matched more than one row in the table".
- Source{[File="YY"]} yields an error because there were no YY value in File column. The error reads: "The key didn't match any rows in the table".

1) **Row Index Lookup** formula has this structure: (Static lookup)

=Table { Row Index Number } [Field Name]

{ } = Row Positional Index Operator

[ ] = Field Access Operator

2) **Key Match Lookup** formula has this structure: (Dynamic lookup)

=Table { [Field Name = Lookup Value] } [Field Name]

Key column (Match column)

[ ] = Field Access Operator

{ } = Row Positional Index Operator

[ ] = Lookup Operator when used inside of { }

Figure-33: Two types of exact match lookup.

Figure-34 shows the syntax for looking up a row as a record, a column as a list, a column, and one or more columns.

Lookup row to retrieve record

=Table { Row Index Number }

=Table { [Field Name = Lookup Value] }

Lookup column to retreive list

=Table [Field Name]

Lookup column

=Table [[Field Name]]

Lookup columns

=Table [[Field Name], [Field Name2]]

Figure-34: Common lookup tasks.

Here are the types of lookup we will learn about in the upcoming examples:

- Row index lookup to retrieve a record.
- Row index two-way lookup to retrieve a value.
- Key match lookup to retrieve a record.
- Key match two-way lookup to retrieve a value.

- Key match lookup yields error when there are duplicates in column.
- Key match lookup yields error when there is no match.
- Lookup a column to return a list.
- Lookup column to return a one column table.
- Drill down lookup when there is no primary key.
- Drill down lookup when there is primary key.
- Check for primary key.
- Add a primary key to a table.

Drill Down, lookup formulas and primary keys

Drill Down is a user interface command that allows you to lookup a value from a table, list or record. When you use drill down on a particular cell or column, the M Code value is extracted as a new step in the query. The drill down scenarios are as follows:

- When you use Drill Down on a cell in a table with **no primary key**, **row index lookup** is used to extract the value.
- When you use Drill Down on a cell in a table **with a primary key**, **key match lookup** is used to extract the value.
- Primary keys are created when you use functions like Excel.CurrentWorkbook, Excel.Workbook, Sql.Database and other functions. A primary key is also created when you use the Remove Duplicate feature.
- When you use Drill Down on a column in a table, column lookup is used to exact a list from the column.
- When you use Drill Down on a cell in a record, a column lookup formula is used to exact the value from the record.
- When you use Drill Down on a value in a list, row index lookup is used to extract a value from the list.

What if we have a table and we are not sure whether it has a primary key? Is there a way to check? The crazy thing is that Microsoft did not put a button or command in the user interface to determine if a table has a primary key. The only way to check if there is a primary key in a table is to use the **Table.Keys function**. This function takes a table as its only argument, and reports if there is primary key. If there is a primary key, it reports the column name/s that make up the primary key. If there is no primary key it returns an empty list.

Lastly, if you want to add a primary key to a table, you can use the **Table.AddKey function**. The arguments for this Table.AddKey function are listed here:

```
Table.AddKey(
      table as table,
      key column/columns as list,
      isPrimary as logical ) as table
```

Join operations used by Merge feature

When making data transformations, the join operation to merge tables is a common. In this section I will cover the concepts of the join operation and then we will move on to a few examples. A **join operation** connects two tables based on a primary key column and a foreign key column to merge the two tables into a new table. The primary key table has a primary key column with a unique list of values, and the rest of the columns contain attributes for each value in the primary key. As shown in Figure-37, an example of a primary key table is the product table, where product ID is the primary key, and the attribute columns are product name, cost and price. In the same figure below, an example of a foreign key table is the sales transaction table, where the product ID column is the foreign key, and the remaining columns are details about the transactions such as date of sale, customer ID and units sold. As shown in Figure-38, Venn diagrams illustrate the six types of common joins used in data analysis to determine the structure of the merged table. Figure-37 illustrates a **left outer join**, where all records from the left are joined with only matching Price column records from the right to create a new merged table. The Venn diagram illustrates this by highlighting the full circle of the left, but only the intersection of the left and right circles on the right. Only product prices that have product IDs on the left are in the merged table. The one Sunshine record on the right has no matching product ID on the left, so the price for that product is not in the merged table.

Left Table: Sales Transaction Table

Foreign Key
Many Side

Date	ProductID	CustomerID	UnitsSold
2/18/2024	Q69	SW01	168
1/29/2024	C43	ITW02	144
2/6/2024	C43	PCC01	120
1/5/2024	Q69	PCC01	120

Right Table: Product Table

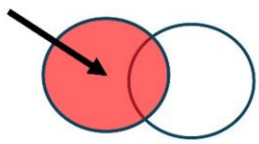
Primary Key
One Side

ProductID	Product	Cost	Price
C43	Carlota	11.50	26.00
S11	Sunshine	9.50	19.95
Q69	Quad	22.00	43.00

Left Outer Join:

(All from the left only matching from the right)

Classic Lookup into Right



Left Outer

New Merged Table:
Using Left Outer Join on Product ID Column:

Date	ProductID	CustomerID	UnitsSold	Price
2/18/2024	Q69	SW01	168	43.00
1/29/2024	C43	ITW02	144	26.00
2/6/2024	C43	PCC01	120	26.00
1/5/2024	Q69	PCC01	120	43.00

Figure-37: Left outer join to create a sales transaction table with a price column.

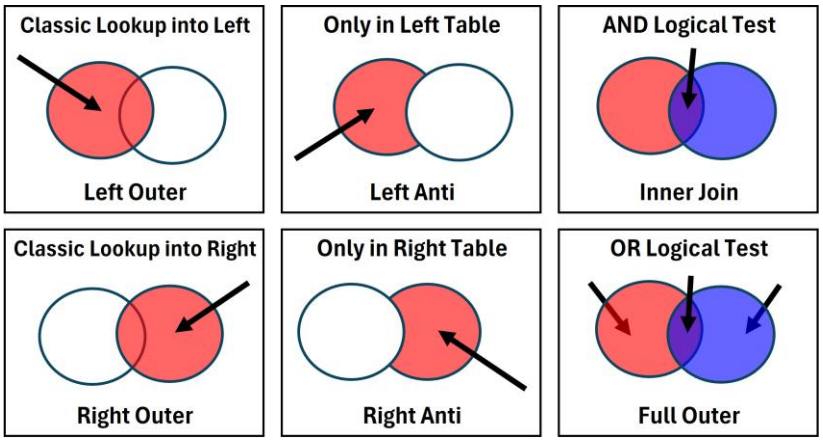


Figure-38: Six common types of joins.

Figure-39 illustrates a **left anti join**, where only records from the left that are not in the right are used to create the new merged table. The Venn diagram illustrates this by highlighting the left table without the overlap (names that are in both). Because the names Sioux Noline, Chantel Xo and Ty Mims are in both tables, those names do not appear in the merged table.

As shown in Figure-38, the **right outer join** and **right anti join** are mirror images of the left out join and left anti join, respectively. Because it is easy to switch tables from right to left and left to right, most of the time people orientate the tables accordingly and just perform left outer and left anti joins.

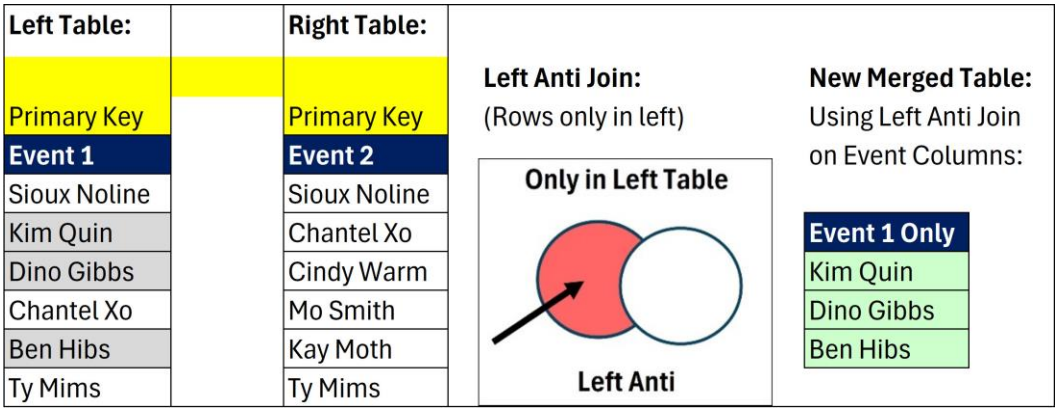


Figure-39: Left anti join to create a table with only people from event 1.

Figure-40 illustrates an **inner join**, where only records that are in both tables are used to create the new merged table. The Venn diagram illustrates this by highlighting only the overlap between the left and right circles. The overlap represents an AND logical where you get a matching name in the left table, TRUE, and a matching name in the right table, TRUE. Because the names Kim Quin, Dino Gibbs and Ben Hibs are in the left table only and the names Cindy Warm, Mo Smith and Kay Moth are in the right table only, none of those names appear in the final table.

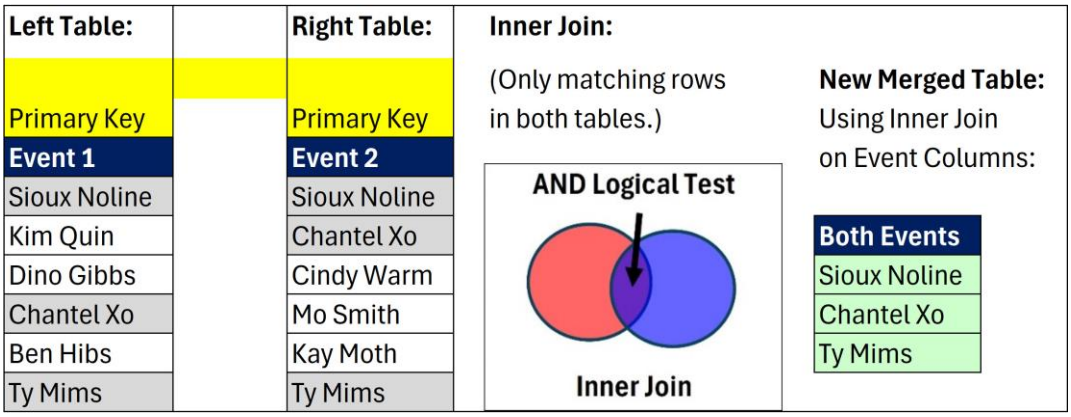


Figure-40: Inner join to create a table with people who went to both events.

In my data analysis experience, the left outer join is the most common join in data transformations. This is because it is common to have to pull data from lookup tables (also known as dimension or attribute tables) into the table with the data to be summarized, often called a fact table. The left outer join is similar to the worksheet functions, XLOOKUP and VLOOKUP, and to relationships in the Power Pivot and Power BI Data Model (Semantic Model). Whether you are working in M Code, the worksheet or the Data Model, the left outer join is the most common type of join. However, I use the left anti and inner joins regularly in data transformations also.

Custom function value

Perhaps the most exciting and useful M Code value is the function value, often called a custom function. The **function value** or **custom function** allows you to create a function that defines variables and the mapping for those variables to deliver an M Code value. In Excel, the parallel function to create function values is the LAMBDA function. In M Code, function values are particularly helpful for data transformations that you must repeat often. The function-expression syntax rules for creating a custom function are listed below and shown in Figure-41:

- Variables names are separated by commas and housed in parentheses. The rules for naming variables are the same as the rules for naming identifiers: no keywords, no spaces and if you do have spaces use the # symbol and quotes. The variable names show up as function arguments when you use the custom function.
- The go to operator comes after the defined variables and indicates that everything after the go to operator is the mapping of the variables, or formula to execute. The go to operator is created with an equal sign and a greater than symbol, like: =>.
- The mapping of the variables comes after the go to operator. The mapping can be any M Code that delivers an M Code value. The mapping almost always uses the defined variables, but it does not have to use the variables.
- After the variable name, you can define value types for variables by typing the keyword *as*, followed by the identifier of the value type. Value types are optional.
- After the close parentheses for the variable names and before the go to operator, you can define value types for the output of the function by typing the keyword *as*, followed by the identifier of the value type. Value types are optional.
- As with all M Code, you can add non-executable comments to your custom function by beginning a single-line comment with two forward slashes, like //, or start your multi-line (multiple hard returns) comments with /* and end with */.

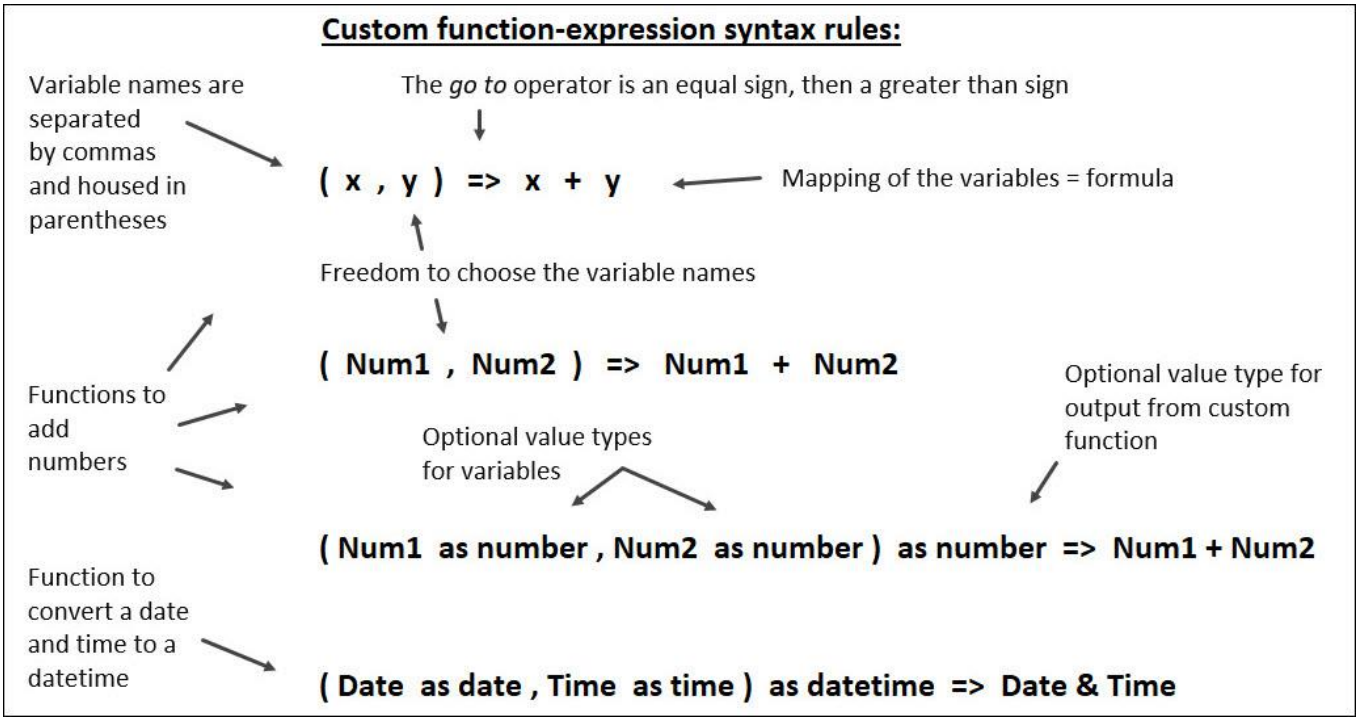


Figure-41: You can create your own custom functions.

Custom functions can be created in any M Code, but are most often created in three places:

- A new query that uses the let expression to deliver a reusable custom function value, called a **re-usable function query**, or just **function query**. The function query is available throughout the Power Query Editor in the file or Dataflow area.
- A function argument that requires a function. The custom function is available only in the function argument.
- As an intermediate step in a let expression, called a **function query step**. The function query step is available throughout the let expression, but the let expression itself does not necessarily have to deliver a custom function value.

Each and underscore to create custom functions

When we used the Table.AddColumn function, we saw that in a function argument that requires a function, like the third argument of the Table.AddColumn function, the *each* keyword allows a formula to calculate in each row of a table. The *each* keyword also allows formulas to calculate in each row of a list value. In a function argument that requires a function, if a formula only accesses the columns from the table that is iterated, you do not need to go through the extra effort to define variables and use the go to operator. All you need to do is type *each*, followed by a formula that pulls values from each row using the field access operator (square brackets). When you use the keyword *each* the formula becomes a custom function. In addition, as we have seen so far, the underscore character (_) works with the *each* keyword to extract a row as a record or a grouped set of records as a table in each row of a table, or to extract an item from each row in a list. Together, the each keyword and the underscore character are the syntax that Microsoft created to make it easy to create custom functions. Here is a bullet list of my full definition of the **keyword *each*** and the **underscore character (_)**:

- The *each* keyword is syntactical shorthand for defining an unnamed custom function taking a single untyped variable named underscore (_).
- You can use the each shorthand anywhere a function can be declared.
- The *each* keyword is often used to pass a custom function to an argument in other functions, such as Table.AddColumn, Table.SelectRows, or List.Transform.
- The *each* keyword can be thought of as "allowing you to make a calculation in each row of a table or list”.
- The underscore character _ can be thought of as extracting everything from a row in a table or list.
- These are examples of equivalent custom functions that are invoked in a custom column that each use a different syntax:

```
each [Years] * [PeriodsPerYear]
each _[Years] * _[PeriodsPerYear]
(_) => _[Years] * _[PeriodsPerYear]
(Row) => Row[Years] * Row[PeriodsPerYear]
```

- To extract a row as a record in each row of a table, or extract the items in each row of a list, these are equivalent custom functions that are invoked in a custom column that each use a different syntax:

```
each _
(_) => _
(Row) => Row
```

The following bullet communicates the most succinct definition of the *each* keyword and underscore character.

- The *each* keyword is syntactical shorthand for defining an unnamed custom function taking a single untyped variable named underscore (_).

To better understand this definition, let's start by considering this custom function:

```
each [Years] * [PeriodsPerYear]
```

The "unnamed custom function" part of the definition means that we did not use an identifier when we wrote the formula. We just used the *each* keyword and the formula. Because there is no identifier, we can't refer to this custom function anywhere else. This custom function only works in this column.

The "taking a single untyped variable named underscore _", part of the definition means that we did not have to type an underscore as a preface to each column reference (column name in the field access operator). However, you could type the underscore character as a preface to each column reference, as shown in the below formula, and the formula works. If you use the *each* keyword for a formula that uses column references, you almost never type the underscore character. That is why the definition uses the adjective "untyped".

As shown below, you could create a custom function with the underscore and field access operators. What is happening with the formula is as follows: first the underscore character extracts the full row, and then the field access operator extracts the value from the specified column for the given record.

```
each _[Years] * _[PeriodsPerYear]
```

As shown below, you could create a custom function with the single defined variable underscore (_) and the go to operator. This syntactic structure is exactly what the each keyword represents: a custom function taking a single variable named underscore _ .

```
(_) => _[Years] * _[PeriodsPerYear]
```

As shown below, you don't have to use underscore as the variable to represent the row, you can use any variable name that you would like. In the below formula, by choosing the variable name "Row", I am explicitly naming the variable for what it does: extract the row as a record in each row of the table.

```
(Row) => Row[Years] * Row[PeriodsPerYear]
```

If you wanted to extract a row as a record in each row of a table, or extract the items in each row of a list, these are equivalent custom functions that each use a different syntax:

```
each _
(_) => _
(Row) => Row
```

If you would like to create the formulas as shown in the above bullets, you can select the query with the name CustomFunctionQ, and create each one in a new custom column. You will have to edit the formulas in the Formula Bar after you click OK in the Custom Column dialog box to get them to match. Figure-42 shows an example of how I did them.

```
    Each = Table.AddColumn(FormulaCF, "eachTotPeriods", each [Years] * [PeriodsPerYear], type number),
    Each_ = Table.AddColumn(Each, "eachUnderscoreTP", each _[Years] * _[PeriodsPerYear], type number),
    CF_ = Table.AddColumn(Each_, "VariableUnderscore", (_) => _[Years] * _[PeriodsPerYear], type number),
    CF = Table.AddColumn(CF_, "VariablesGoTo", (Row) => Row[Years] * Row[PeriodsPerYear], type number),
    RowEach_ = Table.AddColumn(CF, "RowEach_", each _),
    RowCF_ = Table.AddColumn(RowEach_, "RowCF_", (_) => _),
    RowCF = Table.AddColumn(RowCF_, "RowCF", (Row) => Row)
in
    RowCF
```

Figure-42: Seven formulas to help illustrate how the each and underscore _ work.

If expression

The **if expression** is used when you want the expression to deliver one of two items based on a logical test. The syntax for the if expression is as follows:

```
if logical_test then expression_if_true else expression_if_false
```

Table.AddColumn function

When you create a formula in a Custom Column dialog box, behind the scenes the formula is placed into the Table.AddColumn function. The **Table.AddColumn** function adds a column to a table by running a formula in each row. The arguments for the function are as follows:

```
Table.AddColumn(
    table as table,
    newColumnName as text,
    columnGenerator as function,
    optional columnType as nullable type) as table
```

Excel.CurrentWorkbook function

The **Excel.CurrentWorkbook()** is an argumentless function that extracts objects from the current Excel file (file where the function is being used) and delivers a table of objects and object names to the Power Query Editor The returned table has two columns: one for the object and for the name of the object. The primary key of the returned table is the Name column. The objects it can return are as follows:

- Excel Tables.
- User created defined names (LAMBDA and Formula names are not imported).
- Automatically created defined names when you use these features:
  - Print range.
  - Advanced Filter criteria and extract ranges.
  - Filter feature.
  - Dynamic arrays.

This function does not import worksheets into the Power Query Editor. The Excel.Workbook can import worksheets, the Excel.CurrentWorkbook function cannot. In addition, if you have dynamic spilled arrays in your Excel file and you have never manually imported them into the Power Query Editor, the Excel.CurrentWorkbook function will not detect or import the dynamic arrays. However, when you manually import a dynamic array into the Power Query Editor, an automatic defined name is created. It is only after the automatic defined name is created, that the Excel.CurrentWorkbook can "see" the dynamic array and therefore import it as an object from the current Excel file into the Power Query Editor.

> **Note**: The Excel.Workbook function imports worksheets, but not dynamic arrays. The Excel.CurrentWorkbook function imports dynamic arrays, but not worksheets.

Finally, if you are using the Excel.CurrentWorkbook function to import multiple Excel Tables, append them into a single table, and then load the table back to the worksheet, you must prevent recursion from occurring. **Recursion** is when a function can call itself and in the case of the Excel.CurrentWorkbook function, because it delivers an Excel Table to the worksheet, and the function is programmed to import all Excel Tables, when you refresh the query, the function imports the original Tables plus the query output Table (importing itself), thereby calling itself and doubling the size of the table. There are a few simple solutions. The first solution is to add a single line of M Code that filters out the name of the query. In this way, when you refresh, the query output table is not allowed to be part of the final output. Another solution is to not load the append table to the worksheet, but instead load it to a PivotTable cache.

Csv.Document function

The **Csv.Document** function converts a text file that contains a specified delimiter into a table. By default, the function assumes a comma delimiter that is most often used in a csv file. The function extracts the data from the text file and converts it to a table. The arguments for the function are shown here:

```
Csv.Document(
    source as any,
    optional columns as any,
    optional delimiter as any,
    optional extraValues as nullable number,
    optional encoding as nullable number) as table
```

The **source** argument can accept text files such as csv or txt text files. The **columns** argument allows you to specify which columns to import and will accept null (all columns), the number of columns, a list of column names, a table type, or an options record. By default, all columns are imported. This argument can be helpful if you want to specify, not all columns, but just certain ones: this allows you to avoid later query steps to remove columns. If you enter 2 and there are four columns, the two columns from the left are imported. You can also include a list of the names of columns to import, which allows you to pick and choose which columns you want to import, regardless of the order of the columns. If the number of columns in the source Csv file might change, omit the columns argument, or use a *null* value to skip it. This will allow the Csv.Document to infer the number of columns each time the file is refreshed. We will look at the record and table options for the columns argument later. The **delimiter** argument can accept almost any set of characters, but most csv and txt files use a comma or a tab character as the delimiter. In the delimiter argument, you can enter a single delimiter, a list of delimiters or a list of fixed widths. Special delimiter characters require specific M Code such as:

- Carriage-return = "#(cr)"
- Linefeed = "#(lf)"
- Tab = "#(tab)"
- Consecutive white spaces = ""

The **extraValues** argument works with the Columns argument to determine what happens to columns that are skipped. Here are the options that you can enter into this argument:

- ExtraValues.List = 0 = If the splitter function returns more columns than the table expects, they should be collected into a list.
- ExtraValues.Error = 1 = If the splitter function returns more columns than the table expects, an error should be raised. This is the default.
- ExtraValues.Ignore = 2 = If the splitter function returns more columns than the table expects, they should be ignored.

If you remove columns using the columns arguments, you probably want to use ExtraValues.Ignore, or the equivalent 2, so that you don't get an error.

The **encoding** argument specifies the text encoding type. For example, when you use Save As to save a csv file, you can see in the coding for some file types in the Save as type text box. For

example the option I use is: CSV UTF-8 (Comma delimited) (*csv) informs me that the coding is UTF-8. Here is a list of some of the many encodings:

- 65000, UTF-7, Unicode (UTF-7)
- 65001, UTF-8, Unicode (UTF-8). **This is the default**.
- 10001, X-Mac-Japanese, Japanese (Mac)
- 1252, Windows-1252, ANSI Latin 1; Western European (Windows)
- 865, IBM865, OEM Nordic; Nordic (DOS)
- Full list here: https://learn.microsoft.com/en-us/windows/win32/intl/code-page-identifiers?redirectedfrom=MSDN

```
= Csv.Document(File.Contents("E:\MCodeExcelisfunBook\FruitSales.csv"),
    [Delimiter=",", Columns=4, Encoding=65001, QuoteStyle=QuoteStyle.None])
```

	Column1	Column2	Column3	Column4
1	Date	Fruit	Sales	Customer
2	1/2/2025	Apple	500	Jun
3	1/3/2025	Orange	600	Iggy
4	1/2/2025	Apple	450	Lim
5	1/4/2025	Orange	375	Ty

Figure-43: A record with parameters can be specified in the columns argument.

As shown in Figure-43 above, you can provide a record in the columns argument that defines the five parameters: Delimiter, Columns, Encoding, QuoteStyle and CsvStyle. You can provide one to five parameters in a record (the above picture uses four of the parameters). If you use the user interface to import text files, this record is automatically generated. The description of each of the five parameters is listed below:

- **Delimiter**. You can set a delimiter in record like: Delimiter="#(tab)" for tab.
- **Columns**. You can set the number of columns to import like: Columns=3. If you anticipate that the number of columns may change, omit the columns parameter to accommodate a varying number of columns.
- **Encoding.** You can set the text encoding type, like: Encoding=65001.
- **QuoteStyle**. You can specify how quoted line breaks are handled with one of two options:
  - **QuoteStyle.None** or **0** means quoted line breaks are ignored. This is the default. An example is shown in Figure-44.
  - **QuoteStyle.Csv** = **1** = means that Apply all line breaks. An example is shown in Figure Ch06-04.

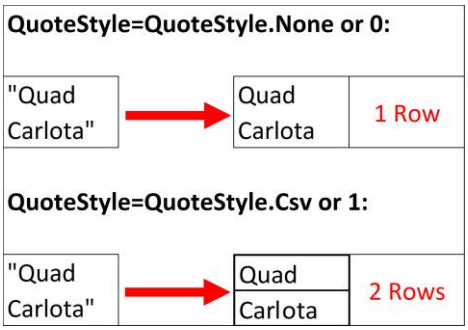


Figure-44: The QuoteStyle argument defines how line breaks are interpreted.

- **CsvStyle**. You can specify how quotes are handled with one of two options:
  - **CsvStyle.QuoteAfterDelimiter** or **0** means that quotes in a field are only significant immediately following the delimiter. This is the default. If you import a record with 43 and "Rad", Figure-45 shows that if you use this option, that the quotes around the word Rad are included in column 2.
  - **CsvStyle.QuoteAlways** or **1** means that quotes in a field are always significant, regardless of where they appear. If you import a record with 43 and "Rad", Figure-46 shows that if you use this option, that the quotes around the word Rad are not included in column 2.

```
= Csv.Document(File.Contents("E:\MCodeExcelisfunBook\CsvStyle.csv"),
    [CsvStyle=CsvStyle.QuoteAfterDelimiter])
```

	Column1	Column2
1	43	"Rad"

Figure-45: The option CsvStyle.QuoteAfterDelimiter allows you to import quotes.

```
= Csv.Document(File.Contents("E:\MCodeExcelisfunBook\CsvStyle.csv"),
    [CsvStyle=CsvStyle.QuoteAlways])
```

	Column1	Column2
1	43	Rad

Figure-46: CsvStyle.QuoteAlways allows you to ignore quotes.

As shown in Figure-47, you can provide a table in the columns argument that defines the column names and data types for each column. However, I have not found a good use for this because although data types are defined at the top of each column, the data is still all text. In Figure-47, the sales numbers are aligned left and are text values. If you try to add the numbers, you will get an error.

```
= Csv.Document(File.Contents
    ("E:\MCodeExcelisfunBook\FruitSalesNoColumnNames.csv"),
     type table [Date=date, Fruit=text, Sales=number])
```

	Date	Fruit	Sales
1	1/2/2025	Apple	500
2	1/1/2025	Orange	600
3	1/2/2025	Apple	450
4	1/2/2025	Orange	375

Figure-47: A table in the columns argument of Csv.Document defines column names.

File.Contents function

The **File.Contents** function gets the contents of the file and is used inside many data connector functions, such as: Csv.Document, Excel.Workbook, Json.Document and Xml.Tables. Inside the File.Contents functions is the hard coded file path that is unique to your computer and is called an on-premises file path, as shown in Figure-48 below.

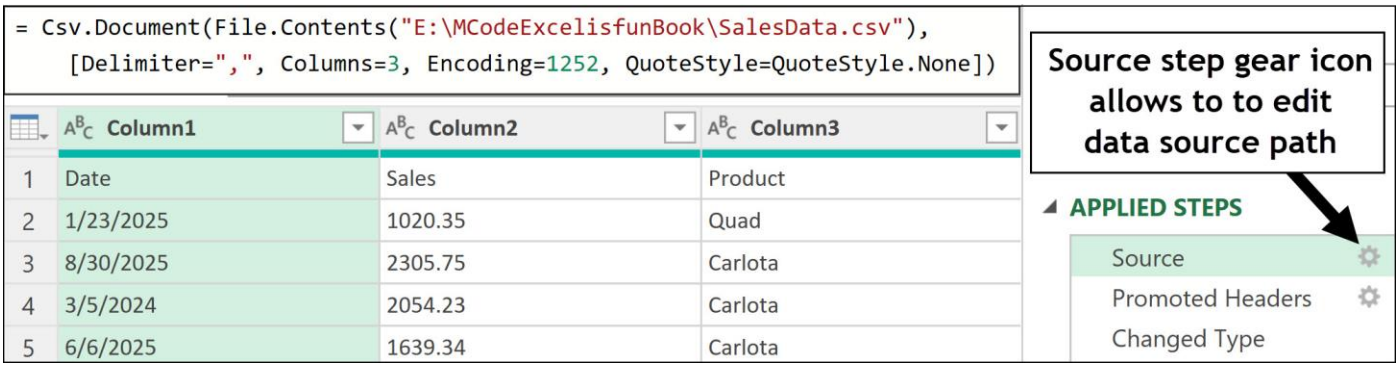


Figure-48: A record with parameters is created in the columns argument.

On-premises file and folder paths

As shown back in Figure-48, in the Source query step, in the first argument of the File.Content function, the file path for the SalesData.csv file is hard coded into the formula. When you hard coded a file path into a query, it means that if you move that source data file to a different location, or you e-mail the file that contains the query to a colleague, the connection to the source data is broken and you will receive the following error: **Data Source Not Found**. When you hard code a file or folder path into a query, the path is called an **on-premises path**. As you can imagine, on-premises paths cause a lot of trouble if you are sharing files or moving files around. An on-line data source, like an SQL server database, Dataflow or Power BI do not have this problem because the data is stored online in a location that does not move. Multiple credentialed people can have access to "a single source of truth" where there are no on-premises paths and no conflicts with multiple versions of the same file. Nevertheless, not all data is stored online, and on-premises paths are common. The good news is that if you know where the source data file is, it is easy to redirect the query to the new location. There are at least three ways to change the on-premises path:

- In the Source step for almost any query, you can edit the on-premises path in the formula bar.
- As shown back in Figure-48, click the gear icon in the Source query step to open the source data dialog box. Many data sources such as Csv, Excel, Sql Databases, Web sites and more allow you to use the gear icon in the source query step to edit the connection details. As shown in Figure-49, the source data dialog box for the SalesDataCsv Source query step allows you to edit the parameters for the csv file import.
- If you have used the same file or folder in multiple queries, it is most efficient to edit the path universally in the Data Source Settings dialog box. There are multiple ways to open this dialog box in Excel and Power BI. If you are in the Power Query Editor:
    - In the Excel Power Query Editor, in the Home tab, Data Sources group, click the Data source settings button.
    - In the Power BI Desktop Power Query Editor, in the Home tab, Data source group, click the Transform data dropdown and then click Data source settings.
    - In the Dataflow Power Query Editor, in the Home tab, Data sources group, click the Manage connections button.

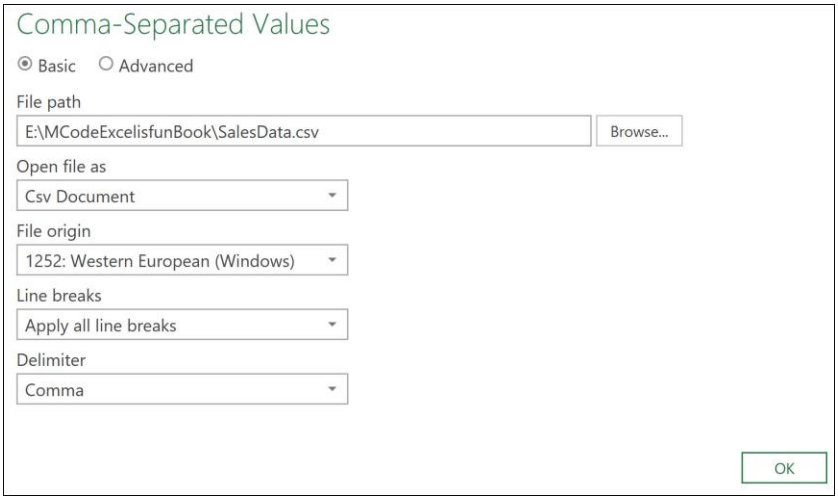


Figure-49: The source data dialog box for the Sales.csv file.

Data source settings

Figures-50 and Figures-51 show the Data source settings dialog box that is the same in both Excel and Power BI Desktop. The dialog box in Dataflow is different because it only contains online sources. Figures-50 shows Data sources in current workbook (or Power BI file) and Figures-51 shows global data sources that have been used in other files. The Change Source button allows you to change the path for the selected the on-premises file. The Edit Permissions and Clear Permissions allow you to edit and clear credentials for online data sources or other sources that require credentials. When you use the same file in many queries, it is convenient to have one universal location to edit and change the file or folder path. In addition, if your credentials for an online source have changed, or you want to clear a permission, these dialog boxes are convenient. Next, we want to create a dynamic on-premises folder or file path that will automatically update when a file is moved.

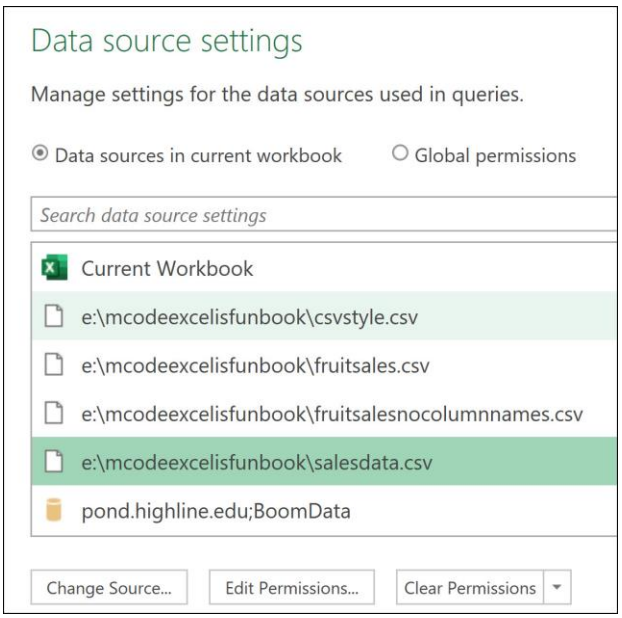
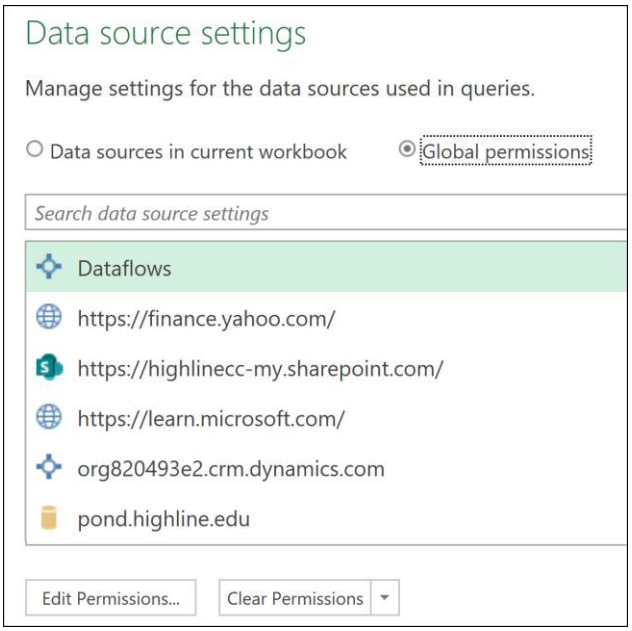


Figure-50: Data source settings for sources in current workbook (or Power BI file).



Figures-51: Data source settings Global permissions.

Approximate match lookup

Although M Code has built-in methods to create exact match lookup formulas, there is no built-in method for approximate match lookup. However, with our knowledge of how to build custom functions, we can build a function to complete an approximate match lookup. As we first learned at the beginning of this chapter, the definition of an approximate match lookup is as follows:

Figure Ch04-52 shows the lookup table that we will use for our examples.

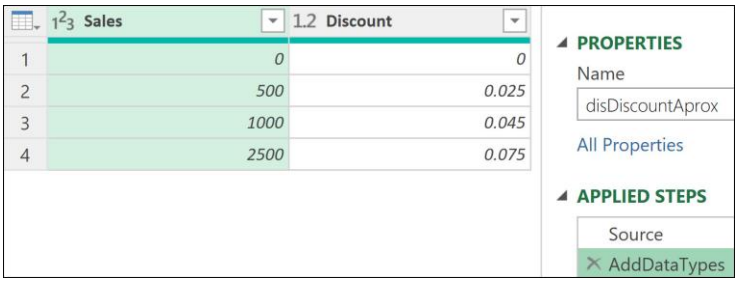


Figure-52: The disDiscountAprox query is the sales discount lookup table.

The categories for the disDiscountAprox query lookup table are shown in Figure-53.

Sales	Discount	Categories
0	0	0 >= Sales < 500
500	0.025	500 >= Sales < 1,000
1000	0.045	1,000 >= Sales < 2,500
2500	0.075	2,500 >= Sales

Figure-53: Categories for the sales discount lookup table.

Table.Buffer function

The **Table.Buffer** function takes a table and stores it in memory. Then each time the table is called, it does have to go back to the query to get the table, instead, it just uses the table that is stored in memory. This helps to reduce the time that it takes the formula to namek the calculation.

Approximate Match Discount Lookup Formulas used in video show in next 3 figures:

ApproximateMatchLookup

```
//ApproximateMatchLookup
//Approximate match function with inputs: LookupTable and Discount table with Sales field and Discount field

(LookupTable as table, LookupValue as number) as number =>

let
    LookupColumn = LookupValue
in
    List.Last( Table.SelectRows(LookupTable,each [Sales] <= LookupColumn)[Discount])
```

Figure-54: Uses the let expression to define LookupColumn

ApproxMatchHardCodeTable

```
//ApproxMatchHardCodeTable
// Aproximate match lookup with hard coded discount table

(LookupValue as number) as number =>

let
    LookupTable =
        #table( type table [Sales = number, Discount = number],
        {{0,0},{500,0.025},{1000,0.045},{2500,0.075}}),

    LookupColumn = LookupValue
in
    List.Last( Table.SelectRows(LookupTable,each [Sales] <= LookupColumn)[Discount])
```

Figure-55: You can hard code lookup table into function and avoid buffering the table

ApproximateMatchLookup

```
//ApproximateMatchLookup
//Approximate match function with inputs: LookupTable and Discount table with Sales field and Discount field

(LookupTable as table, LookupValue as number) as number =>

let
    LookupColumn = LookupValue
in
    List.Last( Table.SelectRows(LookupTable,each [Sales] <= LookupColumn)[Discount])
```

Figure-56: In the comments, Geert Delmulle said we don't need the let expression!

Table.Group function

The Table.Group function has the following five arguments:

```
Table.Group(
    table as table,
    key as any,
    aggregatedColumns as list,
    optional groupKind as nullable number,
    optional comparer as nullable function) as table
```

The **table** argument contains the table where the group by calculation are made.

The **key** argument can contain a single text column name or a list of column names for the columns that determine the unique set of row criteria.

The **aggregatedColumns** argument contains a list within a list, where each sub-list details the three components for each aggregate calculation: column name, function and data type.

The optional **groupKind** argument allows you to choose the type of grouping as shown below in Figure-57:

Name	Value	Description
GroupKind.Local	0	A local group is formed from a consecutive sequence of rows from an input table with the same key value.
GroupKind.Global	1	A global group is formed from all rows in an input table with the same key value.

Figure-57 Options for the groupKind argument in Table.Group.

The optional **comparer** argument allows you to test equality and determine inclusion for the grouping action.

Figure-58 and 59 shows the baseball example we did in the video:

	Date	Team	Opponent	Result	As Runs	O Runs
1	4/8/2022	OAK	PHI	Win	10	9
2	4/9/2022	OAK	PHI	Win	8	4
3	4/10/2022	OAK	PHI	Win	4	1
4	4/11/2022	OAK	TBR	Win	13	2
5	4/12/2022	OAK	TBR	Win	8	2
6	4/13/2022	OAK	TBR	Win	4	2
7	4/14/2022	OAK	TBR	Win	6	3
8	4/15/2022	OAK	TOR	Loss	1	4
9	4/16/2022	OAK	TOR	Win	7	5
10	4/17/2022	OAK	TOR	Loss	3	4
11	4/18/2022	OAK	BAL	Win	5	1
12	4/19/2022	OAK	BAL	Win	2	1
13	4/20/2022	OAK	BAL	Win	2	1
14	4/21/2022	OAK	BAL	Win	6	4
15	4/22/2022	OAK	TEX	Win	1	0
16	4/23/2022	OAK	TEX	Loss	0	2
17	4/24/2022	OAK	TEX	Loss	2	3

Figure-58: A's Baseball Team wins and losses data.

```
= Table.Group(AddDataTypes, {"Result"},
{{"Win or Loss Strek", each Table.RowCount(_), Int64.Type},
{"Dates", each Text.From(List.Min(_[Date]))&" to "&Text.From(List.Max(_[Date]))}},0)
```

	Result	Win or Loss Strek	Dates
1	Win	7	4/8/2022 to 4/14/2022
2	Loss	1	4/15/2022 to 4/15/2022
3	Win	1	4/16/2022 to 4/16/2022
4	Loss	1	4/17/2022 to 4/17/2022
5	Win	5	4/18/2022 to 4/22/2022

Figure-59 GroupKind.Local.

Table.Group fifth argument: Comparer

The **comparer** argument in the Table.Group function allows you to test equality and determine grouping categories with the comparer functions as shown in Figure Ch05-60.

Name	Description
Comparer.Equals	Returns a logical value based on the equality check over the two given values.
Comparer.FromCulture	Returns a comparer function based on the specified culture (local in Regional settings) and case-sensitivity.
Comparer.Ordinal	Returns a comparer function which uses ordinal Unicode characters to compare values.
Comparer.OrdinalIgnoreCase	Returns a case-insensitive comparer function which uses ordinal Unicode characters to compare values.
Custom function	Build your own two variable function to compare values. **Example: group by rows where date is not a null value:** = Table.Group(AddDataTypes, {"Date"}, {{"Total Sales", each List.Sum([Amount]), type number}}, 0, (InitialValue,CurrentRow) => Number.From(CurrentRow[Date]<>null))
Some of the functions that have arguments that can use these comparer functions are: List.Contains, List.ContainsAll, List.ContainsAny, List.Difference, List.Distinct, List.Intersect, List.IsDistinct, List.Max, List.MaxN, List.Min, List.MinN, List.Mode, List.Modes, List.PositionOf, List.PositionOfAny, List.RemoveMatchingItems, List.ReplaceMatchingItems, List.Sort, List.Union, Table.Contains, Table.ContainsAll, Table.ContainsAny, Table.Distinct, Table.Group, Table.PositionOf, Table.PositionOfAny, Table.RemoveMatchingRows, Table.ReplaceMatchingRows, Text.Contains, Text.EndsWith, Text.PositionOf, Text.StartsWith...	

Figure Ch05-60: Comparer functions used in Table.Group, List.Distinct, and other functions.

Comparer functions and where to use them

Comparer functions allow you to test equality to determine order, inclusion, grouping and other tasks. As shown in the above figure, there are four built-in functions plus the ability to build your own custom function for comparing. You can use these functions in the arguments of other functions such as List.Sort, List.Distinct and Table.Group.

Microsoft's documentation states that the **custom function option** require two arguments, one for the **seed** value (value to start the iteration over a list of values, and one for the **row** values (list of values to iterate). The possible values for the custom function can have a pattern like these patterns:

```
If x > y then 1 else if x < y then -1 else 0
If x = y then 1 else 0
If x <> y then 1 else 0
If x > y then 1 else 0
If x < y then 1 else 0
```

If you build a custom function that delivers logical values, you must convert the logical values to numbers using functions such as Number.From.

An example of the comparer functions is shown in Figure-61 and 62:

	SalesRep	Sales
1	Sioux Rad	45
2	chantel Mims	98
3	Juniper Snap	59
4	sioux rad	39
5	Chantel mims	62
6	juniper snap	73

Figure-61: This is the data set that we started with and the cases of letters need to be ignored.

```
= Table.Group(AddDataTypes, {"SalesRep"},
    {{"Total Sales", each List.Sum([Sales]), type nullable number}},
    null,
    Comparer.OrdinalIgnoreCase)
```

	SalesRep	Total Sales
1	Sioux Rad	128
2	Chantel Mims	223
3	Juniper Snap	132

Figure-62: This groups and ignores case

An example of using a custom function in the fifth argumnet is shown in Figure 63 and 64:

	SalesRep
1	Sioux
2	25
3	41
4	6
5	Chantel
6	35
7	78
8	78

Figure-63: Records in a single column.

```
= Table.Group(Source, {"SalesRep"},
    {{"Total Sales", each List.Sum(Table.Skip(_)[SalesRep])}},
    0,
    (Seed,Row) => Number.From(Row[SalesRep] is text))
```

	SalesRep	Total Sales
1	Sioux	72
2	Chantel	269
3	Juniper	132

Figure-64: Custom Function in fifth argument of Table.Group.