

Power Query M formula language

Microsoft Power Query provides a powerful data import experience that encompasses many features. Power Query works with desktop Analysis Services, Excel, and Power BI workbooks, in addition to many online services, such as Fabric, Power BI service, Power Apps, Microsoft 365 Customer Insights, and more. A core capability of Power Query is to filter and combine, that is, to mash-up data from one or more of a rich collection of supported data sources. Any such data mashup is expressed using the Power Query M formula language. The M language is a functional, case sensitive language similar to F#.

Get started

QUICKSTART

[M quick tour](#)

CONCEPT

[Expressions, values, and let expression](#)

[Evaluation model](#)

[Types and type conversion](#)

[How culture affects text formatting](#)

TRAINING

[Power Query documentation](#)

[Understanding Power Query M functions](#)

Specification

ARCHITECTURE

[Power Query M language specification](#)

[Power Query M type system](#)

Reference

REFERENCE

[Power Query M function reference](#)

[Power Query M enumeration reference](#)

[Power Query M constant reference](#)

[Power Query M dynamic value reference](#)

Quick tour of the Power Query M formula language

Article • 01/27/2025

This quick tour describes creating Power Query M formula language queries.

ⓘ Note

M is a case-sensitive language.

Create a query with the Power Query editor

To create an advanced query, you use the [Power Query advanced editor](#). A mashup query is composed of variables, expressions, and values encapsulated by a `let` expression. A variable can contain spaces by using the `#` identifier with the name in quotes as in `#"Variable name"`.

A `let` expression follows this structure:

Power Query M

```
let
    Variablename = expression,
    #"Variable name" = expression2
in
    Variablename
```

To create an M query in the advanced editor, you follow this basic process:

1. Create a series of query formula steps that start with the `let` statement. Each step is defined by a step variable name. An M *variable* can include spaces by using the `#` character, such as `#"Step Name"`. A formula step can be a custom formula. Also note that the Power Query formula language is case sensitive.
2. Each query formula step builds upon a previous step by referring to a step by its variable name.
3. Output a query formula step using the `in` statement. Generally, the last query step is used as the `in` final data set result.

To learn more about expressions and values, go to [Expressions, values, and let expression](#).

Simple Power Query M formula steps

Let's assume you created the following transform in the Power Query editor. This query converts product names to the appropriate case, in this instance, to all initial capitalization.

Power Query

Home

Transform

Add column

View

Help

Get data

Enter data

New query

Manage connections

Data sources

Options

Options

Manage parameters

Parameters

Refresh

Query

Properties

Advanced editor

Manage

>

Queries [1]

Table.TransformColumns(Orders, {"Item", Text.Proper})

	OrderID	CustomerID	Item	Price
1	1	1	Fishing Rod	100
2	2	1	1 Lb. Worms	5
3	3	2	Fishing Net	25

To begin with, you have a table that looks like this:

[Expand table](#)

OrderID	CustomerID	Item	Price
1	1	fishing rod	100
2	1	1 lb. worms	5
3	2	fishing net	25

And, you want to capitalize the first letter in each word in the Item column to produce the following table:

[Expand table](#)

OrderID	CustomerID	Item	Price
1	1	Fishing Rod	100
2	1	1 Lb. Worms	5
3	2	Fishing Net	25

The M formula steps to project the original table into the results table look like this in the Power Query advanced editor:

Advanced editor

```

1 let Orders = Table.FromRecords({
2   [OrderID = 1, CustomerID = 1, Item = "fishing rod", Price = 100.0],
3   [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
4   [OrderID = 3, CustomerID = 2, Item = "fishing net", Price = 25.0]}),
5   #"Capitalized Each Word" = Table.TransformColumns(Orders, {"Item", Text.Proper})
6 in
7   #"Capitalized Each Word"

```

OK

Cancel

Here's the code you can paste into the Power Query advanced editor:

Power Query M

```

let Orders = Table.FromRecords({
    [OrderID = 1, CustomerID = 1, Item = "fishing rod", Price = 100.0],
    [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
    [OrderID = 3, CustomerID = 2, Item = "fishing net", Price = 25.0]}),
    #"Capitalized Each Word" = Table.TransformColumns(Orders, {"Item",
Text.Proper})
in
    #"Capitalized Each Word"

```

Let's review each formula step.

1. **Orders:** Create a table with data for Orders.
2. **#"Capitalized Each Word":** To capitalize each word, you use [Table.TransformColumns](#).

3. in #"Capitalized Each Word": Output the table with the first letter of each word capitalized.

Related content

- [Expressions, values, and let expression](#)
- [Operators](#)
- [Type conversion](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Power Query M language specification

Article • 11/14/2024

The specification describes the values, expressions, environments and variables, identifiers, and the evaluation model that form the Power Query M language's basic concepts.

The specification is contained in the following topics.

- [Introduction](#)
- [Lexical Structure](#)
- [Basic Concepts](#)
- [Values](#)
- [Types](#)
- [Operators](#)
- [Let](#)
- [Conditionals](#)
- [Functions](#)
- [Error Handling](#)
- [Sections](#)
- [Consolidated Grammar](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Introduction

Article • 07/21/2023

Overview

Microsoft Power Query provides a powerful "get data" experience that encompasses many features. A core capability of Power Query is to filter and combine, that is, to "mash-up" data from one or more of a rich collection of supported data sources. Any such data mashup is expressed using the Power Query formula language (informally known as "M"). Power Query embeds M documents in a wide range of Microsoft products, including Excel, Power BI, Analysis Services, and Dataverse, to enable repeatable mashup of data.

This document provides the specification for M. After a brief introduction that aims at building some first intuition and familiarity with the language, the document covers the language precisely in several progressive steps:

1. The *lexical structure* defines the set of texts that are lexically valid.
2. Values, expressions, environments and variables, identifiers, and the evaluation model form the language's *basic concepts*.
3. The detailed specification of *values*, both primitive and structured, defines the target domain of the language.
4. Values have *types*, themselves a special kind of value, that both characterize the fundamental kinds of values and carry additional metadata that is specific to the shapes of structured values.
5. The set of *operators* in M defines what kinds of expressions can be formed.
6. *Functions*, another kind of special values, provide the foundation for a rich standard library for M and allow for the addition of new abstractions.
7. *Errors* can occur when applying operators or functions during expression evaluation. While errors aren't values, there are ways to *handle errors* that map errors back to values.
8. *Let expressions* allow for the introduction of auxiliary definitions used to build up complex expressions in smaller steps.
9. *If expressions* support conditional evaluation.

10. *Sections* provide a simple modularity mechanism. (Sections aren't yet leveraged by Power Query.)
11. Finally, a *consolidated grammar* collects the grammar fragments from all other sections of this document into a single complete definition.

For computer language theorists: the formula language specified in this document is a mostly pure, higher-order, dynamically typed, partially lazy functional language.

Expressions and values

The central construct in M is the *expression*. An expression can be evaluated (computed), yielding a single *value*.

Although many values can be written literally as an expression, a value isn't an expression. For example, the expression `1` evaluates to the value `1`; the expressions `1+1` evaluates to the value `2`. This distinction is subtle, but important. Expressions are recipes for evaluation; values are the results of evaluation.

The following examples illustrate the different kinds of values available in M. As a convention, a value is written using the literal form in which they would appear in an expression that evaluates to just that value. (Note that the `//` indicates the start of a comment which continues to the end of the line.)

- A *primitive* value is single-part value, such as a number, logical, text, or null. A null value can be used to indicate the absence of any data.

Power Query M	
<code>123</code>	<code>// A number</code>
<code>true</code>	<code>// A logical</code>
<code>"abc"</code>	<code>// A text</code>
<code>null</code>	<code>// null value</code>

- A *list* value is an ordered sequence of values. M supports infinite lists, but if written as a literal, lists have a fixed length. The curly brace characters `{` and `}` denote the beginning and end of a list.

Power Query M	
<code>{123, true, "A"}</code>	<code>// list containing a number, a logical, and // a text</code>
<code>{1, 2, 3}</code>	<code>// list of three numbers</code>

- A *record* is a set of *fields*. A field is a name/value pair where the name is a text value that's unique within the field's record. The literal syntax for record values allows the names to be written without quotes, a form also referred to as *identifiers*. The following shows a record containing three fields named "A", "B", and "C", which have values 1, 2, and 3.

```
Power Query M

[
    A = 1,
    B = 2,
    C = 3
]
```

- A *table* is a set of values organized into columns (which are identified by name), and rows. There's no literal syntax for creating a table, but there are several standard functions that can be used to create tables from lists or records.

For example:

```
Power Query M

#table( {"A", "B"}, { {1, 2}, {3, 4} } )
```

This creates a table of the following shape:

A	B
1	2
3	4

- A *function* is a value that, when invoked with arguments, produces a new value. A function is written by listing the function's *parameters* in parentheses, followed by the goes-to symbol `=>`, followed by the expression defining the function. That expression typically refers to the parameters (by name).

```
Power Query M

(x, y) => (x + y) / 2`
```

Evaluation

The evaluation model of the M language is modeled after the evaluation model commonly found in spreadsheets, where the order of calculation can be determined based on dependencies between the formulas in the cells.

If you've written formulas in a spreadsheet such as Excel, you may recognize the formulas on the left result in the values on the right when calculated:

	A		A
1	=A2 * 2	1	4
2	=A3 + 1	2	2
3	1	3	1

In M, parts of an expression can reference other parts of the expression by name, and the evaluation process automatically determines the order in which referenced expressions are calculated.

You can use a record to produce an expression that's equivalent to the previous spreadsheet example. When initializing the value of a field, you can refer to other fields within the record by using the name of the field, as follows:

Power Query M

```
[  
    A1 = A2 * 2,  
    A2 = A3 + 1,  
    A3 = 1  
]
```

The above expression is equivalent to the following (in that both evaluate to equal values):

Power Query M

```
[  
    A1 = 4,  
    A2 = 2,  
    A3 = 1  
]
```

Records can be contained within, or *nest*, within other records. You can use the *lookup operator* (`[ ]`) to access the fields of a record by name. For example, the following record

has a field named `Sales` containing a record, and a field named `Total` that accesses the `FirstHalf` and `SecondHalf` fields of the `Sales` record:

Power Query M

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = Sales[FirstHalf] + Sales[SecondHalf]
]
```

The above expression is equivalent to the following when it is evaluated:

Power Query M

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = 2100
]
```

Records can also be contained within lists. You can use the *positional index operator* (`{}`) to access an item in a list by its numeric index. The values within a list are referred to using a zero-based index from the beginning of the list. For example, the indexes `0` and `1` are used to reference the first and second items in the list below:

Power Query M

```
[
    Sales =
        {
            [
                Year = 2007,
                FirstHalf = 1000,
                SecondHalf = 1100,
                Total = FirstHalf + SecondHalf // 2100
            ],
            [
                Year = 2008,
                FirstHalf = 1200,
                SecondHalf = 1300,
                Total = FirstHalf + SecondHalf // 2500
            ]
        },
    TotalSales = Sales{0}[Total] + Sales{1}[Total] // 4600
]
```

List and record member expressions (as well as [let expressions](#)) are evaluated using *lazy evaluation*, which means that they are evaluated only as needed. All other expressions

are evaluated using *eager evaluation*, which means that they are evaluated immediately when encountered during the evaluation process. A good way to think about this is to remember that evaluating a list or record expression returns a list or record value that itself remembers how its list items or record fields need to be computed, when requested (by lookup or index operators).

Functions

In M, a *function* is a mapping from a set of input values to a single output value. A function is written by first naming the required set of input values (the parameters to the function) and then providing an expression that computes the result of the function using those input values (the body of the function) following the goes-to (`=>`) symbol. For example:

Power Query M

```
(x) => x + 1           // function that adds one to a value
(x, y) => x + y        // function that adds two values
```

A function is a value just like a number or a text value. The following example shows a function that is the value of an Add field which is then *invoked*, or executed, from several other fields. When a function is invoked, a set of values are specified that are logically substituted for the required set of input values within the function body expression.

Power Query M

```
[
    Add = (x, y) => x + y,
    OnePlusOne = Add(1, 1),    // 2
    OnePlusTwo = Add(1, 2)     // 3
]
```

Library

M includes a common set of definitions available for use from an expression called the *standard library*, or just *library* for short. These definitions consist of a set of named values. The names of values provided by a library are available for use within an expression without having been defined explicitly by the expression. For example:

Power Query M

```
Number.E // Euler's number e (2.7182...)
Text.PositionOf("Hello", "ll") // 2
```

Operators

M includes a set of operators that can be used in expressions. *Operators* are applied to *operands* to form symbolic expressions. For example, in the expression `1 + 2` the numbers `1` and `2` are operands and the operator is the addition operator (`+`).

The meaning of an operator can vary depending on what kind of values its operands are. For example, the plus operator can be used with other kinds of values besides numbers:

Power Query M

```
1 + 2 // numeric addition: 3
#time(12,23,0) + #duration(0,0,2,0)
// time arithmetic: #time(12,25,0)
```

Another example of an operator with operand-dependent meaning is the combination operator (`&`):

Power Query M

```
"A" & "BC" // text concatenation: "ABC"
{1} & {2, 3} // list concatenation: {1, 2, 3}
[ a = 1 ] & [ b = 2 ] // record merge: [ a = 1, b = 2 ]
```

Note that some operators don't support all combinations of values. For example:

Power Query M

```
1 + "2" // error: adding number and text isn't supported
```

Expressions that, when evaluated, encounter undefined operator conditions evaluate to [errors](#).

Metadata

Metadata is information about a value that's associated with a value. Metadata is represented as a record value, called a *metadata record*. The fields of a metadata record

can be used to store the metadata for a value.

Every value has a metadata record. If the value of the metadata record hasn't been specified, then the metadata record is empty (has no fields).

Metadata records provide a way to associate additional information with any kind of value in an unobtrusive way. Associating a metadata record with a value doesn't change the value or its behavior.

A metadata record value `y` is associated with an existing value `x` using the syntax `x meta y`. For example, the following code associates a metadata record with `Rating` and `Tags` fields with the text value `"Mozart"`:

Power Query M

```
"Mozart" meta [ Rating = 5, Tags = {"Classical"} ]
```

For values that already carry a non-empty metadata record, the result of applying `meta` is that of computing the record merge of the existing and the new metadata record. For example, the following two expressions are equivalent to each other and to the previous expression:

Power Query M

```
("Mozart" meta [ Rating = 5 ]) meta [ Tags = {"Classical"} ]  
"Mozart" meta ([ Rating = 5 ] & [ Tags = {"Classical"} ])
```

A metadata record can be accessed for a given value using the [Value.Metadata](#) function. In the following example, the expression in the `ComposerRating` field accesses the metadata record of the value in the `Composer` field, and then accesses the `Rating` field of the metadata record.

Power Query M

```
[  
    Composer = "Mozart" meta [ Rating = 5, Tags = {"Classical"} ],  
    ComposerRating = Value.Metadata(Composer)[Rating] // 5  
]
```

Let expression

Many of the examples shown so far have included all the literal values of the expression in the result of the expression. The `let` expression allows a set of values to be

computed, assigned names, and then used in a subsequent expression that follows the `in`. For example, in our sales data example, you could do:

Power Query M

```
let
    Sales2007 =
        [
            Year = 2007,
            FirstHalf = 1000,
            SecondHalf = 1100,
            Total = FirstHalf + SecondHalf // 2100
        ],
    Sales2008 =
        [
            Year = 2008,
            FirstHalf = 1200,
            SecondHalf = 1300,
            Total = FirstHalf + SecondHalf // 2500
        ]
in Sales2007[Total] + Sales2008[Total] // 4600
```

The result of the above expression is a number value (`4600`) that's computed from the values bound to the names `Sales2007` and `Sales2008`.

If expression

The `if` expression selects between two expressions based on a logical condition. For example:

Power Query M

```
if 2 > 1 then
    2 + 2
else
    1 + 1
```

The first expression (`2 + 2`) is selected if the logical expression (`2 > 1`) is true, and the second expression (`1 + 1`) is selected if it's false. The selected expression (in this case `2 + 2`) is evaluated and becomes the result of the `if` expression (`4`).

Errors

An *error* is an indication that the process of evaluating an expression couldn't produce a value.

Errors are raised by operators and functions encountering error conditions or by using the error expression. Errors are handled using the `try` expression. When an error is raised, a value is specified that can be used to indicate why the error occurred.

Power Query M

```
let Sales =  
    [  
        Revenue = 2000,  
        Units = 1000,  
        UnitPrice = if Units = 0 then error "No Units"  
                     else Revenue / Units  
    ],  
    UnitPrice = try Number.ToText(Sales[UnitPrice])  
in "Unit Price: " &  
    (if UnitPrice[HasError] then UnitPrice[Error][Message]  
     else UnitPrice[Value])
```

The above example accesses the `Sales[UnitPrice]` field and formats the value producing the result:

Power Query M

"Unit Price: 2"

If the `Units` field had been zero, then the `UnitPrice` field would have raised an error, which would have been handled by the `try`. The resulting value would then have been:

Power Query M

"No Units"

A `try` expression converts proper values and errors into a record value that indicates whether the `try` expression handled an error, or not, and either the proper value or the error record it extracted when handling the error. For example, consider the following expression that raises an error and then handles it right away:

Power Query M

```
try error "negative unit count"
```

This expression evaluates to the following nested record value, explaining the `[HasError]`, `[Error]`, and `[Message]` field lookups in the previous unit-price example.

Power Query M

```
[
    HasError = true,
    Error =
        [
            Reason = "Expression.Error",
            Message = "negative unit count",
            Detail = null
        ]
]
```

A common case is to replace errors with default values. The `try` expression can be used with an optional `otherwise` clause to achieve just that in a compact form:

Power Query M

```
try error "negative unit count" otherwise 42
// 42
```

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Lexical Structure

Article • 08/09/2022

Documents

An M *document* is an ordered sequence of Unicode characters. M allows different classes of Unicode characters in different parts of an M document. For information on Unicode character classes, see *The Unicode Standard, Version 3.0*, section 4.5.

A document either consists of exactly one *expression* or of groups of *definitions* organized into *sections*. Sections are described in detail in Chapter 10. Conceptually speaking, the following steps are used to read an expression from a document:

1. The document is decoded according to its character encoding scheme into a sequence of Unicode characters.
2. Lexical analysis is performed, thereby translating the stream of Unicode characters into a stream of tokens. The remaining subsections of this section cover lexical analysis.
3. Syntactic analysis is performed, thereby translating the stream of tokens into a form that can be evaluated. This process is covered in subsequent sections.

Grammar conventions

The lexical and syntactic grammars are presented using *grammar productions*. Each grammar production defines a non-terminal symbol and the possible expansions of that nonterminal symbol into sequences of non-terminal or terminal symbols. In grammar productions, *non-terminal*+ symbols are shown in italic type, and *terminal* symbols are shown in a fixed-width font.

The first line of a grammar production is the name of the non-terminal symbol being defined, followed by a colon. Each successive indented line contains a possible expansion of the nonterminal given as a sequence of non-terminal or terminal symbols. For example, the production:

if-expression:

```
if if-condition then true-expression else false-expression
```

defines an *if-expression* to consist of the token `if`, followed by an *if-condition*, followed by the token `then`, followed by a *true-expression*, followed by the token `else`, followed

by a *false-expression*.

When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separate lines. For example, the production:

variable-list:

variable

variable-list , *variable*

defines a *variable-list* to either consist of a *variable* or consist of a *variable-list* followed by a *variable*. In other words, the definition is recursive and specifies that a variable list consists of one or more variables, separated by commas.

A subscripted suffix "opt" is used to indicate an optional symbol. The production:

field-specification:

optional_{opt} *field-name* = *field-type*

is shorthand for:

field-specification:

field-name = *field-type*

optional *field-name* = *field-type*

and defines a *field-specification* to optionally begin with the terminal symbol optional followed by a *field-name*, the terminal symbol =, and a *field-type*.

Alternatives are normally listed on separate lines, though in cases where there are many alternatives, the phrase "one of" may precede a list of expansions given on a single line. This is simply shorthand for listing each of the alternatives on a separate line. For example, the production:

decimal-digit: one of

0 1 2 3 4 5 6 7 8 9

is shorthand for:

decimal-digit:

0

1

2

3

4

5

6

7

8

9

Lexical Analysis

The *lexical-unit* production defines the lexical grammar for an M document. Every valid M document conforms to this grammar.

lexical-unit:

*lexical-elements*_{opt}

lexical-elements:

lexical-element

lexical-element

lexical-elements

lexical-element:

whitespace

token comment

At the lexical level, an M document consists of a stream of *whitespace*, *comment*, and *token* elements. Each of these productions is covered in the following sections. Only *token* elements are significant in the syntactic grammar.

Whitespace

Whitespace is used to separate comments and tokens within an M document. Whitespace includes the space character (which is part of Unicode class Zs), as well as horizontal and vertical tab, form feed, and newline character sequences. Newline character sequences include carriage return, line feed, carriage return followed by line feed, next line, and paragraph separator characters.

whitespace:

Any character with Unicode class Zs

Horizontal tab character (U+0009)

Vertical tab character (U+000B)

Form feed character (U+000C)

Carriage return character (U+000D) followed by line feed character (U+000A)

new-line-character

new-line-character:

Carriage return character (U+000D)

Line feed character (U+000A)

Next line character (U+0085)

Line separator character (U+2028)

Paragraph separator character (U+2029)

For compatibility with source code editing tools that add end-of-file markers, and to enable a document to be viewed as a sequence of properly terminated lines, the following transformations are applied, in order, to an M document:

- If the last character of the document is a Control-Z character (U+001A), this character is deleted.
- A carriage-return character (U+000D) is added to the end of the document if that document is non-empty and if the last character of the document is not a carriage return (U+000D), a line feed (U+000A), a line separator (U+2028), or a paragraph separator (U+2029).

Comments

Two forms of comments are supported: single-line comments and delimited comments. *Single-line comments* start with the characters `//` and extend to the end of the source line. *Delimited comments* start with the characters `/*` and end with the characters `*/`.

Delimited comments may span multiple lines.

comment:

single-line-comment

delimited-comment

single-line-comment:

`//` *single-line-comment-characters*_{opt}

single-line-comment-characters:

single-line-comment-character *single-line-comment-characters*_{opt}

single-line-comment-character:

Any Unicode character except a *new-line-character*

delimited-comment:

`/*` *delimited-comment-text*_{opt} *asterisks* `/`

delimited-comment-text:

delimited-comment-section *delimited-comment-text*_{opt}

delimited-comment-section:

`/`

asterisks_{opt} not-slash-or-asterisk

asterisks:

** asterisks_{opt}*

not-slash-or-asterisk:

Any Unicode character except *** or */*

Comments do not nest. The character sequences */** and **/* have no special meaning within a single-line comment, and the character sequences *//* and */** have no special meaning within a delimited comment.

Comments are not processed within text literals. The example

Power Query M

```
/* Hello, world
*/
    "Hello, world"
```

includes a delimited comment.

The example

Power Query M

```
// Hello, world
//
"Hello, world" // This is an example of a text literal
```

shows several single-line comments.

Tokens

A *token* is an identifier, keyword, literal, operator, or punctuator. Whitespace and comments are used to separate tokens, but are not considered tokens.

token:

identifier

keyword

literal

operator-or-punctuator

Character Escape Sequences

M text values can contain arbitrary Unicode characters. Text literals, however, are limited to graphic characters and require the use of *escape sequences* for non-graphic characters. For example, to include a carriage-return, linefeed, or tab character in a text literal, the `#(cr)`, `#(lf)`, and `#(tab)` escape sequences can be used, respectively. To embed the escape sequence start characters `#(` in a text literal, the `#` itself needs to be escaped:

Power Query M

```
#(#)(
```

Escape sequences can also contain short (four hex digits) or long (eight hex digits) Unicode code-point values. The following three escape sequences are therefore equivalent:

Power Query M

```
#(000D)      // short Unicode hexadecimal value
#(0000000D)  // long Unicode hexadecimal value
#(cr)        // compact escape shorthand for carriage return
```

Multiple escape codes can be included in a single escape sequence, separated by commas; the following two sequences are thus equivalent:

Power Query M

```
#(cr,lf)
#(cr)#(lf)
```

The following describes the standard mechanism of character escaping in an M document.

character-escape-sequence:

`#( escape-sequence-list )`

escape-sequence-list:

single-escape-sequence

single-escape-sequence , *escape-sequence-list*

single-escape-sequence:

long-unicode-escape-sequence

short-unicode-escape-sequence

control-character-escape-sequence

escape-escape

long-unicode-escape-sequence:

hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

short-unicode-escape-sequence:

hex-digit hex-digit hex-digit hex-digit

control-character-escape-sequence:

control-character

control-character:

`cr`

`lf`

`tab`

escape-escape:

`#`

Literals

A *literal* is a source code representation of a value.

literal:

logical-literal

number-literal

text-literal

null-literal

verbatim-literal

Null literals

The null literal is used to write the `null` value. The `null` value represents an absent value.

null-literal:

`null`

Logical literals

A logical literal is used to write the values `true` and `false` and produces a logical value.

logical-literal:

`true`

`false`

Number literals

A number literal is used to write a numeric value and produces a number value.

number-literal:

decimal-number-literal

hexadecimal-number-literal

decimal-number-literal:

decimal-digits `.` *decimal-digits* *exponent-part*_{opt}

`.` *decimal-digits* *exponent-part*_{opt}

decimal-digits *exponent-part*_{opt}

decimal-digits:

decimal-digit *decimal-digits*_{opt}

decimal-digit: one of

`0 1 2 3 4 5 6 7 8 9`

exponent-part:

`e` *sign*_{opt} *decimal-digits*

`E` *sign*_{opt} *decimal-digits*

sign: one of

`+` `-`

hexadecimal-number-literal:

`0x` *hex-digits*

`0X` *hex-digits*

hex-digits:

hex-digit *hex-digits*_{opt}

hex-digit: one of

`0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`

A number can be specified in hexadecimal format by preceding the *hex-digits* with the characters `0x`. For example:

```
Power Query M
```

```
0xff // 255
```

Note that if a decimal point is included in a number literal, then it must have at least one digit following it. For example, `1.3` is a number literal but `1.` and `1.e3` are not.

Text literals

A text literal is used to write a sequence of Unicode characters and produces a text value.

text-literal:

`" text-literal-charactersopt "`

text-literal-characters:

`text-literal-character text-literal-charactersopt`

text-literal-character:

`single-text-character`

`character-escape-sequence`

`double-quote-escape-sequence`

single-text-character:

Any character except `"` (U+0022) or `#` (U+0023) followed by `(` (U+0028)

double-quote-escape-sequence:

`""` (U+0022, U+0022)

To include quotes in a text value, the quote mark is repeated, as follows:

Power Query M

```
"The ""quoted"" text" // The "quoted" text
```

The [character-escape-sequence](#) production can be used to write characters in text values without having to directly encode them as Unicode characters in the document. For example, a carriage return and line feed can be written in a text value as:

Power Query M

```
"Hello world#(cr,lf)"
```

Verbatim literals

A verbatim literal is used to store a sequence of Unicode characters that were entered by a user as code, but which cannot be correctly parsed as code. At runtime, it produces an error value.

verbatim-literal:

`#!" text-literal-charactersopt "`

Identifiers

An *identifier* is a name used to refer to a value. Identifiers can either be regular identifiers or quoted identifiers.

identifier:

regular-identifier

quoted-identifier

regular-identifier:

available-identifier

available-identifier dot-character regular-identifier

available-identifier:

A *keyword-or-identifier* that is not a *keyword*

keyword-or-identifier:

*identifier-start-character identifier-part-characters*_{opt}

identifier-start-character:

letter-character

underscore-character

identifier-part-characters:

*identifier-part-character identifier-part-characters*_{opt}

identifier-part-character:

letter-character

decimal-digit-character

underscore-character

connecting-character

combining-character

formatting-character

dot-character:

 (U+002E)

underscore-character:

 (U+005F)

letter-character:

A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:

A Unicode character of classes Mn or Mc

decimal-digit-character:

A Unicode character of the class Nd

connecting-character:

A Unicode character of the class Pc

formatting-character:

A Unicode character of the class Cf

A *quoted-identifier* can be used to allow any sequence of zero or more Unicode characters to be used as an identifier, including keywords, whitespace, comments, operators and punctuators.

quoted-identifier:

```
#" text-literal-charactersopt "
```

Note that escape sequences and double-quotes to escape quotes can be used in a *quoted identifier*, just as in a *text-literal*.

The following example uses identifier quoting for names containing a space character:

Power Query M

```
[
    #"1998 Sales" = 1000,
    #"1999 Sales" = 1100,
    #"Total Sales" = #"1998 Sales" + #"1999 Sales"
]
```

The following example uses identifier quoting to include the `+` operator in an identifier:

Power Query M

```
[
    #"A + B" = A + B,
    A = 1,
    B = 2
]
```

Generalized Identifiers

There are two places in M where no ambiguities are introduced by identifiers that contain blanks or that are otherwise keywords or number literals. These places are the names of record fields in a record literal and in a field access operator (`[ ]`). There, M allows such identifiers without having to use quoted identifiers.

Power Query M

```
[
    Data = [ Base Line = 100, Rate = 1.8 ],
    Progression = Data[Base Line] * Data[Rate]
]
```

The identifiers used to name and access fields are referred to as *generalized identifiers* and defined as follows:

generalized-identifier:

generalized-identifier-part

generalized-identifier separated only by blanks (U+0020)

generalized-identifier-part

generalized-identifier-part:

generalized-identifier-segment

decimal-digit-character generalized-identifier-segment

generalized-identifier-segment:

keyword-or-identifier

keyword-or-identifier dot-character keyword-or-identifier

Keywords

A *keyword* is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when using the [identifier-quoting mechanism](#) or where a [generalized identifier is allowed](#).

keyword: one of

and as each else error false if in is let meta not null or otherwise

section shared then true try type #binary #date #datetime

#datetimezone #duration #infinity #nan #sections #shared #table #time

Operators and punctuators

There are several kinds of operators and punctuators. Operators are used in expressions to describe operations involving one or more operands. For example, the expression `a + b` uses the `+` operator to add the two operands `a` and `b`. Punctuators are for grouping and separating.

operator-or-punctuator: one of

, ; = < <= > >= <> + - * / & () [] { } @ ! ? ?? =>

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Basic concepts

Article • 08/22/2023

This section discusses basic concepts that appear throughout the subsequent sections.

Values

A single piece of data is called a *value*. Broadly speaking, there are two general categories of values: *primitive values*, which are atomic, and *structured values*, which are constructed out of primitive values and other structured values. For example, the values

Power Query M

```
1
true
3.14159
"abc"
```

are primitive in that they are not made up of other values. On the other hand, the values

Power Query M

```
{1, 2, 3}
[ A = {1}, B = {2}, C = {3} ]
```

are constructed using primitive values and, in the case of the record, other structured values.

Expressions

An *expression* is a formula used to construct values. An expression can be formed using a variety of syntactic constructs. The following are some examples of expressions. Each line is a separate expression.

Power Query M

```
"Hello World"           // a text value
123                     // a number
1 + 2                   // sum of two numbers
{1, 2, 3}                // a list of three numbers
[ x = 1, y = 2 + 3 ]     // a record containing two fields:
                        //           x and y
(x, y) => x + y          // a function that computes a sum
```

```
if 2 > 1 then 2 else 1    // a conditional expression
let x = 1 + 1 in x * 2    // a let expression
error "A"                // error with message "A"
```

The simplest form of expression, as seen above, is a literal representing a value.

More complex expressions are built from other expressions, called *sub-expressions*. For example:

Power Query M

1 + 2

The above expression is actually composed of three expressions. The `1` and `2` literals are subexpressions of the parent expression `1 + 2`.

Executing the algorithm defined by the syntactic constructs used in an expression is called *evaluating* the expression. Each kind of expression has rules for how it is evaluated. For example, a literal expression like `1` will produce a constant value, while the expression `a + b` will take the resulting values produced by evaluating two other expressions (`a` and `b`) and add them together according to some set of rules.

Environments and variables

Expressions are evaluated within a given environment. An *environment* is a set of named values, called *variables*. Each variable in an environment has a unique name within the environment called an *identifier*.

A top-level (or *root*) expression is evaluated within the *global environment*. The global environment is provided by the expression evaluator instead of being determined from the contents of the expression being evaluated. The contents of the global environment includes the standard library definitions and can be affected by exports from sections from some set of documents. (For simplicity, the examples in this section will assume an empty global environment. That is, it is assumed that there is no standard library and that there are no other section-based definitions.)

The environment used to evaluate a sub-expression is determined by the parent expression. Most parent expression kinds will evaluate a sub-expression within the same environment they were evaluated within, but some will use a different environment. The global environment is the *parent environment* within which the global expression is evaluated.

For example, the *record-initializer-expression* evaluates the sub-expression for each field with a modified environment. The modified environment includes a variable for each of the fields of the record, except the one being initialized. Including the other fields of the record allows the fields to depend upon the values of the fields. For example:

Power Query M

```
[
    x = 1,          // environment: y, z
    y = 2,          // environment: x, z
    z = x + y       // environment: x, y
]
```

Similarly, the *let-expression* evaluates the sub-expression for each variable with an environment containing each of the variables of the let except the one being initialized. The *let-expression* evaluates the expression following the in with an environment containing all the variables:

Power Query M

```
let
    x = 1,          // environment: y, z
    y = 2,          // environment: x, z
    z = x + y       // environment: x, y
in
    x + y + z       // environment: x, y, z
```

(It turns out that both *record-initializer-expression* and *let-expression* actually define *two* environments, one of which does include the variable being initialized. This is useful for advanced recursive definitions and is covered in [Identifier references](#) .

To form the environments for the sub-expressions, the new variables are "merged" with the variables in the parent environment. The following example shows the environments for nested records:

Power Query M

```
[
    a =
    [
        x = 1,      // environment: b, y, z
        y = 2,      // environment: b, x, z
        z = x + y    // environment: b, x, y
    ],

```

```
b = 3          // environment: a
]
```

The following example shows the environments for a record nested within a let:

Power Query M

```
Let
  a =
  [
    x = 1,      // environment: b, y, z
    y = 2,      // environment: b, x, z
    z = x + y    // environment: b, x, y
  ],
  b = 3          // environment: a
in
  a[z] + b       // environment: a, b
```

Merging variables with an environment may introduce a conflict between variables (since each variable in an environment must have a unique name). The conflict is resolved as follows: if the name of a new variable being merged is the same as an existing variable in the parent environment, then the new variable will take precedence in the new environment. In the following example, the inner (more deeply nested) variable `x` will take precedence over the outer variable `x`.

Power Query M

```
[
  a =
  [
    x = 1,      // environment: b, x (outer), y, z
    y = 2,      // environment: b, x (inner), z
    z = x + y    // environment: b, x (inner), y
  ],
  b = 3,        // environment: a, x (outer)
  x = 4         // environment: a, b
]
```

Identifier references

An *identifier-reference* is used to refer to a variable within an environment.

identifier-expression:

identifier-reference

identifier-reference:

exclusive-identifier-reference

inclusive-identifier-reference

The simplest form of identifier reference is an *exclusive-identifier-reference*:

exclusive-identifier-reference:

identifier

It is an error for an *exclusive-identifier-reference* to refer to a variable that is not part of the environment of the expression that the identifier appears within.

It is an error for an *exclusive-identifier-reference* to refer to an identifier that is currently being initialized if the referenced identifier is defined inside a *record-initializer-expression* or *let-expression*. Instead, an *inclusive-identifier-reference* can be used to gain access to the environment that includes the identifier being initialized. If an *inclusive-identifier-reference* is used in any other situation, then it is equivalent to an *exclusive-identifier-reference*.

inclusive-identifier-reference:

@ *identifier*

This is useful when defining recursive functions since the name of the function would normally not be in scope.

Power Query M

```
[
    Factorial = (n) =>
        if n <= 1 then
            1
        else
            n * @Factorial(n - 1), // @ is scoping operator

    x = Factorial(5)
]
```

As with a *record-initializer-expression*, an *inclusive-identifier-reference* can be used within a *let-expression* to access the environment that includes the identifier being initialized.

Order of evaluation

Consider the following expression which initializes a record:

Power Query M

```
[  
  C = A + B,  
  A = 1 + 1,  
  B = 2 + 2  
]
```

When evaluated, this expression produces the following record value:

Power Query M

```
[  
  C = 6,  
  A = 2,  
  B = 4  
]
```

The expression states that in order to perform the `A + B` calculation for field `C`, the values of both field `A` and field `B` must be known. This is an example of a *dependency ordering* of calculations that is provided by an expression. The M evaluator abides by the dependency ordering provided by expressions, but is free to perform the remaining calculations in any order it chooses. For example, the computation order could be:

Power Query M

```
A = 1 + 1  
B = 2 + 2  
C = A + B
```

Or it could be:

Power Query M

```
B = 2 + 2  
A = 1 + 1  
C = A + B
```

Or, since `A` and `B` do not depend on each other, they can be computed concurrently:

```
B = 2 + 2 concurrently with A = 1 + 1  
C = A + B
```

Side effects

Allowing an expression evaluator to automatically compute the order of calculations for cases where there are no explicit dependencies stated by the expression is a simple and powerful computation model.

It does, however, rely on being able to reorder computations. Since expressions can call functions, and those functions could observe state external to the expression by issuing external queries, it is possible to construct a scenario where the order of calculation does matter, but is not captured in the partial order of the expression. For example, a function may read the contents of a file. If that function is called repeatedly, then external changes to that file can be observed and, therefore, reordering can cause observable differences in program behavior. Depending on such observed evaluation ordering for the correctness of an M expression causes a dependency on particular implementation choices that might vary from one evaluator to the next or may even vary on the same evaluator under varying circumstances.

Immutability

Once a value has been calculated, it is *immutable*, meaning it can no longer be changed. This simplifies the model for evaluating an expression and makes it easier to reason about the result since it is not possible to change a value once it has been used to evaluate a subsequent part of the expression. For instance, a record field is only computed when needed. However, once computed, it remains fixed for the lifetime of the record. Even if the attempt to compute the field raised an error, that same error will be raised again on every attempt to access that record field.

An important exception to the immutable-once-calculated rule applies to list, table, and binary values, which have *streaming semantics*. Streaming semantics allow M to transform data sets that don't fit into memory all at once. With streaming, the values returned when enumerating a given table, list, or binary value are produced on demand each time they're requested. Since the expressions defining the enumerated values are evaluated each time they're enumerated, the output they produce can be different across multiple enumerations. This doesn't mean that multiple enumerations always results in different values, just that they can be different if the data source or M logic being used is non-deterministic.

Also, note that function application is *not* the same as value construction. Library functions may expose external state (such as the current time or the results of a query against a database that evolves over time), rendering them *non-deterministic*. While functions defined in M will not, as such, expose any such non-deterministic behavior, they can if they are defined to invoke other functions that are non-deterministic.

A final source of non-determinism in M are *errors*. Errors stop evaluations when they occur (up to the level where they are handled by a try expression). It is not normally observable whether `a + b` caused the evaluation of `a` before `b` or `b` before `a` (ignoring concurrency here for simplicity). However, if the subexpression that was evaluated first raises an error, then it can be determined which of the two expressions was evaluated first.

Feedback

Was this page helpful?



Yes



No

[Get help at Microsoft Q&A](#)

Values

Article • 03/28/2025

A value is data produced by evaluating an expression. This section describes the kinds of values in the M language. Each kind of value is associated with a literal syntax, a set of values that are of that kind, a set of operators defined over that set of values, and an intrinsic type ascribed to newly constructed values.

[Expand table](#)

Kind	Literal
<i>Null</i>	<code>null</code>
<i>Logical</i>	<code>true</code> <code>false</code>
<i>Number</i>	<code>0</code> <code>1</code> <code>-1</code> <code>1.5</code> <code>2.3e-5</code>
<i>Time</i>	<code>#time(09,15,00)</code>
<i>Date</i>	<code>#date(2013,02,26)</code>
<i>DateTime</i>	<code>#datetime(2013,02,26, 09,15,00)</code>
<i>DateTimeZone</i>	<code>#datetimezone(2013,02,26, 09,15,00, 09,00)</code>
<i>Duration</i>	<code>#duration(0,1,30,0)</code>
<i>Text</i>	<code>"hello"</code>
<i>Binary</i>	<code>#binary("AQID")</code>
<i>List</i>	<code>{1, 2, 3}</code>
<i>Record</i>	<code>[A = 1, B = 2]</code>
<i>Table</i>	<code>#table({"X", "Y"}, {{0,1},{1,0}})</code>
<i>Function</i>	<code>(x) => x + 1</code>
<i>Type</i>	<code>type { number }</code> <code>type table [A = any, B = text]</code>

The following sections cover each value kind in detail. Types and type ascription are defined formally in [Types](#). Function values are defined in [Functions](#). The following sections list the operators defined for each value kind and give examples. The full definition of operator semantics follows in [Operators](#).

Null

A *null value* is used to represent the absence of a value, or a value of indeterminate or unknown state. A null value is written using the literal `null`. The following operators are defined for null values:

 Expand table

Operator	Result
<code>x > y</code>	Greater than
<code>x >= y</code>	Greater than or equal
<code>x < y</code>	Less than
<code>x <= y</code>	Less than or equal
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x ?? y</code>	Coalesce

The native type of the `null` value is the intrinsic type `null`.

Logical

A *logical value* is used for Boolean operations has the value true or false. A logical value is written using the literals `true` and `false`. The following operators are defined for logical values:

 Expand table

Operator	Result
<code>x > y</code>	Greater than
<code>x >= y</code>	Greater than or equal
<code>x < y</code>	Less than
<code>x <= y</code>	Less than or equal
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal

Operator	Result
<code>x or y</code>	Conditional logical OR
<code>x ?? y</code>	Coalesce
<code>x and y</code>	Conditional logical AND
<code>not x</code>	Logical NOT

The native type of both logical values (`true` and `false`) is the intrinsic type `logical`.

Number

A *number value* is used for numeric and arithmetic operations. The following are examples of number literals:

Power Query M

```
3.14 // Fractional number
-1.5 // Fractional number
1.0e3 // Fractional number with exponent
123 // Whole number
1e3 // Whole number with exponent
0xff // Whole number in hex (255)
```

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 have an approximate dynamic range from 5.0×10^{324} to 1.7×10^{308} with a precision of 15-16 digits.)

The following special values are also considered to be *number* values:

- Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but [certain operations distinguish between the two](#).
- Positive infinity (`#infinity`) and negative infinity (`-#infinity`). Infinities are produced by such operations as dividing a non-zero number by zero. For example, `1.0 / 0.0` yields positive infinity, and `-1.0 / 0.0` yields negative infinity.
- The *Not-a-Number* value (`#nan`), often abbreviated NaN. NaNs are produced by invalid floating-point operations, such as dividing zero by zero.

Binary mathematical operations are performed using a *Precision*. The precision determines the domain to which the operands are rounded and the domain in which the operation is performed. In the absence of an explicitly specified precision, such operations are performed using *Double Precision*.

- If the result of a mathematical operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
- If the result of a mathematical operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
- If a mathematical operation is invalid, the result of the operation becomes NaN.
- If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

The following operators are defined for number values:

[Expand table](#)

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

The native type of number values is the intrinsic type `number`.

Time

A *time value* stores an opaque representation of time of day. A time is encoded as the number of *ticks since midnight*, which counts the number of 100-nanosecond ticks that have elapsed on a 24-hour clock. The maximum number of *ticks since midnight* corresponds to 23:59:59.9999999 hours.

Although there is no literal syntax for times, several standard library functions are provided to construct them. Times may also be constructed using the intrinsic function `#time`:

Power Query M

```
#time(hour, minute, second)
```

The following must hold or an error with reason code `Expression.Error` is raised:

- $0 \leq \text{hour} \leq 24$
- $0 \leq \text{minute} \leq 59$
- $0 \leq \text{second} \leq 59$

In addition, if hour = 24, then minute and second must be zero.

The following operators are defined for time values:

 Expand table

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x >= y</code>	Greater than or equal
<code>x > y</code>	Greater than
<code>x < y</code>	Less than
<code>x <= y</code>	Less than or equal
<code>x ?? y</code>	Coalesce

The following operators permit one or both of their operands to be a date:

Operator	Left Operand	Right Operand	Meaning
<code>x + y</code>	<code>time</code>	<code>duration</code>	Date offset by duration
<code>x + y</code>	<code>duration</code>	<code>time</code>	Date offset by duration
<code>x - y</code>	<code>time</code>	<code>duration</code>	Date offset by negated duration
<code>x - y</code>	<code>time</code>	<code>time</code>	Duration between dates
<code>x & y</code>	<code>date</code>	<code>time</code>	Merged datetime

The native type of time values is the intrinsic type `time`.

Date

A *date value* stores an opaque representation of a specific day. A date is encoded as a 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.

Although there is no literal syntax for dates, several standard library functions are provided to construct them. Dates may also be constructed using the intrinsic function `#date`:

Power Query M

```
#date(year, month, day)
```

The following must hold or an error with reason code `Expression.Error` is raised:

$1 \leq \text{year} \leq 9999$

$1 \leq \text{month} \leq 12$

$1 \leq \text{day} \leq 31$

In addition, the day must be valid for the chosen month and year.

The following operators are defined for date values:

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x >= y</code>	Greater than or equal
<code>x > y</code>	Greater than
<code>x < y</code>	Less than
<code>x <= y</code>	Less than or equal
<code>x ?? y</code>	Coalesce

The following operators permit one or both of their operands to be a date:

 Expand table

Operator	Left Operand	Right Operand	Meaning
<code>x + y</code>	<code>date</code>	<code>duration</code>	Date offset by duration
<code>x + y</code>	<code>duration</code>	<code>date</code>	Date offset by duration
<code>x - y</code>	<code>date</code>	<code>duration</code>	Date offset by negated duration
<code>x - y</code>	<code>date</code>	<code>date</code>	Duration between dates
<code>x & y</code>	<code>date</code>	<code>time</code>	Merged datetime

The native type of date values is the intrinsic type `date`.

DateTime

A *datetime value* contains both a date and time.

Although there is no literal syntax for datetimes, several standard library functions are provided to construct them. Datetimes may also be constructed using the intrinsic function `#datetime`:

Power Query M

#datetime(year, month, day, hour, minute, second)

The following must hold or an error with reason code `Expression.Error` is raised: $1 \leq \text{year} \leq 9999$
 $1 \leq \text{month} \leq 12$
 $1 \leq \text{day} \leq 31$
 $0 \leq \text{hour} \leq 23$
 $0 \leq \text{minute} \leq 59$
 $0 \leq \text{second} \leq 59$

In addition, the day must be valid for the chosen month and year.

The following operators are defined for datetime values:

[Expand table](#)

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x >= y</code>	Greater than or equal
<code>x > y</code>	Greater than
<code>x < y</code>	Less than
<code>x <= y</code>	Less than or equal
<code>x ?? y</code>	Coalesce

The following operators permit one or both of their operands to be a datetime:

[Expand table](#)

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

The native type of datetime values is the intrinsic type `datetime`.

DateTimeZone

A *datetimezone* value contains a datetime and a timezone. A *timezone* is encoded as a number of *minutes offset from UTC*, which counts the number of minutes the time portion of the *datetime* should be offset from Universal Coordinated Time (UTC). The minimum number of *minutes offset from UTC* is -840, representing a UTC offset of -14:00, or fourteen hours earlier than UTC. The maximum number of *minutes offset from UTC* is 840, corresponding to a UTC offset of 14:00.

Although there is no literal syntax for datetimezones, several standard library functions are provided to construct them. Datetimezones may also be constructed using the intrinsic function `#datetimezone`:

Power Query M

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

The following must hold or an error with reason code `Expression.Error` is raised:

- $1 \leq \text{year} \leq 9999$
- $1 \leq \text{month} \leq 12$
- $1 \leq \text{day} \leq 31$
- $0 \leq \text{hour} \leq 23$
- $0 \leq \text{minute} \leq 59$
- $0 \leq \text{second} \leq 59$
- $-14 \leq \text{offset-hours} \leq 14$
- $-59 \leq \text{offset-minutes} \leq 59$

In addition, the day must be valid for the chosen month and year and, if offset-hours = 14, then offset-minutes <= 0 and, if offset-hours = -14, then offset-minutes >= 0.

The following operators are defined for datetimezone values:

[Expand table](#)

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x >= y</code>	Greater than or equal
<code>x > y</code>	Greater than

Operator	Result
<code>x < y</code>	Less than
<code>x <= y</code>	Less than or equal
<code>x ?? y</code>	Coalesce

The following operators permit one or both of their operands to be a `datetimezone`:

[Expand table](#)

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

The native type of `datetimezone` values is the intrinsic type `datetimezone`.

Duration

A *duration value* stores an opaque representation of the distance between two points on a timeline measured 100-nanosecond ticks. The magnitude of a *duration* can be either positive or negative, with positive values denoting progress forwards in time and negative values denoting progress backwards in time. The minimum value that can be stored in a *duration* is -9,223,372,036,854,775,808 ticks, or 10,675,199 days 2 hours 48 minutes 05.4775808 seconds backwards in time. The maximum value that can be stored in a *duration* is 9,223,372,036,854,775,807 ticks, or 10,675,199 days 2 hours 48 minutes 05.4775807 seconds forwards in time.

Although there is no literal syntax for durations, several standard library functions are provided to construct them. Durations may also be constructed using the intrinsic function `#duration`:

Power Query M

```
#duration(0, 0, 0, 5.5)           // 5.5 seconds
#duration(0, 0, 0, -5.5)          // -5.5 seconds
#duration(0, 0, 5, 30)            // 5.5 minutes
#duration(0, 0, 5, -30)           // 4.5 minutes
```

```
#duration(0, 24, 0, 0)      // 1 day
#duration(1, 0, 0, 0)      // 1 day
```

The following operators are defined on duration values:

[Expand table](#)

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x >= y</code>	Greater than or equal
<code>x > y</code>	Greater than
<code>x < y</code>	Less than
<code>x <= y</code>	Less than or equal
<code>x ?? y</code>	Coalesce

Additionally, the following operators allow one or both of their operands to be a duration value:

[Expand table](#)

Operator	Left Operand	Right Operand	Meaning
<code>x + y</code>	<code>datetime</code>	<code>duration</code>	Datetime offset by duration
<code>x + y</code>	<code>duration</code>	<code>datetime</code>	Datetime offset by duration
<code>x + y</code>	<code>duration</code>	<code>duration</code>	Sum of durations
<code>x - y</code>	<code>datetime</code>	<code>duration</code>	Datetime offset by negated duration
<code>x - y</code>	<code>datetime</code>	<code>datetime</code>	Duration between datetimes
<code>x - y</code>	<code>duration</code>	<code>duration</code>	Difference of durations
<code>x * y</code>	<code>duration</code>	<code>number</code>	N times a duration
<code>x * y</code>	<code>number</code>	<code>duration</code>	N times a duration
<code>x / y</code>	<code>duration</code>	<code>number</code>	Fraction of a duration
<code>x / y</code>	<code>duration</code>	<code>duration</code>	Numeric quotient of durations

The native type of duration values is the intrinsic type `duration`.

Text

A *text* value represents a sequence of Unicode characters. Text values have a literal form conformant to the following grammar:

_text-literal:

`"` *text-literal-characters*_{opt} `"`

text-literal-characters:

text-literal-character *text-literal-characters*_{opt}

text-literal-character:

single-text-character

character-escape-sequence

double-quote-escape-sequence

single-text-character:

Any character except `"` (U+0022) or `#` (U+0023) followed by `(` (U+0028)

double-quote-escape-sequence:

`" "` (U+0022, U+0022)

The following is an example of a *text* value:

Power Query M

"ABC" // the text value ABC

The following operators are defined on *text* values:

 Expand table

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x >= y</code>	Greater than or equal
<code>x > y</code>	Greater than
<code>x < y</code>	Less than
<code>x <= y</code>	Less than or equal

Operator	Result
<code>x & y</code>	Concatenation
<code>x ?? y</code>	Coalesce

The native type of text values is the intrinsic type `text`.

Binary

A *binary value* represents a sequence of bytes.

Although there is no literal syntax for binary values, several standard library functions are provided to construct them. Binary values may also be constructed using the intrinsic function `#binary`.

The following example constructs a binary value from a list of bytes:

Power Query M

```
#binary( {0x00, 0x01, 0x02, 0x03} )
```

The following operators are defined on *binary* values:

 Expand table

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x >= y</code>	Greater than or equal
<code>x > y</code>	Greater than
<code>x < y</code>	Less than
<code>x <= y</code>	Less than or equal
<code>x ?? y</code>	Coalesce

The native type of binary values is the intrinsic type *binary*.

List

A *list value* is a value which produces a sequence of values when enumerated. A value produced by a list can contain any kind of value, including a list. Lists can be constructed using the initialization syntax, as follows:

list-expression:

`{ item-listopt }`

item-list:

`item`

`item , item-list`

item:

`expression`

`expression .. expression`

The following is an example of a *list-expression* that defines a list with three text values:

`"A", "B", and "C"`.

Power Query M

```
{ "A", "B", "C" }
```

The value `"A"` is the first item in the list, and the value `"C"` is the last item in the list.

- The items of a list are not evaluated until they are accessed.
- While list values constructed using the list syntax will produce items in the order they appear in *item-list*, in general, lists returned from library functions may produce a different set or a different number of values each time they are enumerated.

To include a sequence of whole number in a list, the `a..b` form can be used:

Power Query M

```
{ 1, 5..9, 11 }      // { 1, 5, 6, 7, 8, 9, 11 }
```

The number of items in a list, known as the *list count*, can be determined using the

`List.Count` function.

Power Query M

```
List.Count({true, false}) // 2
```

```
List.Count({})           // 0
```


A list may effectively have an infinite number of items; `List.Count` for such lists is undefined and may either raise an error or not terminate.

If a list contains no items, it is called an *empty list*. An empty list is written as:

Power Query M

```
{ } // empty list
```

The following operators are defined for lists:

 Expand table

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x & y</code>	Concatenate
<code>x ?? y</code>	Coalesce

For example:

Power Query M

```
{1, 2} & {3, 4, 5} // {1, 2, 3, 4, 5}
{1, 2} = {1, 2}    // true
{2, 1} <> {1, 2}   // true
```

The native type of list values is the intrinsic type `list`, which specifies an item type of `any`.

Record

A *record value* is an ordered sequence of fields. A *field* consists of a *field name*, which is a text value that uniquely identifies the field within the record, and a *field value*. The field value can be any kind of value, including record. Records can be constructed using initialization syntax, as follows:

record-expression:

`[ field-listopt ]`

field-list:

field

field , *field-list*

field:

field-name = *expression*

field-name:

generalized-identifier

quoted-identifier

The following example constructs a record with a field named `x` with value `1`, and a field named `y` with value `2`.

Power Query M

```
[ x = 1, y = 2 ]
```

The following example constructs a record with a field named `a` with a nested record value. The nested record has a field named `b` with value `2`.

Power Query M

```
[ a = [ b = 2 ] ]
```

The following holds when evaluating a record expression:

- The expression assigned to each field name is used to determine the value of the associated field.
- If the expression assigned to a field name produces a value when evaluated, then that becomes the value of the field of the resulting record.
- If the expression assigned to a field name raises an error when evaluated, then the fact that an error was raised is recorded with the field along with the error value that was raised. Subsequent access to that field will cause an error to be re-raised with the recorded error value.
- The expression is evaluated in an environment like the parent environment only with variables merged in that correspond to the value of every field of the record, except the one being initialized.
- A value in a record is not evaluated until the corresponding field is accessed.
- A value in a record is evaluated at most once.
- The result of the expression is a record value with an empty metadata record.

- The order of the fields within the record is defined by the order that they appear in the *record-initializer-expression*.
- Every field name that is specified must be unique within the record, or it is an error. Names are compared using an ordinal comparison.

Power Query M

```
[ x = 1, x = 2 ] // error: field names must be unique  
[ X = 1, x = 2 ] // OK
```

A record with no fields is called an *empty record*, and is written as follows:

Power Query M

```
[] // empty record
```

Although the order of the fields of a record is not significant when accessing a field or comparing two records, it is significant in other contexts such as when the fields of a record are enumerated.

The same two records produce different results when the fields are obtained:

Power Query M

```
Record.FieldNames([ x = 1, y = 2 ]) // [ "x", "y" ]  
Record.FieldNames([ y = 1, x = 2 ]) // [ "y", "x" ]
```

The number of fields in a record can be determined using the `Record.FieldCount` function. For example:

Power Query M

```
Record.FieldCount([ x = 1, y = 2 }) // 2  
Record.FieldCount([]) // 0
```

In addition to using the record initialization syntax `[ ]`, records can be constructed from a list of values, and a list of field names or a record type. For example:

Power Query M

```
Record.FromList({1, 2}, {"a", "b"})
```

The above is equivalent to:

Power Query M

```
[ a = 1, b = 2 ]
```

The following operators are defined for record values:

[Expand table](#)

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x & y</code>	Merge
<code>x ?? y</code>	Coalesce

The following examples illustrate the above operators. Note that record merge uses the fields from the right operand to override fields from the left operand, should there be an overlap in field names.

Power Query M

```
[ a = 1, b = 2 ] & [ c = 3 ]    // [ a = 1, b = 2, c = 3 ]  
[ a = 1, b = 2 ] & [ a = 3 ]    // [ a = 3, b = 2 ]  
[ a = 1, b = 2 ] = [ b = 2, a = 1 ]    // true  
[ a = 1, b = 2, c = 3 ] <> [ a = 1, b = 2 ] // true
```

The native type of record values is the intrinsic type `record`, which specifies an open empty list of fields.

Table

A *table value* is an ordered sequence of rows. A *row* is an ordered sequence of column values. The table's type determines the length of all rows in the table, the names of the table's columns, the types of the table's columns, and the structure of the table's keys (if any).

Although there is no literal syntax for tables, several standard library functions are provided to construct them. Tables may also be constructed using the intrinsic function `#table`.

The following example constructs a table from a list of column names and a list of rows. The resulting table will contain two columns of `type any` and three rows.

Power Query M

```
#table({"x", "x^2"}, {{1,1}, {2,4}, {3,9}})
```

`#table` can also be used to specify a full table type:

Power Query M

```
#table(  
    type table [Digit = number, Name = text],  
    {{1,"one"}, {2,"two"}, {3,"three"}}  
)
```

Here the new table value has a table type that specifies column names and column types.

The following operators are defined for table values:

[Expand table](#)

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x & y</code>	Concatenation
<code>x ?? y</code>	Coalesce

Table concatenation aligns like-named columns and fills in `null` for columns appearing in only one of the operand tables. The following example illustrates table concatenation:

Power Query M

```
#table({"A","B"}, {{1,2}})  
& #table({"B","C"}, {{3,4}})
```

[Expand table](#)

A	B	C
1	2	null
null	3	4

The native type of table values is a custom table type (derived from the intrinsic type `table`) that lists the column names, specifies all column types to be any, and has no keys. (Go to [Table types](#) for details on table types.)

Function

A *function value* is a value that maps a set of arguments to a single value. The details of *function* values are described in [Functions](#).

Type

A *type value* is a value that classifies other values. The details of *type* values are described in [Types](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Types

Article • 01/29/2025

A *type value* is a value that *classifies* other values. A value that is classified by a type is said to *conform* to that type. The M type system consists of the following kinds of types:

- Primitive types, which classify primitive values (`binary`, `date`, `datetime`, `datetimezone`, `duration`, `list`, `logical`, `null`, `number`, `record`, `text`, `time`, `type`) and also include a number of abstract types (`function`, `table`, `any`, `anynonnull` and `none`)
- Record types, which classify record values based on field names and value types
- List types, which classify lists using a single item base type
- Function types, which classify function values based on the types of their parameters and return values
- Table types, which classify table values based on column names, column types, and keys
- Nullable types, which classifies the value `null` in addition to all the values classified by a base type
- Type types, which classify values that are types

The set of *primitive types* includes the types of primitive values, and a number of *abstract types*, which are types that do not uniquely classify any values: `function`, `table`, `any`, `anynonnull` and `none`. All function values conform to the abstract type `function`, all table values to the abstract type `table`, all values to the abstract type `any`, all non-null values to the abstract type `anynonnull`, and no values to the abstract type `none`. An expression of type `none` must raise an error or fail to terminate since no value could be produced that conforms to type `none`. Note that the primitive types `function` and `table` are abstract because no function or table is directly of those types, respectively. The primitive types `record` and `list` are non-abstract because they represent an open record with no defined fields and a list of type `any`, respectively.

All types that are not members of the closed set of primitive types plus their nullable counterparts are collectively referred to as *custom types*. Custom types can be written using a `type-expression`:

type-expression:

primary-expression

`type` *primary-type*

primary-type:

primitive-or-nullable-primitive-type

record-type

list-type

function-type

table-type

nullable-type

primitive-or-nullable-primitive-type:

`nullable`_{opt} *primitive-type*

primitive-type: one of

`any anynonnull binary date datetime datetimezone duration function list logical`

`none null number record table text time type`

The *primitive-type* names are *contextual keywords* recognized only in a *type* context. The use of parentheses in a *type* context moves the grammar back to a regular expression context, requiring the use of the `type` keyword to move back into a type context. For example, to invoke a function in a *type* context, parentheses can be used:

Power Query M

```
type nullable ( Type.ForList({type number}) )  
// type nullable {number}
```

Parentheses can also be used to access a variable whose name collides with a *primitive-type* name:

Power Query M

```
let record = type [ A = any ] in type {(record)}  
// type {[ A = any ]}
```

The following example defines a type that classifies a list of numbers:

Power Query M

```
type { number }
```

Similarly, the following example defines a custom type that classifies records with mandatory fields named `x` and `y` whose values are numbers:

Power Query M

```
type [ X = number, Y = number ]
```

The ascribed type of a value is obtained using the standard library function [Value.Type](#), as shown in the following examples:

Power Query M

```
Value.Type( 2 )           // type number  
Value.Type( {2} )         // type list  
Value.Type( [ X = 1, Y = 2 ] ) // type record
```

The `is` operator is used to determine whether a value's type is compatible with a given type, as shown in the following examples:

Power Query M

```
1 is number      // true  
1 is text        // false  
{2} is list      // true
```

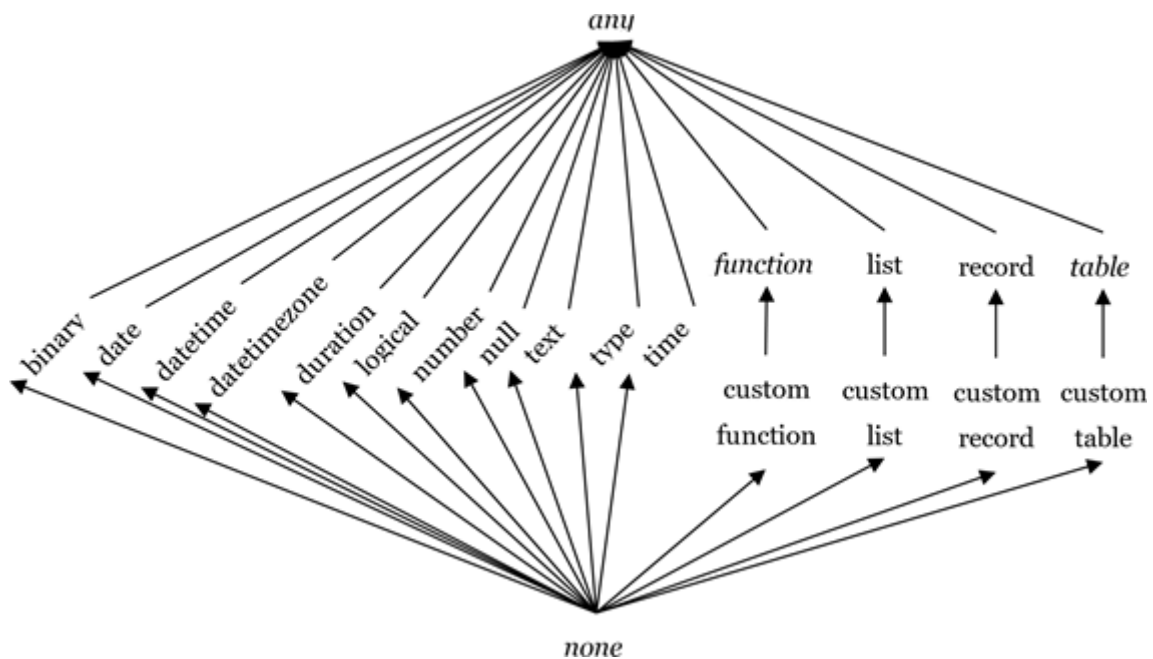
The `as` operator checks if the value is compatible with the given type, and raises an error if it is not. Otherwise, it returns the original value.

Power Query M

```
Value.Type( 1 as number ) // type number  
{2} as text              // error, type mismatch
```

Note that the `is` and `as` operators only accept a *primitive or nullable primitive type* (i.e. a *non-custom type*) as their right operand. M does not provide means to check values for conformance to custom types.

A type `X` is *compatible* with a type `Y` if and only if all values that conform to `X` also conform to `Y`. All types are compatible with type `any` and no types (but `none` itself) are compatible with type `none`. The following graph shows the compatibility relation. (Type compatibility is reflexive and transitive. It forms a lattice with type `any` as the top and type `none` as the bottom value.) The names of abstract types are set in *italics*.



The following operators are defined for type values:

[Expand table](#)

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal
<code>x ?? y</code>	Coalesce

The native type of type values is the intrinsic type `type`.

Primitive Types

Types in the M language form a disjoint hierarchy rooted at type `any`, which is the type that classifies all values. Any M value conforms to exactly one primitive subtype of `any`.

The closed set of primitive types deriving from type `any` are as follows:

- `type null`, which classifies the null value.
- `type logical`, which classifies the values true and false.
- `type number`, which classifies number values.
- `type time`, which classifies time values.
- `type date`, which classifies date values.
- `type datetime`, which classifies datetime values.
- `type datetimezone`, which classifies datetimezone values.
- `type duration`, which classifies duration values.

- `type text`, which classifies text values.
- `type binary`, which classifies binary values.
- `type type`, which classifies type values.
- `type list`, which classifies list values.
- `type record`, which classifies record values.
- `type table`, which classifies table values.
- `type function`, which classifies function values.
- `type anynonnull`, which classifies all values excluding null.
- `type none`, which classifies no values.

Any Type

The type `any` is abstract, classifies all values in M, and all types in M are compatible with `any`. Variables of type `any` can be bound to all possible values. Since `any` is abstract, it cannot be ascribed to values—that is, no value is directly of type `any`.

List Types

Any value that is a list conforms to the intrinsic type `list`, which does not place any restrictions on the items within a list value.

list-type:

```
{ item-type }
```

item-type:

```
type
```

The result of evaluating a *list-type* is a *list type value* whose base type is `list`.

The following examples illustrate the syntax for declaring homogeneous list types:

Power Query M

```
type { number }      // list of numbers type
    { record }       // list of records type
    {{ text }}       // list of lists of text values
```

A value conforms to a list type if the value is a list and each item in that list value conforms to the list type's item type.

The item type of a list type indicates a bound: all items of a conforming list conform to the item type.

Record Types

Any value that is a record conforms to the intrinsic type record, which does not place any restrictions on the field names or values within a record value. A *record-type value* is used to restrict the set of valid names as well as the types of values that are permitted to be associated with those names.

record-type:

```
[ open-record-marker ]  
[ field-specification-listopt ]  
[ field-specification-list , open-record-marker ]
```

field-specification-list:

```
field-specification  
field-specification , field-specification-list
```

field-specification:

```
optionalopt field-name field-type-specificationopt
```

field-type-specification:

```
= field-type
```

field-type:

```
type
```

open-record-marker:

```
...
```

The result of evaluating a *record-type* is a type value whose base type is `record`.

The following examples illustrate the syntax for declaring record types:

Power Query M

```
type [ X = number, Y = number]  
type [ Name = text, Age = number ]  
type [ Title = text, optional Description = text ]  
type [ Name = text, ... ]
```

Record types are *closed* by default, meaning that additional fields not present in the *fieldspecification-list* are not allowed to be present in conforming values. Including the *openrecord-marker* in the record type declares the type to be *open*, which permits fields not present in the field specification list. The following two expressions are equivalent:

Power Query M

```
type record // primitive type classifying all records
```

```
type [ ... ] // custom type classifying all records
```

A value conforms to a record type if the value is a record and each field specification in the record type is satisfied. A field specification is satisfied if any of the following are true:

- A field name matching the specification's identifier exists in the record and the associated value conforms to the specification's type
- The specification is marked as optional and no corresponding field name is found in the record

A conforming value may contain field names not listed in the field specification list if and only if the record type is open.

Function Types

Any function value conforms to the primitive type `function`, which does not place any restrictions on the types of the function's formal parameters or the function's return value. A custom *function-type value* is used to place type restrictions on the signatures of conformant function values.

function-type:

`function` (*parameter-specification-list*_{opt}) *return-type*

parameter-specification-list:

required-parameter-specification-list

required-parameter-specification-list , *optional-parameter-specification-list*

optional-parameter-specification-list

required-parameter-specification-list:

required-parameter-specification

required-parameter-specification , *required-parameter-specification-list*

required-parameter-specification:

parameter-specification

optional-parameter-specification-list:

optional-parameter-specification

optional-parameter-specification , *optional-parameter-specification-list*

optional-parameter-specification:

`optional` *parameter-specification*

parameter-specification:

parameter-name *parameter-type*

parameter-type:

type-assertion

type-assertion:

`as` *type*

The result of evaluating a *function-type* is a type value whose base type is `function`.

The following examples illustrate the syntax for declaring function types:

Power Query M

```
type function (x as text) as number
type function (y as number, optional z as text) as any
```

A function value conforms to a function type if the return type of the function value is compatible with the function type's return type and each parameter specification of the function type is compatible to the positionally corresponding formal parameter of the function. A parameter specification is compatible with a formal parameter if the specified *parameter-type* type is compatible with the type of the formal parameter and the parameter specification is optional if the formal parameter is optional.

Formal parameter names are ignored for the purposes of determining function type conformance.

Specifying a parameter as optional implicitly makes its type nullable. The following create identical function types:

Power Query M

```
type function (optional x as text) as any
type function (optional x as nullable text) as any
```

Table types

A *table-type value* is used to define the structure of a table value.

table-type:

`table` *row-type*

row-type:

[*field-specification-list*_{opt}]

The result of evaluating a *table-type* is a type value whose base type is `table`.

The *row type* of a table specifies the column names and column types of the table as a closed record type. So that all table values conform to the type `table`, its row type is type `record` (the empty open record type). Thus, type `table` is abstract since no table value can have type `table`'s row type (but all table values have a row type that is compatible with type `table`'s row type). The following example shows the construction of a table type:

Power Query M

```
type table [A = text, B = number, C = binary]
// a table type with three columns named A, B, and C
// of column types text, number, and binary, respectively
```

A table-type value also carries the definition of a table value's *keys*. A key is a set of column names. At most one key can be designated as the table's *primary key*. (Within M, table keys have no semantic meaning. However, it is common for external data sources, such as databases or OData feeds, to define keys over tables. Power Query uses key information to improve performance of advanced functionality, such as cross-source join operations.)

The standard library functions `Type.TableKeys`, `Type.AddTableKey`, and `Type.ReplaceTableKeys` can be used to obtain the keys of a table type, add a key to a table type, and replace all keys of a table type, respectively.

Power Query M

```
Type.AddTableKey(tableType, {"A", "B"}, false)
// add a non-primary key that combines values from columns A and B
Type.ReplaceTableKeys(tableType, {})
// returns type value with all keys removed
```

Nullable types

For any type `T`, a nullable variant can be derived by using *nullable-type*:

nullable-type:

`nullable` *type*

The result is an abstract type that allows values of type `T` or the value `null`.

Power Query M

```
42 is nullable number      // true null is
nullable number            // true
```

Ascription of `type nullable T` reduces to ascription of `type null` or `type T`. (Recall that nullable types are abstract and no value can be directly of abstract type.)

Power Query M

```
Value.Type(42 as nullable number)    // type number
Value.Type(null as nullable number)  // type null
```

The standard library functions `Type.IsNullable` and `Type.NonNullable` can be used to test a type for nullability and to remove nullability from a type.

The following hold (for any `type T`):

- `type T` is compatible with `type nullable T`
- `Type.NonNullable(type T)` is compatible with `type T`

The following are pairwise equivalent (for any `type T`):

`type nullable any`

`any`

`Type.NonNullable(type any)`

`type anynonnull`

`type nullable none`

`type null`

`Type.NonNullable(type null)`

`type none`

`type nullable nullable T`

`type nullable T`

`Type.NonNullable(Type.NonNullable(type T))`

`Type.NonNullable(type T)`

`Type.NonNullable(type nullable T)`

`Type.NonNullable(type T)`


```
type nullable (Type.NonNullable(type T))
```

```
type nullable T
```

Ascribed type of a value

A value's *ascribed type* is the type to which a value is *declared* to conform.

A value may be ascribed a type using the library function `Value.ReplaceType`. This function either returns a new value with the type ascribed or raises an error if the new type is incompatible with the value.

When a value is ascribed a type, only a limited conformance check occurs:

- The type being ascribed must be non-abstract, non-nullable, and compatible with the value's intrinsic (native) *primitive-type*.
- When a custom type that defines structure is ascribed, it must match the structure of the value.
 - For records: The type must be closed, must define the same number of fields as the value, and must not contain any optional fields. (The type's field names and field types will replace those currently associated with the record. However, existing field values will not be checked against the new field types.)
 - For tables: The type must define the same number of columns as the value. (The type's column names and column types will replace those currently associated with the table. However, existing column values will not be checked against the new column types.)
 - For functions: The type must define the same number of required parameters, as well as the same number of optional parameters, as the value. (The type's parameter and return assertions, as well as its parameter names, will replace those associated with the function value's current type. However, the new assertions will have no effect on the actual behavior of the function.)
 - For lists: The value must be a list. (However, existing list items will not be checked against the new item type.)

Library functions may choose to compute and ascribe complex types to results based on the ascribed types of the input values.

The ascribed type of a value may be obtained using the library function `Value.Type`. For example:

```
Power Query M
```

```
Value.Type( Value.ReplaceType( {1}, type {number} )
```

```
// type {number}
```

Type equivalence and compatibility

Type equivalence is not defined in M. An M implementation may optionally choose to use its own rules to perform equality comparisons between type values. Comparing two type values for equality should evaluate to `true` if they are considered identical by the implementation, and `false` otherwise. In either case, the response returned must be consistent if the same two values are repeatedly compared. Note that within a given implementation, comparing some identical type values (such as `(type text) = (type text)`) may return `true`, while comparing others (such as `(type [a = text]) = (type [a = text])`) may not.

Compatibility between a given type and either a primitive type or a nullable primitive type can be determined using the library function `Type.Is`, which accepts an arbitrary type value as its first argument and a primitive or nullable primitive type value as its second argument:

Power Query M

```
Type.Is(type text, type nullable text) // true
Type.Is(type nullable text, type text) // false
Type.Is(type number, type text)       // false
Type.Is(type [a=any], type record)    // true
Type.Is(type [a=any], type list)      // false
```

There is no support in M for determining compatibility of a given type with a custom type.

The standard library does include a collection of functions to extract the defining characteristics from a custom type, so specific compatibility tests can be implemented as M expressions. Below are some examples; consult the M library specification for full details.

Power Query M

```
Type.ListItem( type {number} )
// type number
Type.NonNullable( type nullable text )
// type text
Type.RecordFields( type [A=text, B=time] )
// [ A = [Type = type text, Optional = false],
//   B = [Type = type time, Optional = false] ]
Type.TableRow( type table [X=number, Y=date] )
```

```
// type [X = number, Y = date]
Type.FunctionParameters(
    type function (x as number, optional y as text) as number)
// [ x = type number, y = type nullable text ]
Type.FunctionRequiredParameters(
    type function (x as number, optional y as text) as number)
// 1
Type.FunctionReturn(
    type function (x as number, optional y as text) as number)
// type number
```

Related content

- [Types and type conversion](#)
- [Types in the Power Query M formula language](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Operator behavior

Article • 02/13/2025

This section defines the behavior of the various M operators.

Operator precedence

When an expression contains multiple operators, the *precedence* of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the binary `+` operator. The precedence of an operator is established by the definition of its associated grammar production. For example, an *additive-expression* consists of a sequence of *multiplicative-expression*'s separated by `+` or `-` operators, thus giving the `+` and `-` operators lower precedence than the `*` and `/` operators.

The *parenthesized-expression* production can be used to change the default precedence ordering.

parenthesized-expression:

`( expression )`

For example:

Power Query M	
<code>1 + 2 * 3</code>	<code>// 7</code>
<code>(1 + 2) * 3</code>	<code>// 9</code>

The following table summarizes the M operators, listing the operator categories in order of precedence from highest to lowest. Operators in the same category have equal precedence.

[Expand table](#)

Category	Expression	Description
Primary	<code>i</code> <code>@i</code>	Identifier expression
	<code>(x)</code>	Parenthesized expression
	<code>x[i]</code>	Lookup

	$x\{y\}$	Item access
	$x(\dots)$	Function invocation
	$\{x, y, \dots\}$	List initialization
	$[i = x, \dots]$	Record initialization
	...	Not implemented
Unary	$+x$	Identity
	$-x$	Negation
	<code>not</code> x	Logical negation
Metadata	x <code>meta</code> y	Associate metadata
Multiplicative	$x * y$	Multiplication
	x / y	Division
Additive	$x + y$	Addition
	$x - y$	Subtraction
Relational	$x < y$	Less than
	$x > y$	Greater than
	$x \leq y$	Less than or equal
	$x \geq y$	Greater than or equal
Equality	$x = y$	Equal
	$x \neq y$	Not equal
Type assertion	x <code>as</code> y	Is compatible primitive/nullable primitive type or error
Type conformance	x <code>is</code> y	Test if type is compatible with primitive type or nullable primitive type
Logical AND	x <code>and</code> y	Short-circuiting conjunction
Logical OR	x <code>or</code> y	Short-circuiting disjunction
Coalesce	x <code>??</code> y	Null coalescing operator

Operators and metadata

Every value has an associated record value that can carry additional information about the value. This record is referred to as the *metadata record* for a value. A metadata record can be associated with any kind of value, even `null`. The result of such an association is a new value with the given metadata.

A metadata record is just a regular record and can contain any fields and values that a regular record can, and itself has a metadata record. Associating a metadata record with a value is "non-intrusive". It does not change the value's behavior in evaluations except for those that explicitly inspect metadata records.

Every value has a default metadata record, even if one has not been specified. The default metadata record is empty. The following examples show accessing the metadata record of a text value using the `Value.Metadata` standard library function:

Power Query M

```
Value.Metadata( "Mozart" ) // []
```

Metadata records are generally *not preserved* when a value is used with an operator or function that constructs a new value. For example, if two text values are concatenated using the `&` operator, the metadata of the resulting text value is the empty record `[]`. The following expressions are equivalent:

Power Query M

```
"Amadeus " & ("Mozart" meta [ Rating = 5 ])
"Amadeus " & "Mozart"
```

The standard library functions `Value.RemoveMetadata` and `Value.ReplaceMetadata` can be used to remove all metadata from a value and to replace a value's metadata (rather than merge metadata into possibly existing metadata).

The only operator that returns results that carry metadata is the [meta operator](#).

Structurally recursive operators

Values can be *cyclic*. For example:

Power Query M

```
let l = {0, @l} in l
// {0, {0, {0, ... }}}
```

```
[A={B}, B={A}]  
// [A = {{ ... }}, B = {{ ... }}]
```

M handles cyclic values by keeping construction of records, lists, and tables lazy. An attempt to construct a cyclic value that does not benefit from interjected lazy structured values yields an error:

Power Query M

```
[A=B, B=A]  
// [A = Error.Record("Expression.Error",  
//           "A cyclic reference was encountered during evaluation"),  
// B = Error.Record("Expression.Error",  
//           "A cyclic reference was encountered during evaluation"),  
// ]
```

Some operators in M are defined by structural recursion. For instance, equality of records and lists is defined by the conjoined equality of corresponding record fields and item lists, respectively.

For non-cyclic values, applying structural recursion yields a *finite expansion* of the value: shared nested values will be traversed repeatedly, but the process of recursion always terminates.

A cyclic value has an *infinite expansion* when applying structural recursion. The semantics of M makes no special accommodations for such infinite expansions—an attempt to compare cyclic values for equality, for instance, will typically run out of resources and terminate exceptionally.

Selection and Projection Operators

The selection and projection operators allow data to be extracted from list and record values.

Item Access

A value may be selected from a list or table based on its zero-based position within that list or table using an *item-access-expression*.

item-access-expression:

item-selection

optional-item-selection

item-selection:

primary-expression { *item-selector* }

optional-item-selection:

primary-expression { *item-selector* } ?

item-selector:

expression

The *item-access-expression* `x{y}` returns:

- For a list `x` and a number `y`, the item of list `x` at position `y`. The first item of a list is considered to have an ordinal index of zero. If the requested position does not exist in the list, an error is raised.
- For a table `x` and a number `y`, the row of table `x` at position `y`. The first row of a table is considered to have an ordinal index of zero. If the requested position does not exist in the table, an error is raised.
- For a table `x` and a record `y`, the row of table `x` that matches the field values of record `y` for fields with field names that match corresponding table-column names. If there is no unique matching row in the table, an error is raised.

For example:

Power Query M

```
{"a","b","c"}{0}           // "a"
{1, [A=2], 3}{1}          // [A=2]
{true, false}{2}          // error
#table({"A","B"},{{0,1},{2,1}}){0} // [A=0,B=1]
#table({"A","B"},{{0,1},{2,1}}){[A=2]} // [A=2,B=1]
#table({"A","B"},{{0,1},{2,1}}){[B=3]} // error
#table({"A","B"},{{0,1},{2,1}}){[B=1]} // error
```

The *item-access-expression* also supports the form `x{y}?`, which returns `null` when position (or match) `y` does not exist in list or table `x`. If there are multiple matches for `y`, an error is still raised.

For example:

Power Query M

```
{"a","b","c"}{0}?          // "a"
{1, [A=2], 3}{1}?          // [A=2]
{true, false}{2}?          // null
#table({"A","B"},{{0,1},{2,1}}){0}? // [A=0,B=1]
#table({"A","B"},{{0,1},{2,1}}){[A=2]}? // [A=2,B=1]
```



```
#table({"A","B"},{{0,1},{2,1}}){[B=3]}? // null
#table({"A","B"},{{0,1},{2,1}}){[B=1]}? // error
```

Item access does not force the evaluation of list or table items other than the one being accessed. For example:

Power Query M

```
{ error "a", 1, error "c" }{1} // 1
{ error "a", error "b" }{1} // error "b"
```

The following holds when the item access operator `x{y}` is evaluated:

- Errors raised during the evaluation of expressions `x` or `y` are propagated.
- The expression `x` produces a list or a table value.
- The expression `y` produces a number value or, if `x` produces a table value, a record value.
- If `y` produces a number value and the value of `y` is negative, an error with reason code `"Expression.Error"` is raised.
- If `y` produces a number value and the value of `y` is greater than or equal to the count of `x`, an error with reason code `"Expression.Error"` is raised unless the optional operator form `x{y}?` is used, in which case the value `null` is returned.
- If `x` produces a table value and `y` produces a record value and there are no matches for `y` in `x`, an error with reason code `"Expression.Error"` is raised unless the optional operator form `x{y}?` is used, in which case the value `null` is returned.
- If `x` produces a table value and `y` produces a record value and there are multiple matches for `y` in `x`, an error with reason code `"Expression.Error"` is raised.

No items in `x` other than that at position `y` is evaluated during the process of item selection. (For streaming lists or tables, the items or rows preceding that at position `y` are skipped over, which may cause their evaluation, depending on the source of the list or table.)

Field Access

The *field-access-expression* is used to *select* a value from a record or to *project* a record or table to one with fewer fields or columns, respectively.

field-access-expression:

field-selection

implicit-target-field-selection

projection

implicit-target-projection

field-selection:

primary-expression field-selector

field-selector:

required-field-selector

optional-field-selector

required-field-selector:

[*field-name*]

optional-field-selector:

[*field-name*] ?

field-name:

generalized-identifier

quoted-identifier

implicit-target-field-selection:

field-selector

projection:

primary-expression required-projection

primary-expression optional-projection

required-projection:

[*required-selector-list*]

optional-projection:

[*required-selector-list*] ?

required-selector-list:

required-field-selector

required-selector-list , *required-field-selector*

implicit-target-projection:

required-projection

optional-projection

The simplest form of field access is *required field selection*. It uses the operator `x[y]` to look up a field in a record by field name. If the field `y` does not exist in `x`, an error is raised. The form `x[y]?` is used to perform *optional* field selection, and returns `null` if the requested field does not exist in the record.

For example:

```
[A=1,B=2][B]          // 2
[A=1,B=2][C]          // error
[A=1,B=2][C]?         // null
```

Collective access of multiple fields is supported by the operators for *required record projection* and *optional record projection*. The operator `x[[y1],[y2],...]` projects the record to a new record with fewer fields (selected by `y1, y2, ...`). If a selected field does not exist, an error is raised. The operator `x[[y1],[y2],...]?` projects the record to a new record with the fields selected by `y1, y2, ...`; if a field is missing, `null` is used instead. For example:

Power Query M

```
[A=1,B=2][[B]]          // [B=2]
[A=1,B=2][[C]]          // error
[A=1,B=2][[B],[C]]?     // [B=2,C=null]
```

The forms `[y]` and `[y]?` are supported as a *shorthand* reference to the identifier `_` (underscore). The following two expressions are equivalent:

Power Query M

```
[A]
_[A]
```

The following example illustrates the shorthand form of field access:

Power Query M

```
let _ = [A=1,B=2] in [A] //1
```

The form `[[y1],[y2],...]` and `[[y1],[y2],...]?` are also supported as a shorthand and the following two expressions are likewise equivalent:

Power Query M

```
[[A],[B]]
_[[A],[B]]
```

The shorthand form is particularly useful in combination with the `each` shorthand, a way to introduce a function of a single parameter named `_` (for details, see [Simplified](#)

[declarations](#)). Together, the two shorthands simplify common higher-order functional expressions:

Power Query M

```
List.Select( {[a=1, b=1], [a=2, b=4]}, each [a] = [b])  
// {[a=1, b=1]}
```

The above expression is equivalent to the following more cryptic looking longhand:

Power Query M

```
List.Select( {[a=1, b=1], [a=2, b=4]}, (_) => _[a] = _[b])  
// {[a=1, b=1]}
```

Field access does not force the evaluation of fields other than the one(s) being accessed. For example:

Power Query M

```
[A=error "a", B=1, C=error "c"][B] // 1  
[A=error "a", B=error "b"][B] // error "b"
```

The following holds when a field access operator `x[y]`, `x[y]?`, `x[[y]]`, or `x[[y]]?` is evaluated:

- Errors raised during the evaluation of expression `x` are propagated.
- Errors raised when evaluating field `y` are permanently associated with field `y`, then propagated. Any future access to field `y` will raise the identical error.
- The expression `x` produces a record or table value, or an error is raised.
- If the identifier `y` names a field that does not exist in `x`, an error with reason code `"Expression.Error"` is raised unless the optional operator form `...?` is used, in which case the value `null` is returned.

No fields of `x` other than that named by `y` is evaluated during the process of field access.

Metadata operator

The metadata record for a value is amended using the *meta operator* (`x meta y`).

metadata-expression:

unary-expression

unary-expression `meta` *unary-expression*

The following example constructs a text value with a metadata record using the `meta` operator and then accesses the metadata record of the resulting value using

`Value.Metadata`:

Power Query M

```
Value.Metadata( "Mozart" meta [ Rating = 5 ] )  
// [Rating = 5 ]  
Value.Metadata( "Mozart" meta [ Rating = 5 ] )[Rating]  
// 5
```

The following holds when applying the metadata combining operator `x meta y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- The `y` expression must be a record, or an error with reason code `"Expression.Error"` is raised.
- The resulting metadata record is `x`'s metadata record merged with `y`. (For the semantics of record merge, see [Record merge](#).)
- The resulting value is the value from the `x` expression, without its metadata, with the newly computed metadata record attached.

The standard library functions `Value.RemoveMetadata` and `Value.ReplaceMetadata` can be used to remove all metadata from a value and to replace a value's metadata (rather than merge metadata into possibly existing metadata). The following expressions are equivalent:

Power Query M

```
x meta y  
Value.ReplaceMetadata(x, Value.Metadata(x) & y)  
Value.RemoveMetadata(x) meta (Value.Metadata(x) & y)
```

Equality operators

The *equality operator* `=` is used to determine if two values are equal. The *inequality operator* `<>` is used to determine if two values are not equal.

equality-expression:

relational-expression

relational-expression `=` *equality-expression*

relational-expression `<>` *equality-expression*

For example:

Power Query M

```
1 = 1           // true
1 = 2           // false
1 <> 1          // false
1 <> 2          // true
null = true     // false
null = null     // true
```

Metadata is not part of equality or inequality comparison. For example:

Power Query M

```
(1 meta [ a = 1 ]) = (1 meta [ a = 2 ]) // true
(1 meta [ a = 1 ]) = 1                  // true
```

The following holds when applying the equality operators `x = y` and `x <> y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- The `=` operator has a result of `true` if the values are equal, and `false` otherwise.
- The `<>` operator has a result of `false` if the values are equal, and `true` otherwise.
- Metadata records are not included in the comparison.
- If values produced by evaluating the `x` and `y` expressions are not the same kind of value, then the values are not equal.
- If the values produced by evaluating the `x` and `y` expression are the same kind of value, then there are specific rules for determining if they are equal, as defined below.
- The following is always true:

Power Query M

```
(x = y) = not (x <> y)
```

The equality operators are defined for the following types:

- The `null` value is only equal to itself.

Power Query M

```
null = null    // true
null = true    // false
null = false   // false
```

- The logical values `true` and `false` are only equal to themselves. For example:

Power Query M

```
true = true    // true
false = false   // true
true = false    // false
true = 1        // false
```

- Numbers are compared using the specified precision:
 - If either number is `#nan`, then the numbers are not the same.
 - When neither number is `#nan`, then the numbers are compared using a bit-wise comparison of the numeric value.
 - `#nan` is the only value that is not equal to itself.

For example:

Power Query M

```
1 = 1,          // true
1.0 = 1         // true
2 = 1           // false
#nan = #nan     // false
#nan <> #nan     // true
```

- Two durations are equal if they represent the same number of 100-nanosecond ticks.
- Two times are equal if the magnitudes of their parts (hour, minute, second) are equal.
- Two dates are equal if the magnitudes of their parts (year, month, day) are equal.

- Two datetimes are equal if the magnitudes of their parts (year, month, day, hour, minute, second) are equal.
- Two datetimezones are equal if the corresponding UTC datetimes are equal. To arrive at the corresponding UTC datetime, the hours/minutes offset is subtracted from the datetime component of the datetimezone.
- Two text values are equal if using an ordinal, case-sensitive, culture-insensitive comparison they have the same length and equal characters at corresponding positions.
- Two list values are equal if all of the following are true:
 - Both lists contain the same number of items.
 - The values of each positionally corresponding item in the lists are equal. This means that not only do the lists need to contain equal items, the items need to be in the same order.

For example:

Power Query M

```
{1, 2} = {1, 2}      // true
{2, 1} = {1, 2}      // false
{1, 2, 3} = {1, 2}   // false
```

- Two records are equal if all of the following are true:
 - The number of fields is the same.
 - Each field name of one record is also present in the other record.
 - The value of each field of one record is equal to the like-named field in the other record.

For example:

Power Query M

```
[ A = 1, B = 2 ] = [ A = 1, B = 2 ]      // true
[ B = 2, A = 1 ] = [ A = 1, B = 2 ]      // true
[ A = 1, B = 2, C = 3 ] = [ A = 1, B = 2 ] // false
[ A = 1 ] = [ A = 1, B = 2 ]             // false
```

- Two tables are equal if all of the following are true:

- The number of columns is the same.
- Each column name in one table is also present in the other table.
- The number of rows is the same.
- Each row has equal values in corresponding cells.

For example:

Power Query M

```
#table({"A","B"},{{1,2}}) = #table({"A","B"},{{1,2}}) // true
#table({"A","B"},{{1,2}}) = #table({"X","Y"},{{1,2}}) // false
#table({"A","B"},{{1,2}}) = #table({"B","A"},{{2,1}}) // true
```

- A function value is equal to itself, but may or may not be equal to another function value. If two function values are considered equal, then they will behave identically when invoked.

Two given function values will always have the same equality relationship.

- A type value is equal to itself, but may or may not be equal to another type value. If two type values are considered equal, then they will behave identically when queried for conformance.

Two given type values will always have the same equality relationship.

Relational operators

The `<`, `>`, `<=`, and `>=` operators are called the *relational operators*.

relational-expression:

additive-expression

additive-expression `<` *relational-expression*

additive-expression `>` *relational-expression*

additive-expression `<=` *relational-expression*

additive-expression `>=` *relational-expression*

These operators are used to determine the relative ordering relationship between two values, as shown in the following table:

Operation	Result
<code>x < y</code>	true if <code>x</code> is less than <code>y</code> , false otherwise
<code>x > y</code>	true if <code>x</code> is greater than <code>y</code> , false otherwise
<code>x <= y</code>	true if <code>x</code> is less than or equal to <code>y</code> , false otherwise
<code>x >= y</code>	true if <code>x</code> is greater than or equal to <code>y</code> , false otherwise

For example:

Power Query M	
<code>0 <= 1</code>	<code>// true</code>
<code>null < 1</code>	<code>// null</code>
<code>null <= null</code>	<code>// null</code>
<code>"ab" < "abc"</code>	<code>// true</code>
<code>#nan >= #nan</code>	<code>// false</code>
<code>#nan <= #nan</code>	<code>// false</code>

The following holds when evaluating an expression containing the relational operators:

- Errors raised when evaluating the `x` or `y` operand expressions are propagated.
- The values produced by evaluating both the `x` and `y` expressions must be a binary, date, datetime, datetimezone, duration, logical, number, null, text or time value. Otherwise, an error with reason code `"Expression.Error"` is raised.
- Both operands must be the same kind of value or `null`. Otherwise, an error with reason code `"Expression.Error"` is raised.
- If either or both operands are `null`, the result is the `null` value.
- Two binaries are compared byte by byte.
- Two dates are compared by comparing their year parts and, if equal, their month parts and, if equal, their day parts.
- Two datetimes are compared by comparing their year parts and, if equal, their month parts and, if equal, their day parts and, if equal, their hour parts and, if equal, their minute parts and, if equal, their second parts.
- Two datetimezones are compared by normalizing them to UTC by subtracting their hour/minute offset and then comparing their datetime components.

- Two durations are compared according to the total number of 100-nanosecond ticks they represent.
- Two logicals are compared such that `true` is considered to be greater than `false`.
- Two numbers `x` and `y` are compared according to the rules of the IEEE 754 standard:
 - If either operand is `#nan`, the result is `false` for all relational operators.
 - When neither operand is `#nan`, the operators compare the values of the two floatingpoint operands with respect to the ordering `-∞ < -max < ... < -min < -0.0 = +0.0 < +min < ... < +max < +∞` where min and max are the smallest and largest positive finite values that can be represented. The M names for `-∞` and `+∞` are `-#infinity` and `#infinity`.

Notable effects of this ordering are:

- Negative and positive zeros are considered equal.
- A `-#infinity` value is considered less than all other number values, but equal to another `-#infinity`.
- A `#infinity` value is considered greater than all other number values, but equal to another `#infinity`.
- Two texts are compared by using a character-by-character ordinal, case-sensitive, culture-insensitive comparison.
- Two times are compared by comparing their hour parts and, if equal, their minute parts and, if equal, their second parts.

Conditional logical operators

The `and` and `or` operators are called the conditional logical operators.

logical-or-expression:

logical-and-expression

logical-and-expression `or` *logical-or-expression*

logical-and-expression:

is-expression

is-expression `and` *logical-and-expression*

The `or` operator returns `true` when at least one of its operands is `true`. The right operand is evaluated if and only if the left operand is not `true`.

The `and` operator returns `false` when at least one of its operands is `false`. The right operand is evaluated if and only if the left operand is not `false`.

Truth tables for the `or` and `and` operators are shown below, with the result of evaluating the left operand expression on the vertical axis and the result of evaluating the right operand expression on the horizontal axis.

[Expand table](#)

<i>and</i>	true	false	null	error
true	true	false	null	error
false	false	false	false	false
null	null	false	null	error
error	error	error	error	error

[Expand table](#)

<i>or</i>	true	false	null	error
or	true	false	null	error
true	true	true	true	true
false	true	false	null	error
null	true	null	null	error
error	error	error	error	error

The following holds when evaluating an expression containing conditional logical operators:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- The conditional logical operators are defined over the types `logical` and `null`. If the operand values are not of those types, an error with reason code `"Expression.Error"` is raised.
- The result is a logical value.

- In the expression `x` or `y`, the expression `y` will be evaluated if and only if `x` does not evaluate to `true`.
- In the expression `x` and `y`, the expression `y` will be evaluated if and only if `x` does not evaluate to `false`.

The last two properties give the conditional logical operators their "conditional" qualification; properties also referred to as "short-circuiting". These properties are useful to write compact *guarded predicates*. For example, the following expressions are equivalent:

Power Query M

```
d <> 0 and n/d > 1 if d <> 0 then n/d > 1 else false
```

Arithmetic Operators

The `+`, `-`, `*` and `/` operators are the *arithmetic operators*.

additive-expression:

multiplicative-expression

additive-expression `+` *multiplicative-expression*

additive-expression `-` *multiplicative-expression*

multiplicative-expression:

metadata-expression

multiplicative-expression `*` *metadata-expression*

multiplicative-expression `/` *metadata-expression*

Precision

Numbers in M are stored using a variety of representations to retain as much information as possible about numbers coming from a variety of sources. Numbers are only converted from one representation to another as needed by operators applied to them. Two precisions are supported in M:

[Expand table](#)

Precision	Semantics
<code>Precision.Decimal</code>	128-bit decimal representation with a range of $\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$ and 28-29 significant digits.

Precision	Semantics
<code>Precision.Double</code>	Scientific representation using mantissa and exponent; conforms to the 64-bit binary double-precision IEEE 754 arithmetic standard IEEE 754-2008 ↗ .

Arithmetic operations are performed by choosing a precision, converting both operands to that precision (if necessary), then performing the actual operation, and finally returning a number in the chosen precision.

The built-in arithmetic operators (`+`, `-`, `*`, `/`) use Double Precision. Standard library functions (`Value.Add`, `Value.Subtract`, `Value.Multiply`, `Value.Divide`) can be used to request these operations using a specific precision model.

- No numeric overflow is possible: `#infinity` or `-#infinity` represent values of magnitudes too large to be represented.
- No numeric underflow is possible: `0` and `-0` represent values of magnitudes too small to be represented.
- The IEEE 754 special value `#nan` (NaN—Not a Number) is used to cover arithmetically invalid cases, such as a division of zero by zero.
- Conversion from Decimal to Double precision is performed by rounding decimal numbers to the nearest equivalent double value.
- Conversion from Double to Decimal precision is performed by rounding double numbers to the nearest equivalent decimal value and, if necessary, overflowing to `#infinity` or `-#infinity` values.

Addition operator

The interpretation of the addition operator (`x + y`) is dependent on the kind of value of the evaluated expressions `x` and `y`, as follows:

[↗](#) Expand table

x	y	Result	Interpretation
<code>type number</code>	<code>type number</code>	<code>type number</code>	Numeric sum
<code>type number</code>	<code>null</code>	<code>null</code>	
<code>null</code>	<code>type number</code>	<code>null</code>	
<code>type duration</code>	<code>type duration</code>	<code>type duration</code>	Numeric sum of magnitudes

x	y	Result	Interpretation
type duration	null	null	
null	type duration	null	
type datetime	type duration	type datetime	Datetime offset by duration
type duration	type datetime	type datetime	
type datetime	null	null	
null	type datetime	null	

In the table, `type datetime` stands for any of `type date`, `type datetime`, `type datetimezone`, or `type time`. When adding a duration and a value of some type *datetime*, the resulting value is of that same type.

For other combinations of values than those listed in the table, an error with reason code `"Expression.Error"` is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

Numeric sum

The sum of two numbers is computed using the *addition operator*, producing a number.

For example:

Power Query M

```
1 + 1           // 2
#nan + #infinity // #nan
```

The addition operator `+` over numbers uses Double Precision; the standard library function `Value.Add` can be used to specify Decimal Precision. The following holds when computing a sum of numbers:

- The sum in Double Precision is computed according to the rules of 64-bit binary doubleprecision IEEE 754 arithmetic [IEEE 754-2008](#). The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x + y`. If `x` and `y` have the same magnitude but opposite signs, `z` is positive zero. If `x +`

`y` is too large to be represented in the destination type, `z` is an infinity with the same sign as `x + y`.

[Expand table](#)

+	y	+0	-0	+∞	-∞	NaN
x	z	x	x	+∞	-∞	NaN
+0	y	+0	+0	+∞	-∞	NaN
-0	y	+0	-0	+∞	-∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN
-∞	-∞	-∞	-∞	NaN	-∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The sum in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

Sum of durations

The sum of two durations is the duration representing the sum of the number of 100nanosecond ticks represented by the durations. For example:

Power Query M

```
#duration(2,1,0,15.1) + #duration(0,1,30,45.3)
// #duration(2, 2, 31, 0.4)
```

Datetime offset by duration

A *datetime* `x` and a duration `y` may be added using `x + y` to compute a new *datetime* whose distance from `x` on a linear timeline is exactly the magnitude of `y`. Here, *datetime* stands for any of `Date`, `DateTime`, `DateTimeZone`, or `Time` and a non-null result will be of the same type. The datetime offset by duration may be computed as follows:

- If the datetime's days since epoch value is specified, construct a new datetime with the following information elements:
 - Calculate a new days since epoch equivalent to dividing the magnitude of `y` by the number of 100-nanosecond ticks in a 24-hour period, truncating the

decimal portion of the result, and adding this value to the x's days since epoch.

- Calculate a new ticks since midnight equivalent to adding the magnitude of y to the x's ticks since midnight, modulo the number of 100-nanosecond ticks in a 24-hour period. If x does not specify a value for ticks since midnight, a value of 0 is assumed.
- Copy x's value for minutes offset from UTC unchanged.
- If the datetime's days since epoch value is unspecified, construct a new datetime with the following information elements specified:
 - Calculate a new ticks since midnight equivalent to adding the magnitude of y to the x's ticks since midnight, modulo the number of 100-nanosecond ticks in a 24-hour period. If x does not specify a value for ticks since midnight, a value of 0 is assumed.
 - Copy x's values for days since epoch and minutes offset from UTC unchanged.

The following examples show calculating the absolute temporal sum when the datetime specifies the *days since epoch*:

Power Query M

```
#date(2010,05,20) + #duration(0,8,0,0)
// #datetime( 2010, 5, 20, 8, 0, 0 )
// 2010-05-20T08:00:00

#date(2010,01,31) + #duration(30,08,0,0)
// #datetime(2010, 3, 2, 8, 0, 0)
// 2010-03-02T08:00:00

#datetime(2010,05,20,12,00,00,-08) + #duration(0,04,30,00)
// #datetime(2010, 5, 20, 16, 30, 0, -8, 0)
// 2010-05-20T16:30:00-08:00

#datetime(2010,10,10,0,0,0,0) + #duration(1,0,0,0)
// #datetime(2010, 10, 11, 0, 0, 0, 0, 0)
// 2010-10-11T00:00:00+00:00
```

The following example shows calculating the datetime offset by duration for a given time:

Power Query M

```
#time(8,0,0) + #duration(30,5,0,0)
// #time(13, 0, 0)
```

Subtraction operator

The interpretation of the subtraction operator ($x - y$) is dependent on the kind of the value of the evaluated expressions x and y , as follows:

[Expand table](#)

x	Y	Result	Interpretation
type number	type number	type number	Numeric difference
type number	null	null	
null	type number	null	
type duration	type duration	type duration	Numeric difference of magnitudes
type duration	null	null	
null	type duration	null	
type <i>datetime</i>	type <i>datetime</i>	type duration	Duration between datetimes
type <i>datetime</i>	type duration	type <i>datetime</i>	Datetime offset by negated duration
type <i>datetime</i>	null	null	
null	type <i>datetime</i>	null	

In the table, type *datetime* stands for any of type date, type datetime, type datetimezone, or type time. When subtracting a duration from a value of some type *datetime*, the resulting value is of that same type.

For other combinations of values than those listed in the table, an error with reason code "Expression.Error" is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

Numeric difference

The difference between two numbers is computed using the *subtraction operator*, producing a number. For example:

```
1 - 1           // 0
#nan - #infinity // #nan
```

The subtraction operator `-` over numbers uses Double Precision; the standard library function `Value.Subtract` can be used to specify Decimal Precision. The following holds when computing a difference of numbers:

- The difference in Double Precision is computed according to the rules of 64-bit binary double-precision IEEE 754 arithmetic [IEEE 754-2008](#). The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x - y`. If `x` and `y` are equal, `z` is positive zero. If `x - y` is too large to be represented in the destination type, `z` is an infinity with the same sign as `x - y`.

[Expand table](#)

-	y	+0	-0	+∞	-∞	NaN
x	z	x	x	-∞	+∞	NaN
+0	-y	+0	+0	-∞	+∞	NaN
-0	-y	-0	+0	-∞	+∞	NaN
+∞	+∞	+∞	+∞	NaN	+∞	NaN
-∞	-∞	-∞	-∞	-∞	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The difference in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

Difference of durations

The difference of two durations is the duration representing the difference between the number of 100-nanosecond ticks represented by each duration. For example:

```
#duration(1,2,30,0) - #duration(0,0,0,30.45)
// #duration(1, 2, 29, 29.55)
```

Datetime offset by negated duration

A *datetime* `x` and a duration `y` may be subtracted using `x - y` to compute a new *datetime*. Here, *datetime* stands for any of `date`, `datetime`, `datetimezone`, or `time`. The resulting *datetime* has a distance from `x` on a linear timeline that is exactly the magnitude of `y`, in the direction opposite the sign of `y`. Subtracting positive durations yields results that are backwards in time relative to `x`, while subtracting negative values yields results that are forwards in time.

Power Query M

```
#date(2010,05,20) - #duration(00,08,00,00)
// #datetime(2010, 5, 19, 16, 0, 0)
// 2010-05-19T16:00:00
#date(2010,01,31) - #duration( 30,08,00,00)
// #datetime(2009, 12, 31, 16, 0, 0)
// 2009-12-31T16:00:00
```

Duration between two datetimes

Two *datetimes* `t` and `u` may be subtracted using `t - u` to compute the duration between them. Here, *datetime* stands for any of `date`, `datetime`, `datetimezone`, or `time`. The duration produced by subtracting `u` from `t` must yield `t` when added to `u`.

Power Query M

```
#date(2010,01,31) - #date(2010,01,15)
// #duration(16,00,00,00)
// 16.00:00:00

#date(2010,01,15) - #date(2010,01,31)
// #duration(-16,00,00,00)
// -16.00:00:00

#datetime(2010,05,20,16,06,00,-08,00) -
#datetime(2008,12,15,04,19,19,03,00)
// #duration(521,22,46,41)
// 521.22:46:41
```

Subtracting `t - u` when `u > t` results in a negative duration:

Power Query M

```
#time(01,30,00) - #time(08,00,00)
// #duration(0, -6, -30, 0)
```

The following holds when subtracting two *datetimes* using `t - u`:

- $u + (t - u) = t$

Multiplication operator

The interpretation of the multiplication operator (`x * y`) is dependent on the kind of value of the evaluated expressions `x` and `y`, as follows:

[Expand table](#)

X	Y	Result	Interpretation
type number	type number	type number	Numeric product
type number	null	null	
null	type number	null	
type duration	type number	type duration	Multiple of duration
type number	type duration	type duration	Multiple of duration
type duration	null	null	
null	type duration	null	

For other combinations of values than those listed in the table, an error with reason code `"Expression.Error"` is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

Numeric product

The product of two numbers is computed using the *multiplication operator*, producing a number. For example:

Power Query M

```
2 * 4           // 8
6 * null        // null
#nan * #infinity // #nan
```

The multiplication operator `*` over numbers uses Double Precision; the standard library function `Value.Multiply` can be used to specify Decimal Precision. The following holds when computing a product of numbers:

- The product in Double Precision is computed according to the rules of 64-bit binary double-precision IEEE 754 arithmetic [IEEE 754-2008](#). The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x * y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

[Expand table](#)

*	+y	-y	+0	-0	+∞	-∞	NaN
+x	+z	-z	+0	-0	+∞	-∞	NaN
-x	-z	+z	-0	+0	-∞	+∞	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+∞	+∞	-∞	NaN	NaN	+∞	-∞	NaN
-∞	-∞	+∞	NaN	NaN	-∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The product in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

Multiples of durations

The product of a duration and a number is the duration representing the number of 100nanosecond ticks represented by the duration operand times the number operand. For example:

Power Query M

```
#duration(2,1,0,15.1) * 2
// #duration(4, 2, 0, 30.2)
```

Division operator

The interpretation of the division operator (`x / y`) is dependent on the kind of value of the evaluated expressions `x` and `y`, as follows:

[Expand table](#)

X	Y	Result	Interpretation
type number	type number	type number	Numeric quotient
type number	null	null	
null	type number	null	
type duration	type number	type duration	Fraction of duration
type duration	type duration	type number	Numeric quotient of durations
type duration	null	null	
null	type duration	null	

For other combinations of values than those listed in the table, an error with reason code `"Expression.Error"` is raised. Each combination is covered in the following sections.

Errors raised when evaluating either operand are propagated.

Numeric quotient

The quotient of two numbers is computed using the *division operator*, producing a number. For example:

Power Query M

```
8 / 2           // 4
8 / 0           // #infinity
0 / 0           // #nan
0 / null        // null
#nan / #infinity // #nan
```

The division operator `/` over numbers uses Double Precision; the standard library function `Value.Divide` can be used to specify Decimal Precision. The following holds when computing a quotient of numbers:

- The quotient in Double Precision is computed according to the rules of 64-bit binary double-precision IEEE 754 arithmetic [IEEE 754-2008](#). The following table

lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x / y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

[Expand table](#)

/	+y	-y	+0	-0	+∞	-∞	NaN
+x	+z	-z	+∞	-∞	+0	-0	NaN
-x	-z	+z	-∞	+∞	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+∞	+∞	-∞	+∞	-∞	NaN	NaN	NaN
-∞	-∞	+∞	-∞	+∞	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The sum in Decimal Precision is computed without losing precision. The scale of the result is the larger of the scales of the two operands.

Quotient of durations

The quotient of two durations is the number representing the quotient of the number of 100nanosecond ticks represented by the durations. For example:

Power Query M

```
#duration(2,0,0,0) / #duration(0,1,30,0)
// 32
```

Scaled durations

The quotient of a duration `x` and a number `y` is the duration representing the quotient of the number of 100-nanosecond ticks represented by the duration `x` and the number `y`. For example:

Power Query M


```
#duration(2,0,0,0) / 32
// #duration(0,1,30,0)
```

Structure Combination

The combination operator (`x & y`) is defined over the following kinds of values:

[Expand table](#)

X	Y	Result	Interpretation
type text	type text	type text	Concatenation
type text	null	null	
null	type text	null	
type date	type time	type datetime	Merge
type date	null	null	
null	type time	null	
type list	type list	type list	Concatenation
type record	type record	type record	Merge
type table	type table	type table	Concatenation

Concatenation

Two text, two list, or two table values can be concatenated using `x & y`.

The following example illustrates concatenating text values:

Power Query M

```
"AB" & "CDE" // "ABCDE"
```

The following example illustrates concatenating lists:

Power Query M

```
{1, 2} & {3} // {1, 2, 3}
```

The following holds when concatenating two values using `x & y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- No error is propagated if an item of either `x` or `y` contains an error.
- The result of concatenating two text values is a text value that contains the value of `x` immediately followed by `y`. If either of the operands is null and the other is a text value, the result is null.
- The result of concatenating two lists is a list that contains all the items of `x` followed by all the items of `y`.
- The result of concatenating two tables is a table that has the union of the two operand table's columns. The column ordering of `x` is preserved, followed by the columns only appearing in `y`, preserving their relative ordering. For columns appearing only in one of the operands, `null` is used to fill in cell values for the other operand.

Merge

Record merge

Two records can be merged using `x & y`, producing a record that includes fields from both `x` and `y`.

The following examples illustrate merging records:

Power Query M

```
[ x = 1 ] & [ y = 2 ] // [ x = 1, y = 2 ]  
[ x = 1, y = 2 ] & [ x = 3, z = 4 ] // [ x = 3, y = 2, z = 4 ]
```

The following holds when merging two records using `x + y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- If a field appears in both `x` and `y`, the value from `y` is used.
- The order of the fields in the resulting record is that of `x`, followed by fields in `y` that are not part of `x`, in the same order that they appear in `y`.
- Merging records does not cause evaluation of the values.

- No error is raised because a field contains an error.
- The result is a record.

Date-time merge

A date `x` can be merged with a time `y` using `x & y`, producing a datetime that combines the parts from both `x` and `y`.

The following example illustrates merging a date and a time:

Power Query M

```
#date(2013,02,26) & #time(09,17,00)
// #datetime(2013,02,26,09,17,00)
```

The following holds when merging two records using `x + y`:

- Errors raised when evaluating the `x` or `y` expressions are propagated.
- The result is a datetime.

Unary operators

The `+`, `-`, and `not` operators are unary operators.

unary-expression:

type-expression

`+` *unary expression*

`-` *unary expression*

`not` *unary expression*

Unary plus operator

The unary plus operator (`+x`) is defined for the following kinds of values:

[Expand table](#)

X	Result	Interpretation
<code>type number</code>	<code>type number</code>	Unary plus
<code>type duration</code>	<code>type duration</code>	Unary plus

X	Result	Interpretation
null	`null	

For other values, an error with reason code `"Expression.Error"` is raised.

The unary plus operator allows a `+` sign to be applied to a number, datetime, or null value. The result is that same value. For example:

Power Query M	
<code>+ - 1</code>	<code>// -1</code>
<code>+ + 1</code>	<code>// 1</code>
<code>+ #nan</code>	<code>// #nan</code>
<code>+ #duration(0,1,30,0)</code>	<code>// #duration(0,1,30,0)</code>

The following holds when evaluating the unary plus operator `+x`:

- Errors raised when evaluating `x` are propagated.
- If the result of evaluating `x` is not a number value, then an error with reason code `"Expression.Error"` is raised.

Unary minus operator

The unary minus operator (`-x`) is defined for the following kinds of values:

[Expand table](#)

X	Result	Interpretation
type number	type number	Negation
type duration	type duration	Negation
null	null	

For other values, an error with reason code `"Expression.Error"` is raised.

The unary minus operator is used to change the sign of a number or duration. For example:

Power Query M	
<code>- (1 + 1)</code>	<code>// -2</code>
<code>- - 1</code>	<code>// 1</code>

```

- - - 1          // -1
- #nan           // #nan
- #infinity      // -#infinity
- #duration(1,0,0,0) // #duration(-1,0,0,0)
- #duration(0,1,30,0) // #duration(0,-1,-30,0)

```

The following holds when evaluating the unary minus operator `-x`:

- Errors raised when evaluating `x` are propagated.
- If the expression is a number, then the result is the number value from expression `x` with its sign changed. If the value is NaN, then the result is also NaN.

Logical negation operator

The logical negation operator (`not`) is defined for the following kinds of values:

[Expand table](#)

X	Result	Interpretation
<code>type logical</code>	<code>type logical</code>	Negation
<code>null</code>	<code>null</code>	

This operator computes the logical `not` operation on a given logical value. For example:

Power Query M

```

not true          // false
not false         // true
not (true and true) // false

```

The following holds when evaluating the logical negation operator `not x`:

- Errors raised when evaluating `x` are propagated.
- The value produced from evaluating expression `x` must be a logical value, or an error with reason code `"Expression.Error"` must be raised. If the value is `true`, the result is `false`. If the operand is `false`, the result is `true`.

The result is a logical value.

Type operators

The operators `is` and `as` are known as the type operators.

Type compatibility operator

The type compatibility operator `x is y` is defined for the following types of values:

[Expand table](#)

X	Y	Result
<code>type any</code>	<i>primitive-or-nullable-primitive-type</i>	<code>type logical</code>

The expression `x is y` returns `true` if the ascribed type of `x` is compatible with `y`, and returns `false` if it is not compatible. `y` must be a primitive type or a nullable primitive type.

is-expression:

as-expression

is-expression `is` *primitive-or-nullable-primitive-type*

primitive-or-nullable-primitive-type:

`nullable`_{opt} *primitive-type*

Type compatibility, as supported by the `is` operator, is a subset of [general type compatibility](#) and is defined using the following rules:

- If `x` is null then it is compatible if `y` is the type `any`, the type `null`, or a nullable primitive type.
- If `x` is non-null then it is compatible if the primitive type of `x` is the same as `y`.

The following holds when evaluating the expression `x is y`:

- An error raised when evaluating expression `x` is propagated.

Type assertion operator

The type assertion operator `x as y` is defined for the following types of values:

[Expand table](#)

X	Y	Result
<code>type any</code>	<i>primitive-or-nullable-primitive-type</i>	<code>type any</code>

The expression `x as y` asserts that the value `x` is compatible with `y` as per the `is` operator. If it is not compatible, an error is raised. `y` must be a primitive type or a nullable primitive type.

as-expression:

equality-expression

as-expression as primitive-or-nullable-primitive-type

primitive-or-nullable-primitive-type:

`nullableopt primitive-type`

The expression `x as y` is evaluated as follows:

- A type compatibility check `x is y` is performed and the assertion returns `x` unchanged if that test succeeds.
- If the compatibility check fails, an error with reason code `"Expression.Error"` is raised.

Examples:

Power Query M

```
1 as number           // 1
"A" as number         // error
null as nullable number // null
```

The following holds when evaluating the expression `x as y`:

- An error raised when evaluating expression `x` is propagated.

Coalesce operator

The coalesce operator `??` returns the result of its left operand if it is not null, otherwise it will return the result of its right operand. The right operand is evaluated if and only if the left operand is null.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Let

Article • 10/10/2022

Let expression

A let expression can be used to capture a value from an intermediate calculation in a variable.

let-expression:

```
let variable-list in expression
```

variable-list:

variable

variable , *variable-list*

variable:

variable-name = *expression*

variable-name:

identifier

The following example shows intermediate results being calculated and stored in variables `x`, `y`, and `z` which are then used in a subsequent calculation `x + y + z`:

Power Query M

```
let    x = 1 + 1,
      y = 2 + 2,
      z = y + 1
in
      x + y + z
```

The result of this expression is:

Power Query M

```
11 // (1 + 1) + (2 + 2) + (2 + 2 + 1)
```

The following holds when evaluating expressions within the *let-expression*:

- The expressions in the variable list define a new scope containing the identifiers from the *variable-list* production and must be present when evaluating the expressions within the *variable-list* productions. Expressions within the *variable-list* may refer to one-another.

- The expressions within the *variable-list* must be evaluated before the expression in the *let-expression* is evaluated.
- Unless the expressions in the *variable-list* are accessed, they must not be evaluated.
- Errors that are raised during the evaluation of the expressions in the *let-expression* are propagated.

A let expression can be seen as syntactic sugar over an implicit record expression. The following expression is equivalent to the example above:

Power Query M

```
[
    x = 1 + 1,
    y = 2 + 2,
    z = y + 1,
    result = x + y + z
][result]
```

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Conditionals

06/13/2025

The *if-expression* selects from two expressions based on the value of a logical input value and evaluates only the selected expression.

if-expression:

```
if if-condition then true-expression else false-expression
```

if-condition:

expression

true-expression:

expression

false-expression:

expression

The following are examples of *if-expressions*:

Power Query M

```
if 2 > 1 then 2 else 1          // 2
if 1 = 1 then "yes" else "no"  // "yes"
```

The following holds when evaluating an *if-expression*:

- If the value produced by evaluating the *if-condition* isn't a logical value, then an error with reason code "Expression.Error" is raised.
- The *true-expression* is only evaluated if the *if-condition* evaluates to the value `true`.
- The *false-expression* is only evaluated if the *if-condition* evaluates to the value `false`.
- The result of the *if-expression* is the value of the *true-expression* if the *if-condition* is `true`, and the value of the *false-expression* if the *if-condition* is `false`.
- Errors raised during the evaluation of the *if-condition*, *true-expression*, or *false-expression* are propagated.

Functions

Article • 02/13/2025

A *function* is a value that represents a mapping from a set of argument values to a single value. A function is invoked by provided a set of input values (the argument values), and produces a single output value (the return value).

Writing functions

Functions are written using a *function-expression*:

function-expression:

(*parameter-list*_{opt}) *return-type*_{opt} => *function-body*

function-body:

expression

parameter-list:

fixed-parameter-list

fixed-parameter-list , *optional-parameter-list*

optional-parameter-list

fixed-parameter-list:

parameter

parameter , *fixed-parameter-list*

parameter:

parameter-name *parameter-type*_{opt}

parameter-name:

identifier

parameter-type:

primitive-or-nullable-primitive-type-assertion

return-type:

primitive-or-nullable-primitive-type-assertion

primitive-or-nullable-primitive-type-assertion:

as *primitive-or-nullable-primitive-type*

optional-parameter-list:

optional-parameter

optional-parameter , *optional-parameter-list*

optional-parameter:

optional *parameter*

primitive-or-nullable-primitive-type:

nullable_{opt} *primitive-type*

The following is an example of a function that requires exactly two values `x` and `y`, and produces the result of applying the `+` operator to those values. The `x` and `y` are *parameters* that are part of the *parameter-list* of the function, and the `x + y` is the *function-body*:

```
Power Query M

(x, y) => x + y
```

The result of evaluating a *function-expression* is to produce a function value (not to evaluate the *function-body*). As a convention in this document, function values (as opposed to function expressions) are shown with the *parameter-list* but with an ellipsis (`...`) instead of the *function-body*. For example, once the function expression above has been evaluated, it would be shown as the following function value:

```
Power Query M

(x, y) => ...
```

The following operators are defined for function values:

[Expand table](#)

Operator	Result
<code>x = y</code>	Equal
<code>x <> y</code>	Not equal

The native type of function values is a custom function type (derived from the intrinsic type `function`) that lists the parameter names and specifies all parameter types and the return type to be `any`. (Go to [Function types](#) for details on function types.)

Invoking functions

The *function-body* of a function is executed by *invoking* the function value using an *invoke-expression*. Invoking a function value means the *function-body* of the function value is evaluated and a value is returned or an error is raised.

invoke-expression:

```
primary-expression ( argument-listopt )
```

argument-list:

expression-list

Each time a function value is invoked, a set of values are specified as an *argument-list*, called the *arguments* to the function.

An *argument-list* is used to specify a fixed number of arguments directly as a list of expressions. The following example defines a record with a function value in a field, and then invokes the function from another field of the record:

Power Query M

```
[
    MyFunction = (x, y, z) => x + y + z,
    Result1 = MyFunction(1, 2, 3)           // 6
]
```

The following holds when invoking a function:

- The environment used to evaluate the *function-body* of the function includes a variable that corresponds to each parameter, with the same name as the parameter. The value of each parameter corresponds to a value constructed from the *argument-list* of the *invoke-expression*, as defined in [Parameters](#).
- All of the expressions corresponding to the function arguments are evaluated before the *function-body* is evaluated.
- Errors raised when evaluating the expressions in the *expression-list* or *function-body* are propagated.
- The number of arguments constructed from the *argument-list* must be compatible with the parameters of the function, or an error is raised with reason code `"Expression.Error"`. The process for determining compatibility is defined in [Parameters](#).

Parameters

There are two kinds of parameters that may be present in a *parameter-list*:

- A *required* parameter indicates that an argument corresponding to the parameter must always be specified when a function is invoked. Required parameters must be specified first in the *parameter-list*. The function in the following example defines required parameters `x` and `y`:

Power Query M

```
[
    MyFunction = (x, y) => x + y,

    Result1 = MyFunction(1, 1),      // 2
    Result2 = MyFunction(2, 2)      // 4
]
```

- An *optional* parameter indicates that an argument corresponding to the parameter may be specified when a function is invoked, but is not required to be specified. If an argument that corresponds to an optional parameter is not specified when the function is invoked, then the value `null` is used instead. Optional parameters must appear after any required parameters in a *parameter-list*. The function in the following example defines a fixed parameter `x` and an optional parameter `y`:

Power Query M

```
[
    MyFunction = (x, optional y) =>
        if (y = null) x else x + y,
    Result1 = MyFunction(1),          // 1
    Result2 = MyFunction(1, null),    // 1
    Result3 = MyFunction(2, 2),      // 4
]
```

The number of arguments that are specified when a function is invoked must be compatible with the parameter list. Compatibility of a set of arguments `A` for a function `F` is computed as follows:

- Let the value `N` represent the number of arguments `A` constructed from the *argument-list*. For example:

Power Query M

```
MyFunction()           // N = 0
MyFunction(1)          // N = 1
MyFunction(null)       // N = 1
MyFunction(null, 2)    // N = 2
MyFunction(1, 2, 3)    // N = 3
MyFunction(1, 2, null) // N = 3
MyFunction(1, 2, {3, 4}) // N = 3
```

- Let the value *Required* represent the number of fixed parameters of `F` and *Optional* the number of optional parameters of `F`. For example:

Power Query M

```
()           // Required = 0, Optional = 0
(x)          // Required = 1, Optional = 0
(optional x) // Required = 0, Optional = 1
(x, optional y) // Required = 1, Optional = 1
```

- Arguments **A** are compatible with function **F** if the following are true:
 - $(N \geq \text{Fixed})$ and $(N \leq (\text{Fixed} + \text{Optional}))$
 - The argument types are compatible with **F**'s corresponding parameter types
- If the function has a declared return type, then the result value of the body of function **F** is compatible with **F**'s return type if the following is true:
 - The value yielded by evaluating the function body with the supplied arguments for the function parameters has a type that is compatible with the return type.
- If the function body yields a value incompatible with the function's return type, an error with reason code `"Expression.Error"` is raised.

Recursive functions

In order to write a function value that is recursive, it is necessary to use the scoping operator (**@**) to reference the function within its scope. For example, the following record contains a field that defines the `Factorial` function, and another field that invokes it:

Power Query M

```
[
    Factorial = (x) =>
        if x = 0 then 1 else x * @Factorial(x - 1),
    Result = Factorial(3) // 6
]
```

Similarly, mutually recursive functions can be written as long as each function that needs to be accessed has a name. In the following example, part of the `Factorial` function has been refactored into a second `Factorial2` function.

Power Query M

```
[
    Factorial = (x) => if x = 0 then 1 else Factorial2(x),
    Factorial2 = (x) => x * Factorial(x - 1),
]
```

```
Result = Factorial(3)    // 6  
]
```

Closures

A function can return another function as a value. This function can in turn depend on one or more parameters to the original function. In the following example, the function associated with the field `MyFunction` returns a function that returns the parameter specified to it:

Power Query M

```
[  
    MyFunction = (x) => () => x,  
    MyFunction1 = MyFunction(1),  
    MyFunction2 = MyFunction(2),  
    Result = MyFunction1() + MyFunction2()    // 3  
]
```

Each time the function is invoked, a new function value will be returned that maintains the value of the parameter so that when it is invoked, the parameter value will be returned.

Functions and environments

In addition to parameters, the *function-body* of a *function-expression* can reference variables that are present in the environment when the function is initialized. For example, the function defined by the field `MyFunction` accesses the field `c` of the enclosing record `A`:

Power Query M

```
[  
    A =  
        [  
            MyFunction = () => C,  
            C = 1  
        ],  
    B = A[MyFunction]()    // 1  
]
```

When `MyFunction` is invoked, it accesses the value of the variable `c`, even though it is being invoked from an environment (`B`) that does not contain a variable `c`.

Simplified declarations

The *each-expression* is a syntactic shorthand for declaring untyped functions taking a single parameter named `_` (underscore).

each-expression:

`each` *each-expression-body*

each-expression-body:

function-body

Simplified declarations are commonly used to improve the readability of higher-order function invocation.

For example, the following pairs of declarations are semantically equivalent:

Power Query M

```
each _ + 1
(_) => _ + 1
each [A]
(_) => _[A]
```

```
Table.SelectRows( aTable, each [Weight] > 12 )
Table.SelectRows( aTable, (_) => _[Weight] > 12 )
```

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Error Handling

Article • 03/07/2023

The result of evaluating an M expression produces one of the following outcomes:

- A single value is produced.
- An *error is raised*, indicating the process of evaluating the expression could not produce a value. An error contains a single record value that can be used to provide additional information about what caused the incomplete evaluation.

Errors can be raised from within an expression, and can be handled from within an expression.

Raising errors

The syntax for raising an error is as follows:

error-raising-expression:

```
error expression
```

Text values can be used as shorthand for error values. For example:

Power Query M

```
error "Hello, world" // error with message "Hello, world"
```

Full error values are records and can be constructed using the `Error.Record` function:

Power Query M

```
error Error.Record("FileNotFound", "File my.txt not found",  
    "my.txt")
```

The above expression is equivalent to:

Power Query M

```
error [  
    Reason = "FileNotFound",  
    Message = "File my.txt not found",  
    Detail = "my.txt"  
]
```

Raising an error will cause the current expression evaluation to stop, and the expression evaluation stack will unwind until one of the following occurs:

- A record field, section member, or let variable—collectively: an *entry*—is reached. The entry is marked as having an error, the error value is saved with that entry, and then propagated. Any subsequent access to that entry will cause an identical error to be raised. Other entries of the record, section, or let expression are not necessarily affected (unless they access an entry previously marked as having an error).
- The top-level expression is reached. In this case, the result of evaluating the top-level expression is an error instead of a value.
- A `try` expression is reached. In this case, the error is captured and returned as a value.

Handling errors

An *error-handling-expression* (informally known as a "try expression") is used to handle an error:

error-handling-expression:

`try` *protected-expression* *error-handler*_{opt}

protected-expression:

expression

error-handler:

otherwise-clause

catch-clause

otherwise-clause:

`otherwise` *default-expression*

default-expression:

expression

catch-clause:

`catch` *catch-function*

catch-function:

(*parameter-name*_{opt}) => *function-body*

The following holds when evaluating an *error-handling-expression* without an *error-handler*:

- If the evaluation of the *protected-expression* does not result in an error and produces a value *x*, the value produced by the *error-handling-expression* is a record

of the following form:

Power Query M

```
[ HasErrors = false, Value = x ]
```

- If the evaluation of the *protected-expression* raises an error value *e*, the result of the *error-handling-expression* is a record of the following form:

Power Query M

```
[ HasErrors = true, Error = e ]
```

The following holds when evaluating an *error-handling-expression* with an *error-handler*:

- The *protected-expression* must be evaluated before the *error-handler*.
- The *error-handler* must be evaluated if and only if the evaluation of the *protected-expression* raises an error.
- If the evaluation of the *protected-expression* raises an error, the value produced by the *error-handling-expression* is the result of evaluating the *error-handler*.
- Errors raised during the evaluation of the *error-handler* are propagated.
- When the *error-handler* being evaluated is a *catch-clause*, the *catch-function* is invoked. If that function accepts a parameter, the error value will be passed as its value.

The following example illustrates an *error-handling-expression* in a case where no error is raised:

Power Query M

```
let
    x = try "A"
in
    if x[HasError] then x[Error] else x[Value]
// "A"
```

The following example shows raising an error and then handling it:

Power Query M

```
let
    x = try error "A"
```

```
in
    if x[HasError] then x[Error] else x[Value]
// [ Reason = "Expression.Error", Message = "A", Detail = null ]
```

The preceding example can be rewritten with less syntax by using a *catch-clause* with a *catch-function* that accepts a parameter:

Power Query M

```
let
    x = try error "A" catch (e) => e
in
    x
// [ Reason = "Expression.Error", Message = "A", Detail = null ]
```

An *otherwise-clause* can be used to replace errors handled by a try expression with an alternative value:

Power Query M

```
try error "A" otherwise 1
// 1
```

A *catch-clause* with a zero-parameter *catch-function* is effectively a longer, alternative syntax for an *otherwise-clause*:

Power Query M

```
try error "A" catch () => 1
// 1
```

If the *error-handler* also raises an error, then so does the entire try expression:

Power Query M

```
try error "A" otherwise error "B"
// error with message "B"
```

Power Query M

```
try error "A" catch () => error "B"
// error with message "B"
```

Power Query M

```
try error "A" catch (e) => error "B"  
// error with message "B"
```

Errors in record and let initializers

The following example shows a record initializer with a field **A** that raises an error and is accessed by two other fields **B** and **C**. Field **B** does not handle the error that is raised by **A**, but **C** does. The final field **D** does not access **A** and so it is not affected by the error in **A**.

Power Query M

```
[  
    A = error "A",  
    B = A + 1,  
    C = let x =  
        try A in  
        if not x[HasError] then x[Value]  
        else x[Error],  
    D = 1 + 1  
]
```

The result of evaluating the above expression is:

Power Query M

```
[  
    A = // error with message "A"  
    B = // error with message "A"  
    C = "A",  
    D = 2  
]
```

Error handling in M should be performed close to the cause of errors to deal with the effects of lazy field initialization and deferred closure evaluations. The following example shows an unsuccessful attempt at handling an error using a `try` expression:

Power Query M

```
let  
    f = (x) => [ a = error "bad", b = x ],  
    g = try f(42) otherwise 123  
in  
    g[a] // error "bad"
```

In this example, the definition `g` was meant to handle the error raised when calling `f`. However, the error is raised by a field initializer that only runs when needed and thus after the record was returned from `f` and passed through the `try` expression.

Not implemented error

While an expression is being developed, an author may want to leave out the implementation for some parts of the expression, but may still want to be able to execute the expression. One way to handle this case is to raise an error for the unimplemented parts. For example:

Power Query M

```
(x, y) =>
    if x > y then
        x - y
    else
        error Error.Record("Expression.Error",
            "Not Implemented")
```

The ellipsis symbol (`...`) can be used as a shortcut for `error`.

not-implemented-expression:

`...`

For example, the following is equivalent to the previous example:

Power Query M

```
(x, y) => if x > y then x - y else ...
```

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Sections

Article • 10/10/2022

A *section-document* is an M program that consists of multiple named expressions.

section-document:

section

section:

*literal-attributes*_{opt} **section** *section-name* **;** *section-members*_{opt}

section-name:

identifier

section-members:

section-member *section-members*_{opt}

section-member:

*literal-attributes*_{opt} **shared**_{opt} *section-member-name* **=** *expression* **;**

section-member-name:

identifier

In M, a section is an organizational concept that allows related expressions to be named and grouped within a document. Each section has a *section-name*, which identifies the section and qualifies the names of the *section-members* declared within the section. A *section-member* consists of a *member-name* and an *expression*. Section member expressions may refer to other section members within the same section directly by member name.

The following example shows a section-document:

Power Query M

```
section Section1;
```

```
A = 1; //1
```

```
B = 2; //2
```

```
C = A + B; //3
```

Section member expressions may refer to section members located in other sections by means of a *section-access-expression*, which qualifies a section member name with the name of the containing section.

section-access-expression:

identifier **!** *identifier*

The following example shows a set of two documents containing sections that are mutually referential:

Power Query M

```
section Section1;
A = "Hello";           //"Hello"
B = 1 + Section2!A;     //3

section Section2;
A = 2;                 //2
B = Section1!A & " world!"; //"Hello, world"
```

Section members may optionally be declared as `shared`, which omits the requirement to use a *section-access-expression* when referring to shared members outside of the containing section. Shared members in external sections may be referred to by their unqualified member name so long as no member of the same name is declared in the referring section and no other section has a like-named shared member.

The following example illustrates the behavior of shared members when used across sections within the same set of documents:

Power Query M

```
section Section1;
shared A = 1;           // 1

section Section2;
B = A + 2;              // 3 (refers to shared A from Section1)

section Section3;
A = "Hello";           // "Hello"
B = A + " world";      // "Hello world" (refers to local A)
C = Section1!A + 2;    // 3
```

Defining a shared member with the same name in different sections will produce a valid global environment, however accessing the shared member will raise an error when accessed.

Power Query M

```
section Section1;
shared A = 1;

section Section2;
shared A = "Hello";
```

```
section Section3;  
B = A;    //Error: shared member A has multiple definitions
```

The following holds when evaluating a set of section-documents:

- Each *section-name* must be unique in the global environment.
- Within a section, each *section-member* must have a unique *section-member-name*.
- Shared section members with more than one definition raise an error when the shared member is accessed.
- The expression component of a *section-member* must not be evaluated before the section member is accessed.
- Errors raised while the expression component of a *section-member* is evaluated are associated with that section member before propagating outward and then re-raised each time the section member is accessed.

Document Linking

A set of M section documents can be linked into an opaque record value that has one field per shared member of the section documents. If shared members have ambiguous names, an error is raised.

The resulting record value fully closes over the global environment in which the link process was performed. Such records are, therefore, suitable components to compose M documents from other (linked) sets of M documents. There are no opportunities for naming conflicts.

The standard library functions `Embedded.Value` can be used to retrieve such "embedded" record values that correspond to reused M components.

Document Introspection

M provides programmatic access to the global environment by means of the `#sections` and `#shared` keywords.

`#sections`

The `#sections` intrinsic variable returns all sections within the global environment as a record. This record is keyed by section name and each value is a record representation

of the corresponding section indexed by section member name.

The following example shows a document consisting of two sections and the record produced by evaluating the `#sections` intrinsic variable within the context of that document:

Power Query M

```
section Section1;
A = 1;
B = 2;

section Section2;
C = "Hello";
D = "world";

#sections
//[
//  Section1 = [ A = 1, B = 2],
//  Section2 = [ C = "Hello", D = "world" ]
//]
```

The following holds when evaluating `#sections`:

- The `#sections` intrinsic variable preserves the evaluation state of all section member expressions within the document.
- The `#sections` intrinsic variable does not force the evaluation of any unevaluated section members.

#shared

The `#shared` intrinsic variable returns the contents of the global environment as a record. (The global environment consists of all shared section members as well as any identifiers directly included in the global environment by the expression evaluator.) This record is keyed by identifier name, with each value being the value of the associated identifier.

The following example shows a document with two shared members and the corresponding record produced by evaluating the `#shared` intrinsic variable within the context of that document:

Power Query M

```
section Section1;
shared A = 1;
B = 2;
```

```
Section Section2;  
C = "Hello";  
shared D = "world";  
  
//[  
//  A = 1,  
//  D = "world"  
//]
```

The following holds when evaluating `#shared`:

- The `#shared` intrinsic variable preserves the evaluation state of the global environment.
- The `#shared` intrinsic variable does not force the evaluation of any unevaluated value.

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Consolidated Grammar

Article • 01/29/2025

Lexical grammar

lexical-unit:

lexical-elements_{opt}

lexical-elements:

lexical-element lexical-elements_{opt}

lexical-element:

whitespace

token

comment

White space

whitespace:

Any character with Unicode class Zs

Horizontal tab character (U+0009)

Vertical tab character (U+000B)

Form feed character (U+000C)

Carriage return character (U+000D) followed by line feed character (U+000A) *new-*

line-character

new-line-character:

Carriage return character (U+000D)

Line feed character (U+000A)

Next line character (U+0085)

Line separator character (U+2028)

Paragraph separator character (U+2029)

Comment

comment:

single-line-comment

delimited-comment

single-line-comment:

// single-line-comment-characters_{opt}

single-line-comment-characters:

single-line-comment-character single-line-comment-characters_{opt}

single-line-comment-character:

Any Unicode character except a *new-line-character*

delimited-comment:

/ *** *delimited-comment-text_{opt}* *** */*

delimited-comment-text:

delimited-comment-section delimited-comment-text_{opt}

delimited-comment-section:

/

asterisks_{opt} *not-slash-or-asterisk*

asterisks:

*** *asterisks_{opt}*

not-slash-or-asterisk:

Any Unicode character except *** or */*

Tokens

token:

identifier

keyword

literal

operator-or-punctuator

Character escape sequences

character-escape-sequence:

*(* *escape-sequence-list* *)*

escape-sequence-list:

single-escape-sequence

escape-sequence-list *,* *single-escape-sequence*

single-escape-sequence:

long-unicode-escape-sequence

short-unicode-escape-sequence

control-character-escape-sequence

escape-escape

long-unicode-escape-sequence:

hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

short-unicode-escape-sequence:

hex-digit hex-digit hex-digit hex-digit

control-character-escape-sequence:

control-character

control-character:

`cr`

`lf`

`tab`

escape-escape:

`#`

Literals

literal:

logical-literal

number-literal

text-literal

null-literal

verbatim-literal

logical-literal:

`true`

`false`

number-literal:

decimal-number-literal

hexadecimal-number-literal

decimal-digits:

decimal-digit decimal-digits_{opt}

decimal-digit: one of

`0 1 2 3 4 5 6 7 8 9`

hexadecimal-number-literal:

`0x` *hex-digits*

`0X` *hex-digits*

hex-digits:

hex-digit hex-digits_{opt}

hex-digit: one of

`0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`

decimal-number-literal:

decimal-digits `.` *decimal-digits exponent-part_{opt}*

`.` *decimal-digits exponent-part_{opt}*

decimal-digits exponent-part_{opt}

exponent-part:

`e` *sign_{opt} decimal-digits*

`E` *sign_{opt} decimal-digits*

sign: one of

`+ -`

text-literal:

`"` *text-literal-characters_{opt}* `"`

text-literal-characters:

text-literal-character *text-literal-characters_{opt}*

text-literal-character:

single-text-character

character-escape-sequence

double-quote-escape-sequence

single-text-character:

Any character except `"` (U+0022) or `#` (U+0023) followed by `(` (U+0028)

double-quote-escape-sequence:

`" "` (U+0022, U+0022)

null-literal:

`null`

verbatim-literal:

`#!` *text-literal-characters_{opt}* `"`

Identifiers

identifier:

regular-identifier

quoted-identifier

regular-identifier:

available-identifier

available-identifier *dot-character* *regular-identifier*

available-identifier:

A *keyword-or-identifier* that is not a *keyword*

keyword-or-identifier:

letter-character

underscore-character

identifier-start-character *identifier-part-characters*

identifier-start-character:

letter-character

underscore-character

identifier-part-characters:

identifier-part-character *identifier-part-characters_{opt}*

identifier-part-character:

letter-character

decimal-digit-character

underscore-character

connecting-character

combining-character

formatting-character

generalized-identifier:

generalized-identifier-part

generalized-identifier separated only by blanks (`U+0020`) *generalized-identifier-part*

generalized-identifier-part:

generalized-identifier-segment

decimal-digit-character generalized-identifier-segment

generalized-identifier-segment:

keyword-or-identifier

keyword-or-identifier dot-character keyword-or-identifier

dot-character:

`.` (`U+002E`)

underscore-character:

`_` (`U+005F`)

letter-character:

A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:

A Unicode character of classes Mn or Mc

decimal-digit-character:

A Unicode character of the class Nd

connecting-character:

A Unicode character of the class Pc

formatting-character:

A Unicode character of the class Cf

quoted-identifier:

`#" text-literal-charactersopt "`

Keywords and predefined identifiers

Predefined identifiers and keywords cannot be redefined. A quoted identifier can be used to handle identifiers that would otherwise collide with predefined identifiers or keywords.

keyword: one of

`and as each else error false if in is let meta not null or otherwise`

section shared then true try type #binary #date #datetime

#datetimezone #duration #infinity #nan #sections #shared #table #time

Operators and punctuators

operator-or-punctuator: one of

, ; = < <= > >= <> + - * / & () [] { } @ ? ?? =>

Syntactic grammar

Documents

document:

section-document

expression-document

Section Documents

section-document:

section

section:

*literal-attributes*_{opt} **section** *section-name* **;** *section-members*_{opt}

section-name:

identifier

section-members:

section-member *section-members*_{opt}

section-member:

*literal-attributes*_{opt} *shared*_{opt} *section-member-name* **=** *expression* **;**

section-member-name:

identifier

Expression Documents

Expressions

expression-document:

expression

expression:

logical-or-expression

each-expression
function-expression
let-expression
if-expression
error-raising-expression
error-handling-expression

Logical expressions

logical-or-expression:
logical-and-expression
logical-and-expression **or** *logical-or-expression*
logical-and-expression:
is-expression
logical-and-expression **and** *is-expression*

Is expression

is-expression:
as-expression
is-expression **is** *primitive-or-nullable-primitive-type*

As expression

as-expression:
equality-expression
as-expression **as** *primitive-or-nullable-primitive-type*

Equality expression

equality-expression:
relational-expression
relational-expression **=** *equality-expression*
relational-expression **<>** *equality-expression*

Relational expression

relational-expression:
additive-expression
additive-expression **<** *relational-expression*

additive-expression **>** *relational-expression*
additive-expression **<=** *relational-expression*
additive-expression **>=** *relational-expression*

Arithmetic expressions

additive-expression:

multiplicative-expression
multiplicative-expression **+** *additive-expression*
multiplicative-expression **-** *additive-expression*
multiplicative-expression **&** *_additive-expression*

multiplicative-expression:

metadata-expression
metadata-expression ***** *multiplicative-expression*
metadata-expression **/** *multiplicative-expression*

Metadata expression

metadata-expression:

unary-expression
unary-expression **meta** *unary-expression*

Unary expression

unary-expression:

type-expression
+ *unary-expression*
- *unary-expression*
not *unary-expression*

Primary expression

primary-expression:

literal-expression
list-expression
record-expression
identifier-expression
section-access-expression
parenthesized-expression

field-access-expression
item-access-expression
invoke-expression
not-implemented-expression

Literal expression

literal-expression:
literal

Identifier expression

identifier-expression:
identifier-reference
identifier-reference:
exclusive-identifier-reference
inclusive-identifier-reference
exclusive-identifier-reference:
identifier
inclusive-identifier-reference:
@ identifier

Section-access expression

section-access-expression:
identifier ! identifier

Parenthesized expression

parenthesized-expression:
(expression)

Not-implemented expression

not-implemented-expression:
...

Invoke expression

invoke-expression:

primary-expression (*argument-list*_{opt})

argument-list:

expression

expression , *argument-list*

List expression

list-expression:

{ *item-list*_{opt} }

item-list:

item

item , *item-list*

item:

expression

expression .. *expression*

Record expression

record-expression:

[*field-list*_{opt}]

field-list:

field

field , *field-list*

field:

field-name = *expression*

field-name:

generalized-identifier

quoted-identifier

Item access expression

item-access-expression:

item-selection

optional-item-selection

item-selection:

primary-expression { *item-selector* }

optional-item-selection:

primary-expression { *item-selector* } ?

item-selector:

expression

Field access expressions

field-access-expression:

field-selection

implicit-target-field-selection

projection

implicit-target-projection

field-selection:

primary-expression field-selector

field-selector:

required-field-selector

optional-field-selector

required-field-selector:

[*field-name*]

optional-field-selector:

[*field-name*] ?

implicit-target-field-selection:

field-selector

projection:

primary-expression required-projection

primary-expression optional-projection

required-projection:

[*required-selector-list*]

optional-projection:

[*required-selector-list*] ?

required-selector-list:

required-field-selector

required-field-selector , *required-selector-list*

implicit-target-projection:

required-projection

optional-projection

Function expression

function-expression:

(*parameter-list*_{opt}) *return-type*_{opt} => *function-body*

function-body:

expression

parameter-list:

fixed-parameter-list

fixed-parameter-list , *optional-parameter-list*

optional-parameter-list

fixed-parameter-list:

parameter

parameter , *fixed-parameter-list*

parameter:

parameter-name *parameter-type*_{opt}

parameter-name:

identifier

parameter-type:

primitive-or-nullable-primitive-type-assertion

return-type:

primitive-or-nullable-primitive-type-assertion

primitive-or-nullable-primitive-type-assertion:

as *primitive-or-nullable-primitive-type*

optional-parameter-list:

optional-parameter

optional-parameter , *optional-parameter-list*

optional-parameter:

optional *parameter*

Each expression

each-expression:

each *each-expression-body*

each-expression-body:

function-body

Let expression

let-expression:

let *variable-list* **in** *expression*

variable-list:

variable

variable , *variable-list*

variable:

variable-name **=** *expression*

variable-name:

identifier

If expression

if-expression:

`if` *if-condition* `then` *true-expression* `else` *false-expression*

if-condition:

expression

true-expression:

expression

false-expression:

expression

Type expression

type-expression:

primary-expression

`type` *primary-type*

type:

primary-expression

primary-type

primary-type:

primitive-or-nullable-primitive-type

record-type

list-type

function-type

table-type

nullable-type

primitive-or-nullable-primitive-type:

`nullable`_{opt} *primitive-type*

primitive-type: one of

`any` `nonnull` `binary` `date` `datetime` `datetimezone` `duration` `function` `list` `logical`

`none` `null` `number` `record` `table` `text` `time` `type`

record-type:

`[` *open-record-marker* `]`

`[` *field-specification-list*_{opt} `]`

`[` *field-specification-list* `,` *open-record-marker* `]`

field-specification-list:

field-specification

field-specification , *field-specification-list*

field-specification:

*optional*_{opt} *field-name* *field-type-specification*_{opt}

field-type-specification:

= *field-type*

field-type:

type

open-record-marker:

...

list-type:

{ *item-type* }

item-type:

type

function-type:

function (*parameter-specification-list*_{opt}) *return-type*

parameter-specification-list:

required-parameter-specification-list

required-parameter-specification-list , *optional-parameter-specification-list*

optional-parameter-specification-list

required-parameter-specification-list:

required-parameter-specification

required-parameter-specification , *required-parameter-specification-list*

required-parameter-specification:

parameter-specification

optional-parameter-specification-list:

optional-parameter-specification

optional-parameter-specification , *optional-parameter-specification-list*

optional-parameter-specification:

optional *parameter-specification*

parameter-specification:

parameter-name *parameter-type*

parameter-type:

type-assertion

type-assertion:

as *type*

table-type:

table *row-type*

row-type:

[*field-specification-list*_{opt}]

nullable-type:

`nullable` *type*

Error raising expression

error-raising-expression:

`error` *expression_*

Error handling expression

error-handling-expression:

`try` *protected-expression* *error-handler*_{opt}

protected-expression:

expression

error-handler:

otherwise-clause

catch-clause

otherwise-clause:

`otherwise` *default-expression*

default-expression:

expression

catch-clause:

`catch` *catch-function*

catch-function:

`(` *parameter-name*_{opt} `) =>` *function-body*

Literal Attributes

literal-attributes:

record-literal

record-literal:

`[` *literal-field-list*_{opt} `]`

literal-field-list:

literal-field

literal-field `,` *literal-field-list*

literal-field:

field-name `=` *any-literal*

list-literal:

`{` *literal-item-list*_{opt} `}`

literal-item-list:

any-literal

any-literal , *literal-item-list*

any-literal:

record-literal

list-literal

logical-literal

number-literal

text-literal

null-literal

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Types in the Power Query M formula language

Article • 01/14/2025

The Power Query M Formula Language is a useful and expressive data mashup language. But it does have some limitations. For example, there is no strong enforcement of the type system. In some cases, a more rigorous validation is needed. Fortunately, M provides a built-in library with support for types to make stronger validation feasible.

Developers should have a thorough understanding of the type system in-order to do this with any generality. And, while the Power Query M language specification explains the type system well, it does leave a few surprises. For example, validation of function instances requires a way to compare types for compatibility.

By exploring the M type system more carefully, many of these issues can be clarified, and developers will be empowered to craft the solutions they need.

Knowledge of predicate calculus and naïve set theory should be adequate to understand the notation used.

PRELIMINARIES

(1) $B := \{ \text{true}; \text{false} \}$

B is the typical set of Boolean values

(2) $N := \{ \text{valid M identifiers} \}$

N is the set of all valid names in M. This is defined elsewhere.

(3) $P := \langle B, T \rangle$

P is the set of function parameters. Each one is possibly optional, and has a type. Parameter names are irrelevant.

(4) $P^n := \bigcup_{0 \leq i \leq n} \langle i, P^i \rangle$

Pⁿ is the set of all ordered sequences of n function parameters.

(5) $P^* := \bigcup_{0 \leq i \leq \infty} P^i$

P^{*} is the set of all possible sequences of function parameters, from length 0 on up.

(6) $F := \langle B, N, T \rangle$

F is the set of all record fields. Each field is possibly optional, has a name, and a type.

$$(7) F^n := \prod_{0 \leq i \leq n} F$$

F^n is the set of all sets of n record fields.

$$(8) F^* := (\bigcup_{0 \leq i \leq \infty} F^i) \setminus \{ F \mid \langle b_1, n_1, t_1 \rangle, \langle b_2, n_2, t_2 \rangle \in F \wedge n_1 = n_2 \}$$

F^* is the set of all sets (of any length) of record fields, except for the sets where more than one field has the same name.

$$(9) C := \langle N, T \rangle$$

C is the set of column types, for tables. Each column has a name and a type.

$$(10) C^n \subset \bigcup_{0 \leq i \leq n} \langle i, C \rangle$$

C^n is the set of all ordered sequences of n column types.

$$(11) C^* := (\bigcup_{0 \leq i \leq \infty} C^i) \setminus \{ C^m \mid \langle a, \langle n_1, t_1 \rangle \rangle, \langle b, \langle n_2, t_2 \rangle \rangle \in C^m \wedge n_1 = n_2 \}$$

C^* is the set of all combinations (of any length) of column types, except for those where more than one column has the same name.

M TYPES

$$(12) T_F := \langle P, P^* \rangle$$

A Function Type consists of a return type, and an ordered list of zero-or-more function parameters.

$$(13) T_L := \llbracket T \rrbracket$$

A List type is indicated by a given type (called the "item type") wrapped in curly braces. Since curly braces are used in the metalanguage, $\llbracket \rrbracket$ brackets are used in this document.

$$(14) T_R := \langle B, F^* \rangle$$

A Record Type has a flag indicating whether it's "open", and zero-or-more unordered record fields.

$$(15) T_R^o := \langle \text{true}, F \rangle$$

$$(16) T_R^\bullet := \langle \text{false}, F \rangle$$

T_R^o and $T_R^\bullet$ are notational shortcuts for open and closed record types, respectively.

$$(17) T_T := C^*$$

A Table Type is an ordered sequence of zero-or-more column types, where there are no name collisions.

$$(18) T_P := \{ \text{any; none; null; logical; number; time; date; datetime; datetimezone; duration; text; binary; type; list; record; table; function; anynonnull} \}$$

A Primitive Type is one from this list of M keywords.

(19) $T_N := \{ t_n, u \in T \mid t_n = u + \text{null} \} = \text{nullable } t$

Any type can additionally be marked as being nullable, by using the "nullable" keyword.

(20) $T := T_F \cup T_L \cup T_R \cup T_T \cup T_P \cup T_N$

The set of all M types is the union of these six sets of types:

Function Types, List Types, Record Types, Table Types, Primitive Types, and Nullable Types.

FUNCTIONS

One function needs to be defined: $\text{NonNullable} : T \leftarrow T$

This function takes a type, and returns a type that is equivalent except it does not conform with the null value.

IDENTITIES

Some identities are needed to define some special cases, and may also help elucidate the above.

(21) nullable any = any

(22) nullable anynonnull = any

(23) nullable null = null

(24) nullable none = null

(25) nullable nullable $t \in T = \text{nullable } t$

(26) $\text{NonNullable}(\text{nullable } t \in T) = \text{NonNullable}(t)$

(27) $\text{NonNullable}(\text{any}) = \text{anynonnull}$

TYPE COMPATIBILITY

As defined elsewhere, an M type is compatible with another M type if and only if all values that conform to the first type also conform to the second type.

Here is defined a compatibility relation that does not depend on conforming values, and is based on the properties of the types themselves. It is anticipated that this relation, as defined in this document, is completely equivalent to the original semantic definition.

The "is compatible with" relation : $\leq : B \leftarrow T \times T$

In the below section, a lowercase t will always represent an M Type, an element of T .

A Φ will represent a subset of F^* , or of C^* .

(28) $t \leq t$

This relation is reflexive.

$$(29) t_a \leq t_b \wedge t_b \leq t_c \rightarrow t_a \leq t_c$$

This relation is transitive.

$$(30) \text{none} \leq t \leq \text{any}$$

M types form a lattice over this relation; none is the bottom, and any is the top.

$$(31) t_a, t_b \in T_N \wedge t_a \leq t_b \rightarrow \text{NonNullable}(t_a) \leq \text{NonNullable}(t_b)$$

If two types are compatible, then the NonNullable equivalents are also compatible.

$$(32) \text{null} \leq t \in T_N$$

The primitive type null is compatible with all nullable types.

$$(33) t \notin T_N \leq \text{anynonnull}$$

All nonnullable types are compatible with anynonnull.

$$(34) \text{NonNullable}(t) \leq t$$

A NonNullable type is compatible with the nullable equivalent.

$$(35) t \in T_F \rightarrow t \leq \text{function}$$

All function types are compatible with function.

$$(36) t \in T_L \rightarrow t \leq \text{list}$$

All list types are compatible with list.

$$(37) t \in T_R \rightarrow t \leq \text{record}$$

All record types are compatible with record.

$$(38) t \in T_T \rightarrow t \leq \text{table}$$

All table types are compatible with table.

$$(39) t_a \leq t_b \leftrightarrow \llbracket t_a \rrbracket \leq \llbracket t_b \rrbracket$$

A list type is compatible with another list type if the item types are compatible, and vice-versa.

$$(40) t_a \in T_F = \langle p_a, p^* \rangle, t_b \in T_F = \langle p_b, p^* \rangle \wedge p_a \leq p_b \rightarrow t_a \leq t_b$$

A function type is compatible with another function type if the return types are compatible, and the parameter lists are identical.

$$(41) t_a \in T_R^0, t_b \in T_R^* \rightarrow t_a \not\leq t_b$$

An open record type is never compatible with a closed record type.

$$(42) t_a \in T_R^* = \langle \text{false}, \Phi \rangle, t_b \in T_R^0 = \langle \text{true}, \Phi \rangle \rightarrow t_a \leq t_b$$

A closed record type is compatible with an otherwise identical open record type.

$$(43) t_a \in T_R^0 = \langle \text{true}, (\Phi, \langle \text{true}, n, \text{any} \rangle) \rangle, t_b \in T_R^0 = \langle \text{true}, \Phi \rangle \rightarrow t_a \leq t_b \wedge t_b \leq t_a$$

An optional field with the type any may be ignored when comparing two open record

types.

$$(44) t_a \in T_R = \langle b, (\Phi, \langle \beta, n, u_a \rangle) \rangle, t_b \in T_R = \langle b, (\Phi, \langle \beta, n, u_b \rangle) \rangle \wedge u_a \leq u_b \rightarrow t_a \leq t_b$$

Two record types that differ only by one field are compatible if the name and optionality of the field are identical, and the types of said field are compatible.

$$(45) t_a \in T_R = \langle b, (\Phi, \langle \text{false}, n, u \rangle) \rangle, t_b \in T_R = \langle b, (\Phi, \langle \text{true}, n, u \rangle) \rangle \rightarrow t_a \leq t_b$$

A record type with a non-optional field is compatible with a record type identical but for that field being optional.

$$(46) t_a \in T_R^0 = \langle \text{true}, (\Phi, \langle b, n, u \rangle) \rangle, t_b \in T_R^0 = \langle \text{true}, \Phi \rangle \rightarrow t_a \leq t_b$$

An open record type is compatible with another open record type with one fewer field.

$$(47) t_a \in T_T = (\Phi, \langle i, \langle n, u_a \rangle \rangle), t_b \in T_T = (\Phi, \langle i, \langle n, u_b \rangle \rangle) \wedge u_a \leq u_b \rightarrow t_a \leq t_b$$

A table type is compatible with a second table type, which is identical but for one column having a differing type, when the types for that column are compatible.

Related content

- [Power Query M language specification](#)
- [Power Query M function reference](#)
- [Types and type conversion](#)
- [Types](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Expressions, values, and let expression

Article • 07/09/2024

A Power Query M formula language query is composed of formula **expression** steps that create a mashup query. A formula expression can be evaluated (computed), yielding a value. The **let** expression encapsulates a set of values to be computed, assigned names, and then used in a subsequent expression that follows the **in** statement. For example, a let expression could contain a **Source** variable that equals the value of `Text.Proper` and yields a text value in proper case.

Let expression

Power Query M

```
let
    Source = Text.Proper("hello world")
in
    Source
```

In the example above, `Text.Proper("hello world")` is evaluated to `"Hello World"`.

The next sections describe value types in the language.

Primitive value

A *primitive* value is single-part value, such as a number, logical, text, or null. A null value can be used to indicate the absence of any data.

 Expand table

Type	Example value
Binary	00 00 00 02 // number of points (2)
Date	5/23/2015
DateTime	5/23/2015 12:00:00 AM
DateTimeZone	5/23/2015 12:00:00 AM -08:00
Duration	15:35:00
Logical	true and false

Type	Example value
Null	null
Number	0, 1, -1, 1.5, and 2.3e-5
Text	"abc"
Time	12:34:12 PM

Function value

A *Function* is a value that, when invoked with arguments, produces a new value. Functions are written by listing the function's *parameters* in parentheses, followed by the goes-to symbol `=>`, followed by the expression defining the function. For example, to create a function called **MyFunction** that has two parameters and performs a calculation on `parameter1` and `parameter2`:

Power Query M

```
let
    MyFunction = (parameter1, parameter2) => (parameter1 + parameter2) / 2
in
    MyFunction
```

Calling **MyFunction** returns the result:

Power Query M

```
let
    Source = MyFunction(2, 4)
in
    Source
```

This code produces the value of 3.

Structured data values

The M language supports the following structured data values:

- [List](#)
- [Record](#)
- [Table](#)

- [Additional structured data examples](#)

ⓘ Note

Structured data can contain any M value. To see a couple of examples, go to [Additional structured data examples](#).

List

A *List* is a zero-based ordered sequence of values enclosed in curly brace characters `{ }`. The curly brace characters `{ }` are also used to retrieve an item from a List by index position. For more information, go to [List values](#).

ⓘ Note

Power Query M supports an infinite list size, but if a list is written as a literal, the list has a fixed length. For example, `{1, 2, 3}` has a fixed length of 3.

The following are some List examples.

[Expand table](#)

Value	Type
<code>{123, true, "A"}</code>	List containing a number, a logical, and text.
<code>{1, 2, 3}</code>	List of numbers
<code>{ {1, 2, 3}, {4, 5, 6} }</code>	List of List of numbers
<code>{ [CustomerID = 1, Name = "Bob", Phone = "123-4567"], [CustomerID = 2, Name = "Jim", Phone = "987-6543"] }</code>	List of Records
<code>{123, true, "A"}{0}</code>	Get the value of the first item in a List. This expression returns the value 123.
<code>{ {1, 2, 3}, }</code>	Get the value of the second item from the first List element. This expression returns the value 2.

Value	Type
{4, 5, 6}	
{0}{1}	

Record

A *Record* is a set of fields. A *field* is a name/value pair where the name is a text value that is unique within the field's record. The syntax for record values allows the names to be written without quotes, a form also referred to as *identifiers*. An identifier can take the following two forms:

- identifier_name such as OrderID.
- #"identifier name" such as #"Today's data is: ".

The following is a record containing fields named "OrderID", "CustomerID", "Item", and "Price" with values 1, 1, "Fishing rod", and 100.00. Square brace characters [] denote the beginning and end of a record expression, and are used to get a field value from a record. The following examples show a record and how to get the Item field value.

Here's an example record:

```
Power Query M

let Source =
    [
        OrderID = 1,
        CustomerID = 1,
        Item = "Fishing rod",
        Price = 100.00
    ]
in Source
```

To get the value of an Item, you use square brackets as `Source[Item]`:

```
Power Query M

let Source =
    [
        OrderID = 1,
        CustomerID = 1,
        Item = "Fishing rod",
        Price = 100.00
    ]
in Source[Item] //equals "Fishing rod"
```

Table

A *Table* is a set of values organized into named columns and rows. The column type can be implicit or explicit. You can use `#table` to create a list of column names and list of rows. A Table of values is a List in a [List](#). The curly brace characters { } are also used to retrieve a row from a Table by index position (go to [Example 3 - Get a row from a table by index position](#)).

Example 1 - Create a table with implicit column types

Power Query M

```
let
    Source = #table(
        {"OrderID", "CustomerID", "Item", "Price"},
        {
            {1, 1, "Fishing rod", 100.00},
            {2, 1, "1 lb. worms", 5.00}
        }
    )
in
    Source
```

Example 2 - Create a table with explicit column types

Power Query M

```
let
    Source = #table(
        type table [OrderID = number, CustomerID = number, Item = text, Price = number],
        {
            {1, 1, "Fishing rod", 100.00},
            {2, 1, "1 lb. worms", 5.00}
        }
    )
in
    Source
```

Both of the previous examples creates a table with the following shape:

[Expand table](#)

OrderID	CustomerID	Item	Price
1	1	Fishing rod	100.00

OrderID	CustomerID	Item	Price
2	1	1 lb. worms	5.00

Example 3 - Get a row from a table by index position

Power Query M
<pre>let Source = #table(type table [OrderID = number, CustomerID = number, Item = text, Price = number], { {1, 1, "Fishing rod", 100.00}, {2, 1, "1 lb. worms", 5.00} }) in Source{1}</pre>

This expression returns the follow record:

[Expand table](#)

Field	Value
OrderID	2
CustomerID	1
Item	1 lb. worms
Price	5

Additional structured data examples

Structured data can contain any M value. Here are some examples:

Example 1 - List with [Primitive](#_Primitive_value_1) values, [Function](#_Function_value), and [Record](#_Record_value)

Power Query M
<pre>let Source = {</pre>

```

1,
"Bob",
DateTime.ToText(DateTime.LocalNow(), "yyyy-MM-dd"),
[OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0]
}
in
Source

```

Evaluating this expression can be visualized as:

A List containing a Record	
1	
"Bob"	
2015-05-22	
OrderID	1
CustomerID	1
Item	"Fishing rod"
Price	100.0

Example 2 - Record containing primitive values and nested records

Power Query M

```

let
    Source = [CustomerID = 1, Name = "Bob", Phone = "123-4567", Orders =
        {
            [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price =
100.0],
            [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0]
        }
    ]
in
    Source

```

Evaluating this expression can be visualized as:

A record containing a List of Records		
CustomerID	1	
Name	"Bob"	
Phone	"123-4567"	
Orders	OrderID	1
	CustomerID	1
	Item	"Fishing rod"
	Price	100.0
	OrderID	2
	CustomerID	1
	Item	"1 lb. worms"
	Price	5.0

ⓘ Note

Although many values can be written literally as an expression, a value isn't an expression. For example, the expression 1 evaluates to the value 1; the expression 1+1 evaluates to the value 2. This distinction is subtle, but important. Expressions are recipes for evaluation; values are the results of evaluation.

If expression

The **if** expression selects between two expressions based on a logical condition. For example:

Power Query M

```
if 2 > 1 then
    2 + 2
else
    1 + 1
```

The first expression $(2 + 2)$ is selected if the logical expression $(2 > 1)$ is true, and the second expression $(1 + 1)$ is selected if it is false. The selected expression (in this case $2 + 2$) is evaluated and becomes the result of the **if** expression (4).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Comments

Article • 10/10/2022

You can add comments to your code with single-line comments `//` or multi-line comments that begin with `/*` and end with `*/`.

Example - Single-line comment

Power Query M

```
let
    //Convert to proper case.
    Source = Text.Proper("hello world")
in
    Source
```

Example - Multi-line comment

Power Query M

```
/* Capitalize each word in the Item column in the Orders table. Text.Proper
is evaluated for each Item in each table row. */
let
    Orders = Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "fishing net", Price =
25.0]}),
    #"Capitalized Each Word" = Table.TransformColumns(Orders, {"Item",
Text.Proper})
in
    #"Capitalized Each Word"
```

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Evaluation model

Article • 10/10/2022

The evaluation model of the Power Query M formula language is modeled after the evaluation model commonly found in spreadsheets, where the order of calculations can be determined based on dependencies between the formulas in the cells.

If you have written formulas in a spreadsheet such as Excel, you may recognize the formulas on the left will result in the values on the right when calculated:

	A
1	=A2 * 2
2	=A3 + 1
3	1

	A
1	4
2	2
3	1

In M, an expression can reference previous expressions by name, and the evaluation process will automatically determine the order in which referenced expressions are calculated.

Let's use a record to produce an expression which is equivalent to the above spreadsheet example. When initializing the value of a field, you refer to other fields within the record by the name of the field, as follows:

Power Query M

```
[  
    A1 = A2 * 2,  
    A2 = A3 + 1,  
    A3 = 1  
]
```

The above expression evaluates to the following record:

Power Query M

```
[  
    A1 = 4,  
    A2 = 2,  
    A3 = 1  
]
```

Records can be contained within, or **nested**, within other records. You can use the **lookup operator** ([]) to access the fields of a record by name. For example, the following record has a field named Sales containing a record, and a field named Total that accesses the FirstHalf and SecondHalf fields of the Sales record:

Power Query M

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = Sales[FirstHalf] + Sales[SecondHalf]
]
```

The above expression evaluates to the following record:

Power Query M

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = 2100
]
```

You use the **positional index operator** ({ }) to access an item in a list by its numeric index. The values within a list are referred to using a zero-based index from the beginning of the list. For example, the indexes 0 and 1 are used to reference the first and second items in the list below:

Power Query M

```
[
    Sales =
        {
            [
                Year = 2007,
                FirstHalf = 1000,
                SecondHalf = 1100,
                Total = FirstHalf + SecondHalf // equals 2100
            ],
            [
                Year = 2008,
                FirstHalf = 1200,
                SecondHalf = 1300,
                Total = FirstHalf + SecondHalf // equals 2500
            ]
        },
    #"Total Sales" = Sales{0}[Total] + Sales{1}[Total] // equals 4600
]
```

Lazy and eager evaluation

List, **Record**, and **Table** member expressions, as well as **let** expressions (Go to [Expressions, values, and let expression](#)), are evaluated using **lazy evaluation**. That is, they are evaluated when needed. All other expressions are evaluated using **eager evaluation**. That is, they are evaluated immediately when encountered during the evaluation process. A good way to think about this is to remember that evaluating a list or record expression will return a list or record value that knows how its list items or record fields need to be computed, when requested (by lookup or index operators).

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Operators

06/13/2025

The Power Query M formula language includes a set of operators that can be used in an expression. **Operators** are applied to **operands** to form symbolic expressions. For example, in the expression `1 + 2` the numbers `1` and `2` are operands and the operator is the addition operator `(+)`.

The meaning of an operator can vary depending on the type of operand values. The language has the following operators:

Plus operator (+)

[Expand table](#)

Expression	Equals
<code>1 + 2</code>	Numeric addition: <code>3</code>
<code>#time(12,23,0) + #duration(0,0,2,0)</code>	Time arithmetic: <code>#time(12,25,0)</code>

Combination operator (&)

[Expand table](#)

Function	Equals
<code>"A" & "BC"</code>	Text concatenation: <code>"ABC"</code>
<code>{1} & {2, 3}</code>	List concatenation: <code>{1, 2, 3}</code>
<code>[a = 1] & [b = 2]</code>	Record merge: <code>[a = 1, b = 2]</code>

List of M operators

Common operators that apply to `null`, `logical`, `number`, `time`, `date`, `datetime`, `datetimezone`, `duration`, `text`, `binary`

[Expand table](#)

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

Logical operators (In addition to **Common operators**)

 Expand table

Operator	Description
or	Conditional logical OR
and	Conditional logical AND
not	Logical NOT

Number operators (In addition to **Common operators**)

 Expand table

Operator	Description
+	Sum
-	Difference
*	Product
/	Quotient
+x	Unary plus
-x	Negation

Text operators (In addition to **Common operators**)

[Expand table](#)

Operator	Description
<code>&</code>	Concatenation

List, record, table operators

[Expand table](#)

Operator	Description
<code>=</code>	Equal
<code><></code>	Not equal
<code>&</code>	Concatenation

Record lookup operator

[Expand table](#)

Operator	Description
<code>[]</code>	Access the fields of a record by name.

List indexer operator

[Expand table](#)

Operator	Description
<code>{}</code>	Access an item in a list by its zero-based numeric index.

Type compatibility and assertion operators

[Expand table](#)

Operator	Description
<code>is</code>	The expression <code>x is y</code> returns <code>true</code> if the type of <code>x</code> is compatible with <code>y</code> , and returns <code>false</code> if the type of <code>x</code> is not compatible with <code>y</code> .
<code>as</code>	The expression <code>x as y</code> asserts that the value <code>x</code> is compatible with <code>y</code> as per the <code>is</code> operator.

Date operators

[Expand table](#)

Operator	Left Operand	Right Operand	Meaning
<code>x + y</code>	<code>time</code>	<code>duration</code>	Date offset by duration
<code>x + y</code>	<code>duration</code>	<code>time</code>	Date offset by duration
<code>x - y</code>	<code>time</code>	<code>duration</code>	Date offset by negated duration
<code>x - y</code>	<code>time</code>	<code>time</code>	Duration between dates
<code>x & y</code>	<code>date</code>	<code>time</code>	Merged datetime

Datetime operators

[Expand table](#)

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

Datetimezone operators

[Expand table](#)

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

Duration operators

[Expand table](#)

Operator	Left Operand	Right Operand	Meaning
<code>x + y</code>	<code>datetime</code>	<code>duration</code>	Datetime offset by duration
<code>x + y</code>	<code>duration</code>	<code>datetime</code>	Datetime offset by duration
<code>x + y</code>	<code>duration</code>	<code>duration</code>	Sum of durations
<code>x - y</code>	<code>datetime</code>	<code>duration</code>	Datetime offset by negated duration
<code>x - y</code>	<code>datetime</code>	<code>datetime</code>	Duration between datetimes
<code>x - y</code>	<code>duration</code>	<code>duration</code>	Difference of durations
<code>x * y</code>	<code>duration</code>	<code>number</code>	N times a duration
<code>x * y</code>	<code>number</code>	<code>duration</code>	N times a duration
<code>x / y</code>	<code>duration</code>	<code>number</code>	Fraction of a duration

ⓘ **Note**

Not all combinations of values can be supported by an operator. Expressions that, when evaluated, encounter undefined operator conditions evaluate to errors. For more information about errors in M, go to [Errors](#)

Error example:

 Expand table

Function	Equals
<code>1 + "2"</code>	Error: adding number and text is not supported

Types and type conversion

Article • 01/10/2025

Power Query M uses types to classify values to have a more structured data set. This article describes the most commonly-used M types and how to convert one type to another type.

Commonly-used types

Data types refers to any type that's used to clarify the structure of specific data. The most commonly used data types are primitive types. These types include:

- `type any`, which classifies any value.
- `type null`, which classifies the null value.
- `type logical`, which classifies the values true and false.
- `type number`, which classifies number values.
- `type time`, which classifies time values.
- `type date`, which classifies date values.
- `type datetime`, which classifies datetime values.
- `type datetimezone`, which classifies datetimezone values.
- `type duration`, which classifies duration values.
- `type text`, which classifies text values.
- `type binary`, which classifies binary values.
- `type type`, which classifies type values.
- `type list`, which classifies list values.
- `type record`, which classifies record values.
- `type table`, which classifies table values.
- `type function`, which classifies function values.
- `type anynonnull`, which classifies all values excluding null.
- `type none`, which classifies no values.

For more information about these types, go to [Types](#).

In addition to these common data types, there is also a set of data types using the format `*.Type`. The most commonly used data types of this format are:

- `Byte.Type`, which classifies an 8-bit number value.
- `Int8.Type`, which classifies an 8-bit number value.
- `Int16.Type`, which classifies a 16-bit number value.

- `Int32.Type`, which classifies a 32-bit number value.
- `Int64.Type`, which classifies a 64-bit number value.
- `Single.Type`, which classifies a 9-digit floating number value.
- `Double.Type`, which classifies a 17-digit floating number value.
- `Decimal.Type`, which classifies a 15-digit floating number value.
- `Currency.Type`, which classifies a 19-digit number value with four digits to the right of the "." separator.
- `Percentage.Type`, which classifies a 15-digit number value with a mask to format the value as a percentage.
- `Guid.Type`, which classifies a GUID text value.

The primitive types can also be written in the `*.Type` format as well. Therefore, you can write `number` as `Number.Type`, `record` as `Record.Type`, and so on.

When you use any of these types, be aware that, like all M code, these types are case-sensitive.

The following table contains more information about each of these types.

 Expand table

Data type	Description
<code>any</code>	The <code>any</code> data type is the status given when a value doesn't have an explicit data type definition. The <code>any</code> type is the data type that classifies all values.
<code>binary</code>	The <code>binary</code> data type can be used to represent any other data with a binary format.
<code>type</code>	A value that classifies other values. For more information, go to Types .
<code>null</code>	Represents the absence of a value, or a value of indeterminate or unknown state.
<code>anynonnull</code>	Represents any type that is nonnullable.
<code>date</code>	Represents just a date (no time portion).
<code>time</code>	Represents just time (no date portion).
<code>datetime</code>	Represents both a date and time value. The time portion of a date is stored as a fraction to whole multiples of 1/300 seconds (3.33 ms). Dates between the years 1900 and 9999 are supported.
<code>datetimezone</code>	Represents a UTC date and time with a time-zone offset.

Data type	Description
<code>duration</code>	Represents a length of time. This type can be added or subtracted from a <code>datetime</code> field with correct results. For more information, go to Duration .
<code>text</code>	A Unicode character data string. Can be strings, numbers, or dates represented in a text format. Maximum string length is 268,435,456 Unicode characters (where each Unicode character is two bytes) or 536,870,912 bytes.
<code>logical</code>	A Boolean value of either <code>true</code> or <code>false</code> .
<code>list</code>	A value which produces a sequence of values when enumerated. For more information, go to List types and List values .
<code>record</code>	An ordered sequence of fields. Each field contains a field name and field value. For more information, go to Record types and Record values .
<code>table</code>	An ordered sequence of rows divided into columns. For more information, go to Table types and Table values .
<code>function</code>	A value that maps a set of arguments to a single value. For more information, go to Functions and Function types .
<code>number</code>	Represents any number used for numeric and arithmetic operations. For more information, go to Number .
<code>Decimal.Type</code>	Represents a 64-bit (eight-byte) floating-point number. It's the most common number type, and corresponds to numbers as you usually think of them. Although designed to handle numbers with fractional values, it also handles whole numbers. The <code>Decimal.Type</code> can handle negative values from $-1.79E+308$ through $-2.23E-308$, 0, and positive values from $2.23E-308$ through $1.79E+308$. For example, numbers like 34, 34.01, and 34.000367063 are valid decimal numbers. The largest precision that can be represented in a <code>Decimal.Type</code> is 15 digits long. The decimal separator can occur anywhere in the number. The <code>Decimal.Type</code> corresponds to how Excel stores its numbers. Note that a binary floating-point number can't represent all numbers within its supported range with 100% accuracy. Thus, minor differences in precision might occur when representing certain decimal numbers.
<code>Currency.Type</code>	This data type has a fixed location for the decimal separator. The decimal separator always has four digits to its right and allows for 19 digits of significance. The largest value it can represent is 922,337,203,685,477.5807 (positive or negative). Unlike <code>Decimal.Type</code> , the <code>Currency.Type</code> is always precise and is thus useful in cases where the imprecision of floating-point notation might introduce errors.
<code>Percentage.Type</code>	Fundamentally the same as a <code>Decimal.Type</code> , but it has a mask to format the values as a percentage value.
<code>Int8.Type</code>	Represents an 8-bit (one-byte) signed integer value. Because it's an integer, it has no digits to the right of the decimal place. It allows for 3 digits; a positive

Data type	Description
	or negative whole number between –128 and 127. As with the <code>Currency.Type</code> , the <code>Int8.Type</code> can be useful in cases where you need to control rounding.
<code>Int16.Type</code>	Represents a 16-bit (two-byte) signed integer value. Because it's an integer, it has no digits to the right of the decimal place. It allows for 6 digits; a positive or negative whole number between –32,768 (-2^{15}) and 32,767 ($2^{15}-1$). As with the <code>Currency.Type</code> , the <code>Int16.Type</code> can be useful in cases where you need to control rounding.
<code>Int32.Type</code>	Represents a 32-bit (four-byte) signed integer value. Because it's an integer, it has no digits to the right of the decimal place. It allows for 10 digits; a positive or negative whole number between –2,147,483,648 (-2^{31}) and 2,147,483,647 ($2^{31}-1$). As with the <code>Currency.Type</code> , the <code>Int32.Type</code> can be useful in cases where you need to control rounding.
<code>Int64.Type</code>	Represents a 64-bit (eight-byte) signed integer value. Because it's an integer, it has no digits to the right of the decimal place. It allows for 19 digits; a positive or negative whole number between –9,223,372,036,854,775,808 (-2^{63}) and 9,223,372,036,854,775,807 ($2^{63}-1$). It can represent the largest possible precision of the various numeric data types. As with the <code>Currency.Type</code> , the <code>Int64.Type</code> can be useful in cases where you need to control rounding.
<code>Byte.Type</code>	Represents an 8-bit (one-byte) unsigned integer value. Because it's an unsigned integer, it has no digits to the right of the decimal place and can only contain positive values. It allows for 3 digits; a positive number between 0 and 255.
<code>Single.Type</code>	Represents a single-precision floating-point number. It has an approximate range of -3.99×10^{38} to 3.99×10^{38} and supports approximately 9 digits of precision. It can also represent positive and negative infinity, and NaN (Not a Number).
<code>Double.Type</code>	Represents a double-precision floating-point number. It has an approximate range of $-1.7976931348623158 \times 10^{307}$ to $1.7976931348623158 \times 10^{307}$ and supports approximately 17 digits of precision. It can also represent positive and negative infinity, and NaN (Not a Number).
<code>Guid.Type</code>	Represents a 128-bit text value consisting of 32 hexadecimal values using the form factor of <8 hex values>-<4 hex values>-<4 hex values>-<4 hex values>-<12 hex values>, which make up the GUID value.
<code>none</code>	The data type that classifies no values.

The only other commonly used `*.Type` values are enumerations. For more information, go to [Enumerations](#).

Type conversion

The Power Query M formula language has formulas to convert between types. The following is a summary of conversion formulas in M.

Number

 Expand table

Type conversion	Description
Number.FromText(text as text) as number	Returns a number value from a text value.
Number.ToText(number as number) as text	Returns a text value from a number value.
Number.From(value as any) as number	Returns a number value from a value.
Byte.From(value as any) as number	Returns an 8-bit integer number value from the given value.
Int8.From(value as any) as number	Returns an 8-bit integer number value from the given value.
Int16.From(value as any) as number	Returns a 16-bit integer number value from the given value.
Int32.From(value as any) as number	Returns a 32-bit integer number value from the given value.
Int64.From(value as any) as number	Returns a 64-bit integer number value from the given value.
Single.From(value as any) as number	Returns a Single number value from the given value.
Double.From(value as any) as number	Returns a Double number value from the given value.
Decimal.From(value as any) as number	Returns a Decimal number value from the given value.
Currency.From(value as any) as number	Returns a Currency number value from the given value.
Percentage.From(value as any) as number	Returns a Percentage number value from the given value.

Text

 Expand table

Type conversion	Description
Text.From(value as any) as text	Returns the text representation of a number, date, time, datetime, datetimezone, logical, duration or binary value.
Guid.From(value as text) as text	Returns the GUID representation of the specified text.

Logical

[Expand table](#)

Type conversion	Description
Logical.FromText(text as text) as logical	Returns a logical value of true or false from a text value.
Logical.ToText(logical as logical) as text	Returns a text value from a logical value.
Logical.From(value as any) as logical	Returns a logical value from a value.

Date, Time, DateTime, and DateTimeZone

[Expand table](#)

Type conversion	Description
.FromText(text as text) as date, time, datetime, or datetimezone	Returns a date, time, datetime, or datetimezone value from a set of date formats and culture value.
.ToText(date, time, dateTime, or dateTimeZone as date, time, datetime, or datetimezone) as text	Returns a text value from a date, time, datetime, or datetimezone value.
.From(value as any)	Returns a date, time, datetime, or datetimezone value from a value.
.ToRecord(date, time, dateTime, or dateTimeZone as date, time, datetime, or datetimezone)	Returns a record containing parts of a date, time, datetime, or datetimezone value.

Related content

- [Types](#)
- [Power Query M type system](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Metadata

Article • 08/28/2024

Metadata is information about a value that is associated with a value. **Metadata** is represented as a record value, called a metadata record. The fields of a **metadata record** can be used to store the metadata for a value. Every value has a metadata record. If the value of the metadata record hasn't been specified, then the metadata record is empty (has no fields). Associating a metadata record with a value doesn't change the value's behavior in evaluations except for those that explicitly inspect metadata records.

Metadata records

A metadata record value is associated with a value *x* using the syntax `value meta [record]`. For example, the following associates a metadata record with `Rating` and `Tags` fields with the text value "Mozart":

Power Query M

```
"Mozart" meta [ Rating = 5,
Tags = {"Classical"} ]
```

A metadata record can be accessed for a value using the [Value.Metadata](#) function. In the following example, the expression in the `ComposerRating` field accesses the metadata record of the value in the `Composer` field, and then accesses the `Rating` field of the metadata record.

Power Query M

```
[
  Composer = "Mozart" meta [ Rating = 5, Tags = {"Classical"} ],
  ComposerRating = Value.Metadata(Composer)[Rating] // 5
]
```

Metadata records aren't preserved when a value is used with an operator or function that constructs a new value. For example, if two text values are concatenated using the `&` operator, the metadata of the resulting text value is an empty record `[]`.

The standard library functions [Value.RemoveMetadata](#) and [Value.ReplaceMetadata](#) can be used to remove all metadata from a value and to replace a value's metadata.

Limitations

Some hosts that use Power Query to transform or move data don't support storing custom metadata into storage. The following hosts don't support storing the custom metadata:

- Power BI dataflows
- Fabric Dataflow Gen2
- Power Platform dataflows

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Errors

Article • 10/10/2022

An *error* in Power Query M formula language is an indication that the process of evaluating an expression could not produce a value. Errors are raised by operators and functions encountering **error** conditions or by using the **error** expression. Errors are handled using the **try** expression. When an error is raised, a value is specified that can be used to indicate why the error occurred.

Try expression

A try expression converts values and errors into a record value that indicates whether the try expression handled an error, or not, and either the proper value or the error record it extracted when handling the error. For example, consider the following expression that raises an error and then handles it right away:

Power Query M

```
try error "negative unit count"
```

This expression evaluates to the following nested record value, explaining the [HasError], [Error], and [Message] field lookups in the unit-price example before.

Error record

Power Query M

```
[
  HasError = true,
  Error =
    [
      Reason = "Expression.Error",
      Message = "negative unit count",
      Detail = null
    ]
]
```

A common case is to replace errors with default values. The try expression can be used with an optional otherwise clause to achieve just that in a compact form:

Power Query M

```
try error "negative unit count" otherwise 42
// equals 42
```

Error example

Power Query M

```
let Sales =
    [
        ProductName = "Fishing rod",
        Revenue = 2000,
        Units = 1000,
        UnitPrice = if Units = 0 then error "No Units"
                    else Revenue / Units
    ],

    //Get UnitPrice from Sales record
    textUnitPrice = try Number.ToText(Sales[UnitPrice]),
    Label = "Unit Price: " &
        (if textUnitPrice[HasError] then textUnitPrice[Error][Message]
        //Continue expression flow
        else textUnitPrice[Value])
in
    Label
```

The previous example accesses the `Sales[UnitPrice]` field and formats the value producing the result:

Power Query M

```
"Unit Price: 2"
```

If the Units field had been zero, then the `UnitPrice` field would have raised an error which would have been handled by the try. The resulting value would then have been:

Power Query M

```
"No Units"
```

Feedback

Was this page helpful?

 Yes

 No

Get help at [Microsoft Q&A](#)

Local, fixed, and UTC variants of current time functions

06/27/2025

When you work with Power Query in tools like Excel and Power BI, handling date and time values accurately is essential—especially when your data transformations depend on the current time. Power Query offers various functions to retrieve the current date and time:

- [DateTime.LocalNow](#)
- [DateTimeZone.LocalNow](#)
- [DateTime.FixedLocalNow](#)
- [DateTimeZone.FixedLocalNow](#)
- [DateTimeZone.UtcNow](#)
- [DateTimeZone.FixedUtcNow](#).

This article explores the distinctions between these functions and clarifies when and why to use each one. In addition, it highlights a critical but often overlooked detail. Power Query Online always returns UTC time even when using a function labeled as "Local." Understanding these nuances can help you avoid unexpected results, especially when building time-sensitive reports or automating data updates in apps such as Power BI service or Power Apps.

Differences between functions

Each of the current time functions have important differences. These functions vary in terms of time zone awareness, volatility (whether the value changes when called multiple times in the same query), and how they behave in different environments (desktop vs. online). The following table contains a breakdown of each function.

 Expand table

Function	Returns	Volatility	Desktop behavior	Online behavior	Typical use case
<code>DateTime.LocalNow</code>	A <code>datetime</code> representing the current local time	Dynamic—returns a new value each time it's invoked during query evaluation	Returns local machine time	Returns UTC time	Quick local timestamp without time zone context

Function	Returns	Volatility	Desktop behavior	Online behavior	Typical use case
<code>DateTimeZone.LocalNow</code>	A <code>datetimezone</code> value representing the current local time with time zone offset	Dynamic—returns a new value each time it's invoked during query evaluation	Returns local time with offset	Returns UTC time with <code>+00:00</code> offset	Local time with time zone awareness
<code>DateTime.FixedLocalNow</code>	A <code>datetime</code> value representing the local time when first invoked during query evaluation	Fixed—returns the same value throughout a single query evaluation	Captures local time when first called	Captures UTC time when first called	Snapshot of local time without time zone
<code>DateTimeZone.FixedLocalNow</code>	A <code>datetimezone</code> value representing the local time with offset when first invoked during query evaluation	Fixed—returns the same value throughout a single query evaluation	Captures local time with offset when first called	Captures UTC time with <code>+00:00</code> offset when first called	Snapshot of local time with time zone
<code>DateTimeZone.UtcNow</code>	A <code>datetimezone</code> value representing the current UTC time	Dynamic—returns a new value each time it's invoked during query evaluation	Returns current UTC time	Returns current UTC time	Consistent UTC timestamp for dynamic scenarios
<code>DateTimeZone.FixedUtcNow</code>	A <code>datetimezone</code> value representing the UTC time when first invoked during query evaluation	Fixed—returns the same value throughout a single query evaluation	Captures UTC time when first called	Captures UTC time when first called	Fixed UTC timestamp for logging or auditing

In Power Query M, choosing between local time and UTC-based date and time functions is a critical design decision that affects the consistency, accuracy, and portability of your queries. Functions like `DateTime.LocalNow` and `DateTime.FixedLocalNow` are useful when your logic depends on the local system time, such as filtering for records that occurred "today" or generating timestamps for user-facing reports. These functions reflect the time zone of the

environment in which the query is executed, making them suitable for Power Query Desktop scenarios where the local context is well-defined.

However, in distributed or cloud-based environments like Power Query Online, these same functions return UTC time, not the actual local time of the user. This discrepancy can lead to subtle inconsistencies if your logic assumes a local time context. In contrast,

`DateTimeZone.UtcNow` and `DateTimeZone.FixedUtcNow` provide a time-zone-neutral reference point that is consistent across environments and unaffected by daylight saving time or regional settings. These UTC-based functions are the preferred choice for scenarios involving data integration, logging, auditing, or any logic that must behave identically regardless of where or when the query runs.

Differences between the LocalNow and FixedLocalNow functions

Power Query M provides four functions for retrieving the current local time:

- `DateTime.LocalNow` returns the current local `datetime` each time the expression is evaluated.
- `DateTime.FixedLocalNow` returns the local `datetime` once per query evaluation, acting as a snapshot.
- `DateTimeZone.LocalNow` returns the current local `datetimezone` each time the expression is evaluated.
- `DateTimeZone.FixedLocal` now returns the local `datetimezone` once per query evaluation, acting as a snapshot

To demonstrate the difference, the following example generates a table with multiple rows. Each row captures a fresh `DateTime.LocalNow` value using a delay to ensure distinct timestamps, while each captured `DateTime.FixedLocalNow` value remains constant across all rows.

ⓘ Note

All the dates and times in the output of the examples in this article depend on when the functions are run. The dates and times shown in the output are for demonstration purposes only.

Power Query M

```
let
    // Create a table with LocalNow and FixedLocalNow columns
    TableWithTimes = Table.FromList(
```

```

        {1..5},
        each {
            _,
            Function.InvokeAfter(() => DateTime.LocalNow(), #duration(0, 0, 0,
0.2)),
            Function.InvokeAfter(() => DateTime.FixedLocalNow(), #duration(0, 0,
0, 0.2))
        },
        {"Index", "LocalNow", "FixedLocalNow"}
    ),

    // Format both datetime columns
    FormatLocalNow = Table.TransformColumns(TableWithTimes,
        {"LocalNow", each DateTime.ToText(_, "yyyy-MM-ddThh:mm:ss.fff")}),
    FormatFixedNow = Table.TransformColumns(FormatLocalNow,
        {"FixedLocalNow", each DateTime.ToText(_, "yyyy-MM-ddThh:mm:ss.fff")}),

    // Change the table types
    FinalTable = Table.TransformColumnTypes(FormatFixedNow, {"Index",
Int64.Type},
        {"LocalNow", type text}, {"FixedLocalNow", type text})

in
    FinalTable

```

The output of this example is:

	1.2 Index	LocalNow	FixedLocalNow
1	1	2025-05-28T09:40:15.277	2025-05-28T09:40:14.976
2	2	2025-05-28T09:40:15.496	2025-05-28T09:40:14.976
3	3	2025-05-28T09:40:15.700	2025-05-28T09:40:14.976
4	4	2025-05-28T09:40:15.903	2025-05-28T09:40:14.976
5	5	2025-05-28T09:40:16.106	2025-05-28T09:40:14.976

If you look at the output, you might notice that even though the `DateTime.LocalNow` function appears first in the code, the value returned for `DateTime.FixedLocalNow` shows a time that occurs before the `DateTime.LocalTime` time. Even though `DateTime.LocalNow` is listed first in the table construction, the order of evaluation in Power Query M isn't guaranteed to follow the order of fields in a table. Instead, Power Query uses a lazy evaluation model. Using this model means that fields are only evaluated when needed and the engine determines the evaluation order, not the order in your code. In this case, the `DateTime.FixedLocalNow` function is evaluated first, so the first time returned for this function occurs before the first time returned for `DateTime.LocalNow`.

The following example shows how to produce similar results using `DateTimeZone.LocalNow` and `DateTimeZone.FixedLocalNow`.

Power Query M

```
let
    // Create a table with LocalNow and FixedLocalNow columns
    TableWithTimes = Table.FromList(
        {1..5},
        each {
            ~
            Function.InvokeAfter(() => DateTimeZone.LocalNow(), #duration(0, 0, 0,
0.2)),
            Function.InvokeAfter(() => DateTimeZone.FixedLocalNow(), #duration(0,
0, 0, 0.2))
        },
        {"Index", "LocalNow", "FixedLocalNow"}
    ),

    // Format both datetimezone columns
    FormatLocalNow = Table.TransformColumns(TableWithTimes,
        {"LocalNow", each DateTimeZone.ToText(_, "yyyy-MM-
ddThh:mm:ss.fff:zzz")})),
    FormatFixedNow = Table.TransformColumns(FormatLocalNow,
        {"FixedLocalNow", each DateTimeZone.ToText(_, "yyyy-MM-
ddThh:mm:ss.fff:zzz")})),

    // Change the table types
    FinalTable = Table.TransformColumnTypes(FormatFixedNow,
        {"Index", Int64.Type}, {"LocalNow", type text}, {"FixedLocalNow", type
text}})
in
    FinalTable
```

The output of this example in Power Query Desktop is:

	123 Index	AB C LocalNow	AB C FixedLocalNow
1	1	2025-05-30T12:34:02.616:-07:00	2025-05-30T12:34:02.382:-07:00
2	2	2025-05-30T12:34:03.026:-07:00	2025-05-30T12:34:02.382:-07:00
3	3	2025-05-30T12:34:03.436:-07:00	2025-05-30T12:34:02.382:-07:00
4	4	2025-05-30T12:34:03.846:-07:00	2025-05-30T12:34:02.382:-07:00
5	5	2025-05-30T12:34:04.257:-07:00	2025-05-30T12:34:02.382:-07:00

ⓘ Note

If you run this example in Power Query Online, the time returned is always UTC time and the time zone portion of the returned values are always `+00:00`.

Differences between the.UtcNow and the FixedUtcNow functions

Power Query M provides two functions for retrieving the current UTC time:

- `DateTimeZone.UtcNow` returns the current UTC `datetimezone` each time the expression is evaluated.
- `DateTimeZone.FixedUtcNow` returns the local `datetimezone` once per query evaluation, acting as a snapshot.

The differences between these two functions are similar to the `LocalNow` and `FixedLocalNow` functions. However, whether the functions are run in Power Query Desktop or Power Query Online, the return values are always returned as UTC time. The following example demonstrates the differences between these two functions.

Power Query M

```
let
    // Create a table with UtcNow and FixedUtcNow columns
    TableWithTimes = Table.FromList(
        {1..5},
        each {
            ~,
            Function.InvokeAfter(() => DateTimeZone.UtcNow(), #duration(0, 0, 0,
0.2)),
            Function.InvokeAfter(() => DateTimeZone.FixedUtcNow(), #duration(0, 0,
0, 0.2))
        },
        {"Index", "UtcNow", "FixedUtcNow"}
    ),

    // Format both datetimezone columns
    FormatLocalNow = Table.TransformColumns(TableWithTimes,
        {{ "UtcNow", each DateTimeZone.ToText(_, "yyyy-MM-ddThh:mm:ss.fff:zzz") }}),
    FormatFixedNow = Table.TransformColumns(FormatLocalNow,
        {{ "FixedUtcNow", each DateTimeZone.ToText(_, "yyyy-MM-
ddThh:mm:ss.fff:zzz") }}),

    // Change the table types
    FinalTable = Table.TransformColumnTypes(FormatFixedNow,
        {{ "Index", Int64.Type }, { "UtcNow", type text }, { "FixedUtcNow", type
text }})
in
    FinalTable
```

The output of this example in both Power Query Desktop and Power Query Online is:

 1 ² ₃ Index ▼	A ^B _C UtcNow ▼	A ^B _C FixedUtcNow ▼
1	1 2025-06-11T03:42:57.269:+00:00	2025-06-11T03:42:56.964:+00:00
2	2 2025-06-11T03:42:57.692:+00:00	2025-06-11T03:42:56.964:+00:00
3	3 2025-06-11T03:42:58.099:+00:00	2025-06-11T03:42:56.964:+00:00
4	4 2025-06-11T03:42:58.506:+00:00	2025-06-11T03:42:56.964:+00:00
5	5 2025-06-11T03:42:58.914:+00:00	2025-06-11T03:42:56.964:+00:00

Effects on other functions

Other Power Query M functions that depend on the current date and time can also be affected by how the local time is returned on either Power Query Desktop or Power Query Online. For example, if you use the [DateTimeZone.ToLocal](#) function to convert UTC time to local time, it still returns the UTC time on Power Query Online.

Another example is any function that can use the current system time as a parameter. These functions include [Date.Month](#), [Date.DayOfYear](#), [DateTime.IsInCurrentYear](#), [DateTimeZone.ZoneHours](#), or any other function that can evaluate the current date and time.

In all of these functions, if your logic depends on whether a value falls within the current day, hour, month, or year, the results might differ between environments. These differences between environments are especially noticeable if the query runs near a boundary (for example, just before or after midnight, the start of a new month, or a new year). If consistency is crucial across different environments, use the `DateTimeZone.UtcNow` or `DateTimeZone.FixedUtcNow` functions to retrieve the date and time.

Best practices and recommendations

Choosing the right time function in Power Query depends on your specific use case, the environment in which your query runs (desktop vs. online), and whether you need a dynamic or fixed timestamp. Here are some best practices to help guide your decision:

- **Be explicit about time zones:** Use the `DateTimeZone` functions instead of `DateTime` functions when time zone context matters. Use `DateTimeZone.UtcNow` or `DateTimeZone.FixedUtcNow` for consistency across environments, especially in cloud-based solutions like Power BI service.
- **Use fixed functions for repeatable results:** Use the fixed variants (such as `DateTimeZone.FixedUtcNow`) when you want the timestamp to remain constant across

query evaluations. This method is especially useful for logging, auditing, or capturing the time of data ingestion.

- **Avoid local functions in Power Query Online:** Functions like `DateTime.LocalNow` and `DateTimeZone.LocalNow` return UTC time in cloud-based solutions like Power BI service, which can lead to confusion or incorrect assumptions. If you need actual local time in the service, consider adjusting UTC manually using known offsets (although this adjustment can be brittle, for example, due to daylight savings time or regional settings).
- **Test in both desktop and online environments:** Always test your queries in both Power Query Desktop and Power Query Online if your logic depends on current time. This testing helps catch discrepancies early, especially for scheduled refresh scenarios.
- **Document your time logic:** Clearly comment or document why a specific time function is used, especially if you're using a workaround for time zone handling. This information helps future collaborators understand the intent behind the logic.
- **Use UTC for scheduled workflows:** For scheduled refreshes or automated pipelines, UTC is the safest and most predictable choice. It avoids ambiguity caused by daylight saving time or regional time zone shifts.
- **Cache time values when needed:** If you need to use the same timestamp across multiple steps in a query, assign it to a variable at the top of your query using a fixed function. This variable ensures consistency throughout the transformation logic.

Standard numeric format strings

Article • 10/09/2024

Standard numeric format strings are used to format common numeric types. A standard numeric format string takes the form `[format specifier][precision specifier]`, where:

- *Format specifier* is a single alphabetic character that specifies the type of number format, for example, currency or percent. Any numeric format string that contains more than one alphabetic character, including white space, is interpreted as a custom numeric format string. For more information, go to [Custom numeric format strings](#).
- *Precision specifier* is an optional integer that affects the number of digits in the resulting string. The precision specifier controls the number of digits in the string representation of a number.

When the precision specifier controls the number of fractional digits in the result string, the result string reflects a number that is rounded to a representable result nearest to the infinitely precise result.

ⓘ Note

The precision specifier determines the number of digits in the result string. To pad a result string with leading or trailing spaces or other characters (such as 0), use the [Text.PadStart](#) and [Text.PadEnd](#) functions and use the overall length `count` in these functions to pad the result string.

Standard numeric format strings are supported by the [Number.ToText](#) function.

Standard format specifiers

The following table describes the standard numeric format specifiers and displays sample output produced by each format specifier. Go to the [Notes](#) section for additional information about using standard numeric format strings, and the [Code example](#) section for a comprehensive illustration of their use.

ⓘ Note

The result of a formatted string for a specific culture might differ from the following examples. Operating system settings, user settings, environment variables, and

other settings on the system you use can all affect the format.

 Expand table

Format specifier	Name	Description	Examples
"C" or "c"	Currency	Result: A currency value. Supported by: All numeric types. Precision specifier: Number of decimal digits. Default precision specifier: Defined by the culture. More information: The Currency ("C") Format Specifier.	123.456 ("C", en-US) -> \ \$123.46 123.456 ("C", fr-FR) -> 123,46 € 123.456 ("C", ja-JP) -> ¥123 -123.456 ("C3", en-US) -> (\ \$123.456) -123.456 ("C3", fr-FR) -> -123,456 € -123.456 ("C3", ja-JP) -> -¥123.456
"D" or "d"	Decimal	Result: Integer digits with optional negative sign. Supported by: Integral types only. Precision specifier: Minimum number of digits. Default precision specifier: Minimum number of digits required. More information: The Decimal("D") Format Specifier.	1234 ("D") -> 1234 -1234 ("D6") -> -001234
"E" or "e"	Exponential (scientific)	Result: Exponential notation. Supported by: All numeric types. Precision specifier: Number of decimal digits. Default precision specifier: 6.	1052.0329112756 ("E", en-US) -> 1.052033E+003 1052.0329112756 ("e", fr-FR) -> 1,052033e+003 -1052.0329112756

Format specifier	Name	Description	Examples
		More information: The Exponential ("E") Format Specifier .	("e2", en-US) -> -1.05e+003 -1052.0329112756 ("E2", fr-FR) -> -1,05E+003
"F" or "f"	Fixed-point	Result: Integral and decimal digits with optional negative sign. Supported by: All numeric types. Precision specifier: Number of decimal digits. Default precision specifier: Defined by the culture. More information: The Fixed-Point ("F") Format Specifier .	1234.567 ("F", en-US) -> 1234.57 1234.567 ("F", de-DE) -> 1234,57 1234 ("F1", en-US) -> 1234.0 1234 ("F1", de-DE) -> 1234,0 -1234.56 ("F4", en-US) -> -1234.5600 -1234.56 ("F4", de-DE) -> -1234,5600
"G" or "g"	General	Result: The more compact of either fixed-point or scientific notation. Supported by: All numeric types. Precision specifier: Number of significant digits. Default precision specifier: Depends on numeric type. More information: The General ("G") Format Specifier .	-123.456 ("G", en-US) -> -123.456 -123.456 ("G", sv-SE) -> -123,456 123.4546 ("G4", en-US) -> 123.5 123.4546 ("G4", sv-SE) -> 123,5 -1.234567890e-25 ("G", en-US) -> -1.23456789E-25 -1.234567890e-25 ("G", sv-SE) -> -1,23456789E-25

Format specifier	Name	Description	Examples
"N" or "n"	Number	Result: Integral and decimal digits, group separators, and a decimal separator with optional negative sign. Supported by: All numeric types. Precision specifier: Desired number of decimal places. Default precision specifier: Defined by the culture. More information: The Numeric ("N") Format Specifier.	1234.567 ("N", en-US) -> 1,234.57 1234.567 ("N", ru-RU) -> 1 234,57 1234 ("N1", en-US) -> 1,234.0 1234 ("N1", ru-RU) -> 1 234,0 -1234.56 ("N3", en-US) -> -1,234.560 -1234.56 ("N3", ru-RU) -> -1 234,560
"P" or "p"	Percent	Result: Number multiplied by 100 and displayed with a percent symbol. Supported by: All numeric types. Precision specifier: Desired number of decimal places. Default precision specifier: Defined by the culture. More information: The Percent ("P") Format Specifier.	1 ("P", en-US) -> 100.00 % 1 ("P", fr-FR) -> 100,00 % -0.39678 ("P1", en-US) -> -39.7 % -0.39678 ("P1", fr-FR) -> -39,7 %
"X" or "x"	Hexadecimal	Result: A hexadecimal string. Supported by: Integral types only. Precision specifier: Number of digits in the result string. More information: The Hexadecimal ("X") Format Specifier.	255 ("X") -> FF -1 ("x") -> ff 255 ("x4") -> 00ff -1 ("X4") -> 00FF

Format specifier	Name	Description	Examples
Any other single character	Unknown specifier	Result: Throws an Expression error at run time.	

Use standard numeric format strings

A standard numeric format string can be used to define the formatting of a numeric value. It can be passed to the [Number.ToText](#) `format` parameter. The following example formats a numeric value as a currency string in the current culture (in this case, the en-US culture).

Power Query M

```
Number.ToText(123.456, "C2")
// Displays $123.46
```

Optionally, you can supply a `count` argument in the [Text.PadStart](#) and [Text.PadEnd](#) functions to specify the width of the numeric field and whether its value is right- or left-aligned. For example, the following sample left-aligns a currency value in a 28-character field, and it right-aligns a currency value in a 14-character field (when using a monospaced font).

Power Query M

```
let
    amounts = {16305.32, 18794.16},
    result = Text.Format("      Beginning Balance      Ending Balance#
(cr,lf)    #{0}#{1}",
    {
        Text.PadEnd(Number.ToText(amounts{0}, "C2"), 28),
        Text.PadStart(Number.ToText(amounts{1}, "C2"), 14)
    })
in
    result

// Displays:
//      Beginning Balance      Ending Balance
//      $16,305.32              $18,794.16
```

The following sections provide detailed information about each of the standard numeric format strings.

Currency format specifier (C)

The "C" (or currency) format specifier converts a number to a string that represents a currency amount. The precision specifier indicates the desired number of decimal places in the result string. If the precision specifier is omitted, the default number of decimal places to use in currency values is 2.

If the value to be formatted has more than the specified or default number of decimal places, the fractional value is rounded in the result string. If the value to the right of the number of specified decimal places is 5 or greater, the last digit in the result string is rounded away from zero.

The result string is affected by the formatting information of the current culture.

The following example formats a value with the currency format specifier:

Power Query M

```
let
    Source =
    {
        Number.ToText(12345.6789, "C"),
        Number.ToText(12345.6789, "C3"),
        Number.ToText(12345.6789, "C3", "da-DK")
    }
in
    Source

// The example displays the following list on a system whose
// current culture is English (United States):
//      $12,345.68
//      $12,345.679
//      12.345,679 kr.
```

Decimal format specifier (D)

The "D" (or decimal) format specifier converts a number to a string of decimal digits (0-9), prefixed by a minus sign if the number is negative. This format is supported only for integral types.

The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier. If no precision specifier is specified, the default is the minimum value required to represent the integer without leading zeros.

The result string is affected by the formatting information of the current culture.

The following example formats a value with the decimal format specifier.

Power Query M

```
let
    Source =
    {
        Number.ToText(12345, "D"),
        // Displays 12345

        Number.ToText(12345, "D8"),
        // Displays 00012345

        Number.ToText(-12345, "D"),
        // Displays -12345

        Number.ToText(-12345, "D8")
        // Displays -00012345
    }
in
    Source
```

Exponential format specifier (E)

The exponential ("E") format specifier converts a number to a string of the form "-d.ddd...E+ddd" or "-d.ddd...e+ddd", where each "d" indicates a digit (0-9). The string starts with a minus sign if the number is negative. Exactly one digit always precedes the decimal point.

The precision specifier indicates the desired number of digits after the decimal point. If the precision specifier is omitted, a default of six digits after the decimal point is used.

The case of the format specifier indicates whether to prefix the exponent with an "E" or an "e". The exponent always consists of a plus or minus sign and a minimum of three digits. The exponent is padded with zeros to meet this minimum, if required.

The result string is affected by the formatting information of the current culture.

The following example formats a value with the exponential format specifier:

Power Query M

```
let
    Source =
    {
        Number.ToText(12345.6789, "E", ""),
        // Displays 1.234568E+004
    }
in
    Source
```

```

        Number.ToText(12345.6789, "E10", ""),
        // Displays 1.2345678900E+004

        Number.ToText(12345.6789, "e4", ""),
        // 1.2346e+004

        Number.ToText(12345.6789, "E", "fr-FR")
        // Displays 1,234568E+004
    }
in
    Source

```

📌 Note

The blank text value ("") in the last parameter of [Number.ToText](#) in the previous sample refers to the invariant culture.

Fixed-point format specifier (F)

The fixed-point ("F") format specifier converts a number to a string of the form "-ddd.ddd..." where each "d" indicates a digit (0-9). The string starts with a minus sign if the number is negative.

The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default number of decimal places to use in numeric values is 2.

The result string is affected by the formatting information of the current culture.

The following example formats a double and an integer value with the fixed-point format specifier:

Power Query M

```

let
    Source =
    {
        Number.ToText(17843, "F", ""),
        // Displays 17843.00

        Number.ToText(-29541, "F3", ""),
        // Displays -29541.000

        Number.ToText(18934.1879, "F", ""),
        // Displays 18934.19

        Number.ToText(18934.1879, "F0", ""),
        // Displays 18934
    }

```

```
        Number.ToText(-1898300.1987, "F1", ""),  
        // Displays -1898300.2  
  
        Number.ToText(-1898300.1987, "F3", "es-ES")  
        // Displays -1898300,199  
    }  
in  
    Source
```

General format specifier (G)

The general ("G") format specifier converts a number to the more compact of either fixed-point or scientific notation, depending on the type of the number and whether a precision specifier is present. The precision specifier defines the maximum number of significant digits that can appear in the result string. If the precision specifier is omitted or zero, the type of the number determines the default precision, as indicated in the following table.

[Expand table](#)

Numeric type	Default precision
<code>Byte.Type</code> or <code>Int8.Type</code>	3 digits
<code>Int16.Type</code>	5 digits
<code>Int32.Type</code>	10 digits
<code>Int64.Type</code>	19 digits
<code>Single.Type</code>	9 digits
<code>Double.Type</code>	17 digits
<code>Decimal.Type</code>	15 digits

Fixed-point notation is used if the exponent that would result from expressing the number in scientific notation is greater than -5 and less than the precision specifier; otherwise, scientific notation is used. The result contains a decimal point if required, and trailing zeros after the decimal point are omitted. If the precision specifier is present and the number of significant digits in the result exceeds the specified precision, the excess trailing digits are removed by rounding.

However, if the number is a `Decimal.Type` and the precision specifier is omitted, fixed-point notation is always used and trailing zeros are preserved.

If scientific notation is used, the exponent in the result is prefixed with "E" if the format specifier is "G", or "e" if the format specifier is "g". The exponent contains a minimum of two digits. This differs from the format for scientific notation that is produced by the exponential format specifier, which includes a minimum of three digits in the exponent.

The result string is affected by the formatting information of the current culture.

The following example formats assorted floating-point values with the general format specifier:

Power Query M

```
let
    Source =
    {
        Number.ToText(12345.6789, "G", ""),
        // Displays 12345.6789

        Number.ToText(12345.6789, "G", "fr-FR"),
        // Displays 12345,6789

        Number.ToText(12345.6789, "G7", ""),
        // Displays 12345.68

        Number.ToText(.0000023, "G", ""),
        // Displays 2.3E-06

        Number.ToText(.0000023, "G", "fr-FR"),
        // Displays 2,3E-06

        Number.ToText(.0023, "G", ""),
        // Displays 0.0023

        Number.ToText(1234, "G2", ""),
        // Displays 1.2E+03

        Number.ToText(Number.PI, "G5", "")
        // Displays 3.1416
    }
in
    Source
```

Numeric format specifier (N)

The numeric ("N") format specifier converts a number to a string of the form "-d,ddd,ddd.ddd...", where "-" indicates a negative number symbol if required, "d" indicates a digit (0-9), "," indicates a group separator, and "." indicates a decimal point symbol. The precision specifier indicates the desired number of digits after the decimal

point. If the precision specifier is omitted, the number of decimal places is defined by the current culture.

The result string is affected by the formatting information of the current culture.

The following example formats assorted floating-point values with the number format specifier:

Power Query M

```
let
    Source =
    {
        Number.ToText(-12445.6789, "N", ""),
        // Displays -12,445.68

        Number.ToText(-12445.6789, "N1", "sv-SE"),
        // Displays -12 445,7

        Number.ToText(123456789, "N1", "")
        // Displays 123,456,789.0
    }
in
    Source
```

Percent format specifier (P)

The percent ("P") format specifier multiplies a number by 100 and converts it to a string that represents a percentage. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision supplied by the current culture is used.

The following example formats floating-point values with the percent format specifier:

Power Query M

```
let
    Source =
    {
        Number.ToText(.2468013, "P", ""),
        // Displays 24.68 %

        Number.ToText(.2468013, "P", "hr-HR"),
        // Displays 24,68 %

        Number.ToText(.2468013, "P1", "en-US")
        // Displays 24.7%
    }
in
    Source
```

in
Source

Hexadecimal format specifier (X)

The hexadecimal ("X") format specifier converts a number to a string of hexadecimal digits. The case of the format specifier indicates whether to use uppercase or lowercase characters for hexadecimal digits that are greater than 9. For example, use "X" to produce "ABCDEF", and "x" to produce "abcdef". This format is supported only for integral types.

The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier.

The result string isn't affected by the formatting information of the current culture.

The following example formats values with the hexadecimal format specifier.

Power Query M

```
let
    Source =
    {
        Number.ToText(0x2045e, "x"),
        // Displays 2045e

        Number.ToText(0x2045e, "X"),
        // Displays 2045E

        Number.ToText(0x2045e, "X8"),
        // Displays 0002045E

        Number.ToText(123456789, "X"),
        // Displays 75BCD15

        Number.ToText(123456789, "X2")
        // Displays 75BCD15
    }
in
    Source
```

Notes

This section contains additional information about using standard numeric format strings.

Integral and floating-point numeric types

Some descriptions of standard numeric format specifiers refer to integral or floating-point numeric types. The integral numeric types are `Byte.Type`, `Int8.Type`, `Int16.Type`, `Int32.Type`, and `Int64.Type`. The floating-point numeric types are `Decimal.Type`, `Single.Type`, and `Double.Type`.

Floating-point infinities and NaN

Regardless of the format string, if the value of a `Decimal.Type`, `Single.Type` or `Double.Type` floating-point type is positive infinity, negative infinity, or not a number (NaN), the formatted string is the value of the respective `Number.PositiveInfinity`, `Number.NegativeInfinity`, or `Number.NaN` constants specified by the currently applicable culture.

Code example

The following example formats a floating point and an integral numeric value using the en-US culture and all the standard numeric format specifiers. This example uses two particular numeric types (`Double.Type` and `Int32.Type`), but would yield similar results for any of the other numeric base types (`Byte.Type`, `Decimal.Type`, `Int8.Type`, `Int16.Type`, `Int64.Type`, and `Single.Type`).

Power Query M

```
let
    // Display text representations of numbers for en-US culture
    culture = "en-US",

    // Output floating point values
    floating = Double.From(10761.937554),
    #"Floating results" =
    {
        Text.Format("C: #{0}", {Number.ToText(floating, "C", culture)}),
        // Displays "C: $10,761.94"
        Text.Format("E: #{0}", {Number.ToText(floating, "E03", culture)}),
        // Displays "E: 1.076E+004"
        Text.Format("F: #{0}", {Number.ToText(floating, "F04", culture)}),
        // Displays "F: 10761.9376"
        Text.Format("G: #{0}", {Number.ToText(floating, "G", culture)}),
        // Displays "G: 10761.937554"
        Text.Format("N: #{0}", {Number.ToText(floating, "N03", culture)}),
        // Displays "N: 10,761.938"
        Text.Format("P: #{0}", {Number.ToText(floating/10000, "P02",
        culture)}) // Displays "P: 107.62%"
```

```

    },

    // Output integral values
    integral = Int32.From(8395),
    #"Integral results" =
    {
        Text.Format("C: #{0}", {Number.ToText(integral, "C", culture)}),
        // Displays "C: $8,395.00"
        Text.Format("D: #{0}", {Number.ToText(integral, "D6", culture)}),
        // Displays "D: 008395"
        Text.Format("E: #{0}", {Number.ToText(integral, "E03", culture)}),
        // Displays "E: 8.395E+003"
        Text.Format("F: #{0}", {Number.ToText(integral, "F01", culture)}),
        // Displays "F: 8395.0"
        Text.Format("G: #{0}", {Number.ToText(integral, "G", culture)}),
        // Displays "G: 8395"
        Text.Format("N: #{0}", {Number.ToText(integral, "N01", culture)}),
        // Displays "N: 8,395.0"
        Text.Format("P: #{0}", {Number.ToText(integral/10000, "P02",
culture)}), // Displays "P: 83.95%"
        Text.Format("X: 0x#{0}", {Number.ToText(integral, "X", culture)})
    },
    results = #"Floating results" & #"Integral results"

in
    results

```

Related content

- [How culture affects text formatting](#)
- [Number type conversion](#)
- [Data Types in Power Query](#)
- [Custom Numeric Format Strings](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Custom numeric format strings

Article • 10/04/2024

You can create a custom numeric format string, which consists of one or more custom numeric specifiers, to define how to format numeric data. A custom numeric format string is any format string that isn't a [standard numeric format string](#).

The following table describes the custom numeric format specifiers and displays sample output produced by each format specifier. Go to the [Notes](#) section for additional information about using custom numeric format strings, and the [Example](#) section for a comprehensive illustration of their use.

 Expand table

Format specifier	Name	Description	Examples
"0"	Zero placeholder	Replaces the zero with the corresponding digit if one is present; otherwise, zero appears in the result string. More information: The "0" Custom Specifier .	1234.5678 ("00000") -> 01235 0.45678 ("0.00", en-US) -> 0.46 0.45678 ("0.00", fr-FR) -> 0,46
"#"	Digit placeholder	Replaces the "#" symbol with the corresponding digit if one is present; otherwise, no digit appears in the result string. No digit appears in the result string if the corresponding digit in the input string is a nonsignificant 0. For example, 0003 ("#####") -> 3. More information: The "#" Custom Specifier .	1234.5678 ("#####") -> 1235 0.45678 ("#.###", en-US) -> .46 0.45678 ("#.###", fr-FR) -> ,46
"."	Decimal point	Determines the location of the decimal separator in the result string. More information: The "." Custom Specifier .	0.45678 ("0.00", en-US) -> 0.46 0.45678 ("0.00", fr-FR) -> 0,46

Format specifier	Name	Description	Examples
","	Group separator and number scaling	Serves as both a group separator and a number scaling specifier. As a group separator, it inserts a localized group separator character between each group. As a number scaling specifier, it divides a number by 1000 for each comma specified.	Group separator specifier: 2147483647 ("##,#", en-US) -> 2,147,483,647
		More information: The "," Custom Specifier .	2147483647 ("##,#", es-ES) -> 2.147.483.647
			Scaling specifier: 2147483647 ("#,#,,", en-US) -> 2,147
			2147483647 ("#,#,,", es-ES) -> 2.147
"%"	Percentage placeholder	Multiplies a number by 100 and inserts a localized percentage symbol in the result string.	0.3697 ("%#0.00", en-US) -> %36.97
		More information: The "%" Custom Specifier .	0.3697 ("%#0.00", el-GR) -> %36,97
			0.3697 ("##.0 %", en-US) -> 37.0 %
			0.3697 ("##.0 %", el-GR) -> 37,0 %
"‰"	Per mille placeholder	Multiplies a number by 1000 and inserts a localized per mille symbol in the result string.	0.03697 ("#0.00‰", en-US) -> 36.97‰
		More information: The "‰" Custom Specifier .	0.03697 ("#0.00‰", ru-RU) -> 36,97‰

Format specifier	Name	Description	Examples
"E0" "E+0" "E-0" "e0" "e+0" "e-0"	Exponential notation	If followed by at least one 0 (zero), formats the result using exponential notation. The case of "E" or "e" indicates the case of the exponent symbol in the result string. The number of zeros following the "E" or "e" character determines the minimum number of digits in the exponent. A plus sign (+) indicates that a sign character always precedes the exponent. A minus sign (-) indicates that a sign character precedes only negative exponents. More information: The "E" and "e" Custom Specifiers .	987654 ("#0.0e0") -> 98.8e4 1503.92311 ("0.0##e+00") -> 1.504e+03 1.8901385E-16 ("0.0e+00") -> 1.9e-16
"\, """, """"	Escape characters	Causes the next character or characters to be interpreted as a literal rather than as a custom format specifier. More information: Escape characters .	987654 ("\"###00\\#") -> #987654# 987654 ("'#'##00'#") -> #987654# 987654 ("\"\"#\"\"##00\"\"#\"\"") -> #987654#
'string' "string"	Literal string delimiter	Indicates that the enclosed characters should be copied to the result string unchanged. More information: Character literals .	68 ("# 'degrees'") -> 68 degrees 68 ("#' degrees'") -> 68 degrees
;	Section separator	Defines sections with separate format strings for positive, negative, and zero numbers. More information: The ";" Section Separator .	12.345 ("#0.0#;(#0.0#);-\\0-") -> 12.35 0 ("#0.0#;(#0.0#);-\\0-") -> -0- -12.345 ("#0.0#;(#0.0#);-\\0-") -> (12.35) 12.345 ("#0.0#;(#0.0#)") -> 12.35 0 ("#0.0#;(#0.0#)") -

Format specifier	Name	Description	Examples
			> 0.0 -12.345 ("#0.0#; (#0.0#)") -> (12.35)
Other	All other characters	The character is copied to the result string unchanged. More information: Character literals .	68 ("# °") -> 68 °

The following sections provide detailed information about each of the custom numeric format specifiers.

The "0" custom specifier

The "0" custom format specifier serves as a zero-placeholder symbol. If the value that is being formatted has a digit in the position where the zero appears in the format string, that digit is copied to the result string; otherwise, a zero appears in the result string. The position of the leftmost zero before the decimal point and the rightmost zero after the decimal point determines the range of digits that are always present in the result string.

The "00" specifier causes the value to be rounded to the nearest digit preceding the decimal, where rounding away from zero is always used. For example, formatting 34.5 with "00" would result in the value 35.

The following example displays several values that are formatted by using custom format strings that include zero placeholders.

Power Query M

```
let
    Source =
    {
        Number.ToText(123, "00000", ""),
        // Displays 00123

        Number.ToText(1.2, "0.00", ""),
        // Displays 1.20

        Number.ToText(1.2, "00.00", ""),
        // Displays 01.20

        Number.ToText(1.2, "00.00", "da-DK"),
        // Displays 01,20
    }
```

```

        Number.ToText(.56, "0.0", ""),
        // Displays 0.6

        Number.ToText(1234567890, "0,0", ""),
        // Displays 1,234,567,890

        Number.ToText(1234567890, "0,0", "el-GR"),
        // Displays 1.234.567.890

        Number.ToText(1234567890.123456, "0,0.0", ""),
        // Displays 1,234,567,890.1

        Number.ToText(1234.567890, "0,0.00", "")
        // Displays 1,234.57
    }
in
Source

```

ⓘ Note

The blank text value (""), in the last parameter of [Number.ToText](#) in the previous sample refers to the invariant culture.

[Back to table](#)

The "#" custom specifier

The "#" custom format specifier serves as a digit-placeholder symbol. If the value that is being formatted has a digit in the position where the "#" symbol appears in the format string, that digit is copied to the result string. Otherwise, nothing is stored in that position in the result string.

Note that this specifier never displays a zero that isn't a significant digit, even if zero is the only digit in the string. It displays zero only if it's a significant digit in the number that is being displayed.

The "###" format string causes the value to be rounded to the nearest digit preceding the decimal, where rounding away from zero is always used. For example, formatting 34.5 with "###" would result in the value 35.

The following example displays several values that are formatted by using custom format strings that include digit placeholders.

```

let
    Source =
    {
        Number.ToText(1.2, "#.##", ""),
        // Displays 1.2

        Number.ToText(123, "#####"),
        // Displays 123

        Number.ToText(123456, "[##-##-##]"),
        // Displays [12-34-56]

        Number.ToText(1234567890, "#"),
        // Displays 1234567890

        Number.ToText(1234567890, "(###) ###-#####")
        // Displays (123) 456-7890
    }
in
    Source

```

To return a result string in which absent digits or leading zeroes are replaced by spaces, use the [Text.PadStart](#) and specify a field width, as the following example illustrates.

Power Query M

```

let
    Source = Text.Format("The value is: '#{0} '",
    {Text.PadStart(Number.ToText(.324, "#.###"), 5)})
in
    Source

// The example displays the following output if the current culture
// is en-US:
//      The value is: ' .324'

```

[Back to table](#)

The "." custom specifier

The "." custom format specifier inserts a localized decimal separator into the result string. The first period in the format string determines the location of the decimal separator in the formatted value; any additional periods are ignored. If the format specifier ends with a "." only the significant digits are formatted into the result string.

The character that is used as the decimal separator in the result string isn't always a period; it's determined by the culture that controls formatting.

The following example uses the "." format specifier to define the location of the decimal point in several result strings.

Power Query M

```
let
    Source =
    {
        Number.ToText(1.2, "0.00", ""),
        // Displays 1.20

        Number.ToText(1.2, "00.00", ""),
        // Displays 01.20

        Number.ToText(1.2, "00.00", "da-DK"),
        // Displays 01,20

        Number.ToText(.086, "#0.##%", ""),
        // Displays 8.6%

        Number.ToText(Double.From(86000), "0.###E+0", "")
        // Displays 8.6E+4
    }
in
    Source
```

[Back to table](#)

The "," custom specifier

The "," character serves as both a group separator and a number scaling specifier.

- Group separator: If one or more commas are specified between two digit placeholders (0 or #) that format the integral digits of a number, a group separator character is inserted between each number group in the integral part of the output.

The culture determines the character used as the number group separator and the size of each number group. For example, if the string "#,#" and the invariant culture are used to format the number 1000, the output is "1,000".

- Number scaling specifier: If one or more commas are specified immediately to the left of the explicit or implicit decimal point, the number to be formatted is divided by 1000 for each comma. For example, if the string "0,," is used to format the number 100 million, the output is "100".

You can use group separator and number scaling specifiers in the same format string. For example, if the string "#,0,," and the invariant culture are used to format the number one billion, the output is "1,000".

The following example illustrates the use of the comma as a group separator.

Power Query M

```
let
    Source =
    {
        Number.ToText(1234567890, "#,#", ""),
        // Displays 1,234,567,890

        Number.ToText(1234567890, "#,##0,,", "")
        // Displays, 1,235
    }
in
    Source
```

The following example illustrates the use of the comma as a specifier for number scaling.

Power Query M

```
let
    Source =
    {
        Number.ToText(1234567890, "#,,", ""),
        // Displays 1235

        Number.ToText(1234567890, "#,,,", ""),
        // Displays 1

        Number.ToText(1234567890, "#,##0,,", "")
        // Displays 1,235
    }
in
    Source
```

[Back to table](#)

The "%" custom specifier

A percent sign (%) in a format string causes a number to be multiplied by 100 before it's formatted. The localized percent symbol is inserted in the number at the location where the % appears in the format string. The percent character used is defined by the culture.

The following example defines a custom format string that includes the "%" custom specifier.

Power Query M

```
let
    Source = Number.ToText(.086, "#0.##%", "")
    // Displays 8.6%
in
    Source
```

[Back to table](#)

The "%o" custom specifier

A per mille character (%o or \u2030) in a format string causes a number to be multiplied by 1000 before it's formatted. The appropriate per mille symbol is inserted in the returned string at the location where the %o symbol appears in the format string. The per mille character used is defined by the culture, which provides culture-specific formatting information.

The following example defines a custom format string that includes the "%o" custom specifier.

Power Query M

```
let
    Source = Number.ToText(.00354, "#0.##" & Character.FromNumber(0x2030),
    "")
    // Displays 3.54%
in
    Source
```

[Back to table](#)

The "E" and "e" custom specifiers

If any of the strings "E", "E+", "E-", "e", "e+", or "e-" are present in the format string and are followed immediately by at least one zero, the number is formatted by using scientific notation with an "E" or "e" inserted between the number and the exponent. The number of zeros following the scientific notation indicator determines the minimum number of digits to output for the exponent. The "E+" and "e+" formats indicate that a

plus sign or minus sign should always precede the exponent. The "E", "E-", "e", or "e-" formats indicate that a sign character should precede only negative exponents.

The following example formats several numeric values using the specifiers for scientific notation.

Power Query M

```
let
    Source =
    {
        Number.ToText(86000, "0.###E+0", ""),
        // Displays 8.6E+4

        Number.ToText(86000, "0.###E+000", ""),
        // Displays 8.6E+004

        Number.ToText(86000, "0.###E-000", "")
        // Displays 8.6E004
    }
in
    Source
```

[Back to table](#)

Escape characters

The "#", "0", ".", ",", "%", and "%o" symbols in a format string are interpreted as format specifiers rather than as literal characters. Depending on their position in a custom format string, the uppercase and lowercase "E" as well as the + and - symbols can also be interpreted as format specifiers.

To prevent a character from being interpreted as a format specifier, you can:

- Precede it with a backslash.
- Surround it with a single quote.
- Surround it with two double quotes.

Each of these characters acts as escape characters. The escape character signifies that the following character is a character literal that should be included in the result string unchanged.

To include a backslash in a result string, you must escape it with another backslash (\\).

To include a single quote in a result string, you must escape it with a backslash (\'). If another single quote that isn't escaped precedes the escaped single quote, the

backslash is displayed instead (`'\'` displays `\`).

To include a double quote in a result string, you must escape two of them with a backslash (`\"`).

The following example uses escape characters to prevent the formatting operation from interpreting the `"#"`, `"0"`, and `""` characters as either escape characters or format specifiers.

Power Query M

```
let
    Source =
    {
        Number.ToText(123, "\#\#\# ##0 dollars and \0\0 cents \#\#\#"),
        // Displays ### 123 dollars and 00 cents ###

        Number.ToText(123, "'###' ##0 dollars and '00' cents '###'"),
        // Displays ### 123 dollars and 00 cents ###

        Number.ToText(123, "\"###\" ##0 dollars and \"00\" cents \"###\""),
        // Displays ### 123 dollars and 00 cents ###

        Number.ToText(123, "\\\\\\\\ ##0 dollars and \\0\\0 cents \\\\\\\\"),
        // Displays \\ 123 dollars and 00 cents \\

        Number.ToText(123, "'\\\\' ##0 dollars and '00' cents '\\\\'"),
        // Displays \\ 123 dollars and 00 cents \\

        Number.ToText(123, "\"\\\\\" ##0 dollars and \"00\" cents \"\\\\\"")
        // Displays \\ 123 dollars and 00 cents \\
    }
in
    Source
```

[Back to table](#)

The ";" section separator

The semicolon (;) is a conditional format specifier that applies different formatting to a number depending on whether its value is positive, negative, or zero. To produce this behavior, a custom format string can contain up to three sections separated by semicolons. These sections are described in the following table.

[Expand table](#)

Number of sections	Description
One section	The format string applies to all values.
Two sections	The first section applies to positive values and zeros, and the second section applies to negative values. If the number to be formatted is negative, but becomes zero after rounding according to the format in the second section, the resulting zero is formatted according to the first section.
Three sections	The first section applies to positive values, the second section applies to negative values, and the third section applies to zeros. The second section can be left empty (by having nothing between the semicolons), in which case the first section applies to all nonzero values. If the number to be formatted is nonzero, but becomes zero after rounding according to the format in the first or second section, the resulting zero is formatted according to the third section.

Section separators ignore any preexisting formatting associated with a number when the final value is formatted. For example, negative values are always displayed without a minus sign when section separators are used. If you want the final formatted value to have a minus sign, you should explicitly include the minus sign as part of the custom format specifier.

The following example uses the ";" format specifier to format positive, negative, and zero numbers differently.

```

Power Query M

let
    Source =
    {
        Number.ToText(1234, "##;(##)"),
        // Displays 1234

        Number.ToText(-1234, "##;(##)"),
        // Displays (1234)

        Number.ToText(0, "##;(##);**Zero**")
        // Displays **Zero**
    }
in
    Source

```

Character literals

Format specifiers that appear in a custom numeric format string are always interpreted as formatting characters and never as literal characters. This includes the following characters:

- 0
- #
- %
- ‰
- '
- \
- ""
- .
- ,
- E or e, depending on its position in the format string.

All other characters are always interpreted as character literals and, in a formatting operation, are included in the result string unchanged. In a parsing operation, they must match the characters in the input string exactly; the comparison is case-sensitive.

The following example illustrates one common use of literal character units (in this case, thousands):

Power Query M

```
let
    Source = Number.ToText(123.8, "#,##0.0K")
    // Displays 123.8K
in
    Source
```

There are two ways to indicate that characters are to be interpreted as literal characters and not as formatting characters, so that they can be included in a result string or successfully parsed in an input string:

- By escaping a formatting character. For more information, go to [Escape characters](#).
- By enclosing the entire literal string in quotation apostrophes.

The following example uses both approaches to include reserved characters in a custom numeric format string.

Power Query M

```

let
    Source =
    {
        Number.ToText(9.3, "##.0%"),
        // Displays 9.3%

        Number.ToText(9.3, "'##'"),
        // Displays '9'

        Number.ToText(9.3, "\\##\\"),
        // Displays \\9\\

        Number.ToText(9.3, "##.0%'"),
        // Displays 9.3%

        Number.ToText(9.3, "'\\'##'\\'"),
        // Displays \\9\\

        Number.ToText(9.3, "##.0""%"""),
        // Displays 9.3%

        Number.ToText(9.3, "\"\"##\"\"")
        // Displays "9"
    }
in
    Source

```

Notes

Floating-Point infinities and NaN

Regardless of the format string, if the value of a `Decimal.Type`, `Single.Type` or `Double.Type` floating-point type is positive infinity, negative infinity, or not a number (NaN), the formatted string is the value of the respective [Number.PositiveInfinity](#), [Number.NegativeInfinity](#), or [Number.NaN](#) constants specified by the currently applicable culture.

Rounding and fixed-point format strings

For fixed-point format strings (that is, format strings that don't contain scientific notation format characters), numbers are rounded to as many decimal places as there are digit placeholders to the right of the decimal point. If the format string doesn't contain a decimal point, the number is rounded to the nearest integer. If the number has more digits than there are digit placeholders to the left of the decimal point, the extra digits are copied to the result string immediately before the first digit placeholder.

[Back to table](#)

Example

The following example demonstrates two custom numeric format strings. In both cases, the digit placeholder (**#**) displays the numeric data, and all other characters are copied to the result string.

Power Query M

```
let
    Source =
    {
        Number.ToText(1234567890, "(###) ###-####"),
        // Displays (123) 456-7890

        Number.ToText(42, "My Number = #")
        // Displays My number = 42
    }
in
    Source
```

[Back to table](#)

Related content

- [How culture affects text formatting](#)
- [Number type conversion](#)
- [Data Types in Power Query](#)
- [Standard Numeric Format Strings](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Standard date and time format strings

Article • 10/11/2024

A standard date and time format string uses a single character as the format specifier to define the text representation of a time and date value. Any date and time format string that contains more than one character, including white space, is interpreted as a [custom date and time format string](#). A standard or custom format string can be used to define the text representation that results from a formatting operation.

Table of format specifiers

The following table describes the standard date and time format specifiers.

 Expand table

Format specifier	Description	Examples
"d"	Short date pattern.	2009-06-15T13:45:30 -> 6/15/2009 (en-US)
	More information: The short date ("d") format specifier .	2009-06-15T13:45:30 -> 15/06/2009 (fr-FR)
		2009-06-15T13:45:30 -> 2009/06/15 (ja-JP)
"D"	Long date pattern.	2009-06-15T13:45:30 -> Monday, June 15, 2009 (en-US)
	More information: The long date ("D") format specifier .	2009-06-15T13:45:30 -> понедельник, 15 июня 2009 г. (ru-RU)
		2009-06-15T13:45:30 -> Montag, 15. Juni 2009 (de-DE)
"f"	Full date/time pattern (short time).	2009-06-15T13:45:30 -> Monday, June 15, 2009 1:45 PM (en-US)
	More information: The full date short time ("f") format specifier .	2009-06-15T13:45:30 -> den 15 juni 2009 13:45 (sv-SE)
		2009-06-15T13:45:30 -> Δευτέρα, 15 Ιουνίου 2009 1:45 μμ (el-GR)
"F"	Full date/time pattern (long time).	2009-06-15T13:45:30 -> Monday, June 15, 2009 1:45:30 PM (en-US)
		2009-06-15T13:45:30 -> den 15 juni 2009

Format specifier	Description	Examples
	More information: The full date long time ("F") format specifier.	13:45:30 (sv-SE) 2009-06-15T13:45:30 -> Δευτέρα, 15 Ιουνίου 2009 1:45:30 μμ (el-GR)
"g"	General date/time pattern (short time). More information: The general date short time ("g") format specifier.	2009-06-15T13:45:30 -> 6/15/2009 1:45 PM (en-US) 2009-06-15T13:45:30 -> 15/06/2009 13:45 (es-ES) 2009-06-15T13:45:30 -> 2009/6/15 13:45 (zh-CN)
"G"	General date/time pattern (long time). More information: The general date long time ("G") format specifier.	2009-06-15T13:45:30 -> 6/15/2009 1:45:30 PM (en-US) 2009-06-15T13:45:30 -> 15/06/2009 13:45:30 (es-ES) 2009-06-15T13:45:30 -> 2009/6/15 13:45:30 (zh-CN)
"M", "m"	Month/day pattern. More information: The month ("M", "m") format specifier.	2009-06-15T13:45:30 -> June 15 (en-US) 2009-06-15T13:45:30 -> 15. juni (da-DK) 2009-06-15T13:45:30 -> 15 Juni (id-ID)
"O", "o"	round-trip date/time pattern. More information: The round-trip ("O", "o") format specifier.	2009-06-15T13:45:30 (Local) --> 2009-06-15T13:45:30.0000000-07:00 2009-06-15T13:45:30 (Utc) --> 2009-06-15T13:45:30.0000000+00:00 2009-06-15T13:45:30 (Unspecified) --> 2009-06-15T13:45:30.0000000
"R", "r"	RFC1123 pattern. More information: The RFC1123 ("R", "r") format specifier.	2009-06-15T13:45:30 -> Mon, 15 Jun 2009 13:45:30 GMT
"s"	Sortable date/time pattern. More information: The sortable ("s") format specifier.	2009-06-15T13:45:30 (Local) -> 2009-06-15T13:45:30

Format specifier	Description	Examples
		2009-06-15T13:45:30 (Utc) -> 2009-06-15T13:45:30
"t"	Short time pattern. More information: The short time ("t") format specifier.	2009-06-15T13:45:30 -> 1:45 PM (en-US) 2009-06-15T13:45:30 -> 13:45 (hr-HR) 2009-06-15T13:45:30 -> 01:45 م (ar-EG)
"T"	Long time pattern. More information: The long time ("T") format specifier.	2009-06-15T13:45:30 -> 1:45:30 PM (en-US) 2009-06-15T13:45:30 -> 13:45:30 (hr-HR) 2009-06-15T13:45:30 -> 01:45:30 م (ar-EG)
"u"	Universal sortable date/time pattern. More information: The universal sortable ("u") format specifier.	2009-06-15T13:45:30 -> 2009-06-15 13:45:30Z
"Y", "y"	Year month pattern. More information: The year month ("Y") format specifier.	2009-06-15T13:45:30 -> June 2009 (en-US) 2009-06-15T13:45:30 -> juni 2009 (da-DK) 2009-06-15T13:45:30 -> Juni 2009 (id-ID)
Any other single character	Unknown specifier.	Throws a run-time expression error.

How standard format strings work

In a formatting operation, a standard format string is simply an alias for a custom format string. The advantage of using an alias to refer to a custom format string is that, although the alias remains invariant, the custom format string itself can vary. This is important because the string representations of date and time values typically vary by culture. For example, the "d" standard format string indicates that a date and time value is to be displayed using a short date pattern. For the invariant culture, this pattern is "MM/dd/yyyy". For the fr-FR culture, it is "dd/MM/yyyy". For the ja-JP culture, it is "yyyy/MM/dd".

If a standard format string in a formatting operation maps to a particular culture's custom format string, your application can define the specific culture whose custom format strings are used in one of these ways:

- You can use the default (or current) culture. The following example displays a date using the current culture's short date format. In this case, the current culture is en-US.

```
powerquery

let
    Source =
    {
        Date.ToText(#date(2024, 3, 15), [Format = "d"])
        //Displays 3/15/2024
    }
in
    Source
```

- You can pass a culture used to format the date according to the rules of that specific culture. The following example displays a date using the short date format of the pt-BR culture.

```
Power Query M

let
    Source =
    {
        Date.ToText(#date(2024, 3, 15), [Format = "d", Culture = "pt-
BR"])
        //Displays 15/03/2024
    }
in
    Source
```

In some cases, the standard format string serves as a convenient abbreviation for a longer custom format string that is invariant. Four standard format strings fall into this category: "O" (or "o"), "R" (or "r"), "s", and "u". These strings correspond to custom format strings defined by the invariant culture. They produce string representations of date and time values that are intended to be identical across cultures. The following table provides information on these four standard date and time format strings.

Standard format string	Defined by	Custom format string
"O" or "o"	None	yyyy'-'MM'-'dd'T'HH':'mm':'ss'.fffffffK
"R" or "r"	IETF RFC 1123 specification	ddd, dd MMM yyyy HH':'mm':'ss 'GMT'
"s"	ISO 8601	yyyy'-'MM'-'dd'T'HH':'mm':'ss
"u"	Sortable because it uses leading zeros for year, month, day, hour, minute, and second	yyyy'-'MM'-'dd HH':'mm':'ss'Z'

Standard format strings can also be used in parsing operations, which require an input string to exactly conform to a particular pattern for the parse operation to succeed. Many standard format strings map to multiple custom format strings, so a date and time value can be represented in a variety of formats and the parse operation still succeeds.

The following sections describe the standard format specifiers for [Date](#), [DateTime](#), [DateTimeZone](#), and [Time](#) values.

Date formats

This group includes the following formats:

- [The short date \("d"\) format specifier](#)
- [The long date \("D"\) format specifier](#)

The short date ("d") format specifier

The "d" standard format specifier represents a custom date format defined by a specific culture. For example, the custom format text returned by the invariant culture is "MM/dd/yyyy".

The following example uses the "d" format specifier to display a date value.

Power Query M

```
let
    Source =
    {
        Date.ToText(#date(2024, 4, 10), [Format = "d", Culture = ""]),
        // Displays 04/10/2024

        Date.ToText(#date(2024, 4, 10), [Format = "d", Culture = "en-US"]),
```

```

// Displays 4/10/2024

Date.ToText(#date(2024, 4, 10), [Format = "d", Culture = "en-NZ"]),
// Displays 10/4/2024

Date.ToText(#date(2024, 4, 10), [Format = "d", Culture = "de-DE"])
// Displays 10.4.2024
}
in
Source

```

[Back to table](#)

The long date ("D") format specifier

The "D" standard format specifier represents a custom date format defined by a specific culture. For example, the custom format string for the invariant culture is "dddd, dd MMMM yyyy".

The following example uses the "D" format specifier to display a date and time value.

Power Query M

```

let
    Source =
    {
        Date.ToText(#date(2024, 4, 10), [Format = "D", Culture = ""]),
        // Displays Wednesday, 10 April, 2024

        Date.ToText(#date(2024, 4, 10), [Format = "D", Culture = "en-US"]),
        // Displays Wednesday, April 10, 2024

        Date.ToText(#date(2024, 4, 10), [Format = "D", Culture = "pt-BR"]),
        // Displays quarta-feira, 10 de abril de 2024

        Date.ToText(#date(2024, 4, 10), [Format = "D", Culture = "es-MX"])
        // Displays miércoles, 10 de abril de 2024
    }
in
Source

```

[Back to table](#)

Date and time formats

This group includes the following formats:

- [The full date short time \("f"\) format specifier](#)

- The full date long time ("F") format specifier
- The general date short time ("g") format specifier
- The general date long time ("G") format specifier
- The round-trip ("O", "o") format specifier
- The RFC1123 ("R", "r") format specifier
- The sortable ("s") format specifier
- The universal sortable ("u") format specifier

The full date short time ("f") format specifier

The "f" standard format specifier represents a combination of the long date ("D") and short time ("t") patterns, separated by a space.

The result string is affected by the formatting information of a specific culture.

The following example uses the "f" format specifier to display a date and time value.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "f",
        Culture = "en-US"]),
        // Displays Wednesday, April 10, 2024 6:30 AM

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "f",
        Culture = "fr-FR"])
        // Displays mercredi 10 avril 2024 06:30
    }
in
    Source
```

[Back to table](#)

The full date long time ("F") format specifier

The "F" standard format specifier represents a custom date and time format defined by a specific culture. For example, the custom format string for the invariant culture is "dddd, dd MMMM yyyy HH:mm:ss".

The result string is affected by the formatting information of a specific culture.

The following example uses the "F" format specifier to display a date and time value.

Power Query M

```

let
    Source =
    {
        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "F",
Culture = ""]),
        // Displays Wednesday, 10 April, 2024 06:30:00

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "F",
Culture = "en-US"]),
        // Displays Wednesday, April 10, 2024 6:30:00 AM

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "F",
Culture = "fr-FR"])
        // Displays mercredi 10 avril 2024 06:30:00
    }
in
    Source

```

[Back to table](#)

The general date short time ("g") format specifier

The "g" standard format specifier represents a combination of the short date ("d") and short time ("t") patterns, separated by a space. The resulting text is affected by the formatting information of a specific culture.

The result string is affected by the formatting information of a specific culture.

The following example uses the "g" format specifier to display a date and time value.

Power Query M

```

let
    Source =
    {
        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "g",
Culture = ""]),
        // Displays 04/10/2024 06:30

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "g",
Culture = "en-US"]),
        // Displays 4/10/2024 6:30 AM

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "g",
Culture = "fr-BE"])
        // Displays 10-04-24 06:30
    }
in
    Source

```

[Back to table](#)

The general date long time ("G") format specifier

The "G" standard format specifier represents a combination of the short date ("d") and long time ("T") patterns, separated by a space. The resulting text is affected by the formatting information of a specific culture.

The result string is affected by the formatting information of a specific culture.

The following example uses the "G" format specifier to display a date and time value.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "G",
        Culture = ""]),
        // Displays 04/10/2024 06:30:00

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "G",
        Culture = "en-US"]),
        // Displays 4/10/2024 6:30:00 AM

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "G",
        Culture = "nl-BE"])
        // Displays 10/04/2024 6:30:00
    }
in
    Source
```

[Back to table](#)

The round-trip ("O", "o") format specifier

The "O" or "o" standard format specifier represents a custom date and time format string using a pattern that preserves time zone information and emits a result string that complies with ISO 8601. For **DateTimeZone** values, this format specifier is designed to preserve date, time, and timezone values in text.

The "O" or "o" standard format specifier corresponds to the "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.fffffffxxx" custom format string for **DateTimeZone** values. In this text, the pairs of single quotation marks that delimit individual characters, such as the hyphens, the colons, and the letter "T", indicate that the individual character is a literal that can't be changed. The apostrophes don't appear in the output string.

The "O" or "o" standard format specifier (and the "yyyy'-MM'-
'dd'T'HH':mm':ss'.ffffffxxx" custom format) takes advantage of the ways that ISO 8601 represents time zone information to preserve the **DateTimeZone** values:

- The time zone component of [DateTimeZone.ToLocal](#) date and time values is an offset from UTC (for example, +01:00, -07:00).
- The time zone component of [DateTimeZone.ToUtc](#) date and time values uses +00.00 to represent UTC.

Because the "O" or "o" standard format specifier conforms to an international standard, the formatting or parsing operation that uses the specifier always uses the invariant culture and the Gregorian calendar.

The following example uses the "o" format specifier to display a series of **DateTimeZone** values on a system in the U.S. Pacific Time zone.

Power Query M

```
let
    date1 = #datetimezone(2024, 6, 15, 13, 45, 30, 0, 0),
    Source =
    {
        Text.Format("#{0} (Unspecified) --> #{1}", {date1,
        DateTimeZone.ToText(date1, [Format = "O"])}),
        Text.Format("#{0} (Utc) --> #{1}", {date1,
        DateTimeZone.ToText(DateTimeZone.ToUtc(date1), [Format = "O"])}),
        Text.Format("#{0} (Local) --> #{1}", {date1,
        DateTimeZone.ToText(DateTimeZone.ToLocal(date1), [Format = "O"])}))
    }
in
    Source

// The example displays the following output:
//      6/15/2024 1:45:30 PM +00:00 (Unspecified) --> 2024-06-
15T13:45:30.0000000+00:00
//      6/15/2024 1:45:30 PM +00:00 (Utc) --> 2024-06-
15T13:45:30.0000000+00:00
//      6/15/2024 1:45:30 PM +00:00 (Local) --> 2024-06-15T08:45:30.0000000-
07:00
```

ⓘ Note

The value returned by [DateTimeZone.ToLocal](#) depends on whether you're running Power Query on a local machine or online. For example, in the sample above on a system in the U.S. Pacific Time zone, Power Query Desktop returns **-07:00** for the

Local time because it's reading the time set on your local machine. However, Power Query Online returns `+00:00` because it's reading the time set on the cloud virtual machines, which are set to UTC.

The following example uses the "o" format specifier to create a formatted string, and then restores the original date and time value by calling a date and time parsing routine.

Power Query M

```
let
    // Round-trip a local time
    #"Origin Local Date" = DateTimeZone.ToLocal(
        #datetimezone(2024, 4, 10, 6, 30, 0, 0, 0)
    ),
    #"Local Date Text" = DateTimeZone.ToText(
        #"Origin Local Date", [Format = "o"]
    ),
    #"New Local Date" = DateTimeZone.FromText(#"Local Date Text"),
    #"Local Round Trip" = Text.Format(
        "Round-tripped #{0} Local to #{1} Local.",
        {
            DateTimeZone.ToText(#"Origin Local Date"),
            DateTimeZone.ToText(#"New Local Date")
        }
    ),

    // Round-trip a UTC time
    #"Origin UTC Date" = DateTimeZone.ToUtc(
        #datetimezone(2024, 4, 12, 9, 30, 0, 0, 0)
    ),
    #"UTC Date Text" = DateTimeZone.ToText(
        #"Origin UTC Date", [Format = "o"]
    ),
    #"New UTC Date" = DateTimeZone.FromText(#"UTC Date Text"),
    #"UTC Round Trip" = Text.Format(
        "Round-tripped #{0} UTC to #{1} UTC.",
        {
            DateTimeZone.ToText(#"Origin UTC Date"),
            DateTimeZone.ToText(#"New UTC Date")
        }
    ),

    // Round-trip an 18 hour offset time
    #"Origin Offset Date" = DateTimeZone.ToLocal(
        #datetimezone(2024, 4, 10, 6, 30, 0, 0, 0) + #duration(0, 18, 0, 0)
    ),
    #"Offset Date Text" = DateTimeZone.ToText(
        #"Origin Offset Date", [Format = "o"]
    ),
    #"New Offset Date" = DateTimeZone.FromText(#"Offset Date Text"),
    #"Offset Round Trip" = Text.Format(
        "Round-tripped #{0} to #{1}.",
```

```

        {
            DateTimeZone.ToText("#Origin Offset Date"),
            DateTimeZone.ToText("#New Offset Date")
        }
    ),

    #"Round Trip Results" =
        {"#Local Round Trip", "#UTC Round Trip", "#Offset Round Trip"}
in
    #"Round Trip Results"

// The example displays the following output in Power Query Desktop:
//     Round-tripped 4/9/2024 11:30:00 PM -07:00 Local to 4/9/2024 11:30:00
//     PM -07:00 Local.
//     Round-tripped 4/12/2024 9:30:00 AM +00:00 UTC to 4/12/2024 9:30:00 AM
//     +00:00 UTC.
//     Round-tripped 4/10/2024 5:30:00 PM -07:00 to 4/10/2024 5:30:00 PM
//     -07:00.

// The example displays the following output in Power Query Online:
//     Round-tripped 4/10/2024 6:30:00 AM +00:00 Local to 4/10/2024 6:30:00
//     AM +00:00 Local.
//     Round-tripped 4/12/2024 9:30:00 AM +00:00 UTC to 4/12/2024 9:30:00 AM
//     +00:00 UTC.
//     Round-tripped 4/11/2024 12:30:00 AM +00:00 to 4/11/2024 12:30:00 AM
//     +00:00.

```

[Back to table](#)

The RFC1123 ("R", "r") format specifier

The "R" or "r" standard format specifier represents a custom date and time format string that's not defined by a specific culture. It is always the same, regardless of the culture used or the format provider supplied. The custom format string is "ddd, dd MMM yyyy HH':'mm':'ss 'GMT'". When this standard format specifier is used, the formatting or parsing operation always uses the invariant culture.

Although the RFC 1123 standard expresses a time as Coordinated Universal Time (UTC), the formatting operation doesn't modify the value of the date and time that's being formatted. Therefore, you must convert the DateTime value to UTC by calling the [DateTimeZone.ToUtc](#) function method before you perform the formatting operation.

The following example uses the "r" format specifier to display a time and date value on a system in the U.S. Pacific Time zone (seven hours behind UTC).

Power Query M

```

let
    date1 = #datetimezone(2024, 4, 10, 6, 30, 0, -7, 0),

```



```

dateOffset = DateTimeZone.ToUtc(date1),
Source =
{
    DateTimeZone.ToText(date1, [Format = "r"]),
    // Displays Wed, 10 Apr 2024 13:30:00 GMT

    DateTimeZone.ToText(dateOffset, [Format = "r"])
    // Displays Wed, 10 Apr 2024 13:30:00 GMT
}
in
    Source

```

[Back to table](#)

The sortable ("s") format specifier

The "s" standard format specifier represents a custom date and time format string that reflects a defined standard (ISO 8601), and is read-only. Therefore, it is always the same, regardless of the culture used or the format provider supplied. The custom format string is "yyyy'-'MM'-'dd'T'HH':'mm':'ss". The purpose of the "s" format specifier is to produce result strings that sort consistently in ascending or descending order based on date and time values.

When this standard format specifier is used, the formatting or parsing operation always uses the invariant culture.

The following example uses the "s" format specifier to display a date and time value on a system in the U.S. Pacific Time zone.

Power Query M

```

let
    Source =
    {
        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "s",
Culture = "en-US"])
        // Displays 2024-04-10T06:30:00
    }
in
    Source

```

[Back to table](#)

The universal sortable ("u") format specifier

The "u" standard format specifier represents a custom date and time format string that is always the same, regardless of the culture used or the format provider supplied. The custom format string is "yyyy'-MM'-'dd HH':'mm':'ss'Z'". The pattern reflects a defined standard, and the property is read-only. When this standard format specifier is used, the formatting or parsing operation always uses the invariant culture.

Although the result string should express a time as Coordinated Universal Time (UTC), no conversion of the original **DateTimeZone** value is performed during the formatting operation. Therefore, you must convert a **DateTimeZone** value to UTC by calling the [DateTimeZone.ToUtc](#) function before formatting it.

The following example uses the "u" format specifier to display a date and time value.

Power Query M

```
let
    date1 = #datetimezone(2024, 4, 10, 6, 30, 0, -7, 0),
    dateOffset = DateTimeZone.ToUtc(date1),
    Source =
    {
        DateTimeZone.ToText(dateOffset, [Format = "u"]),
        // Displays 2024-04-10 13:30:00Z
    }
in
    Source
```

[Back to table](#)

Time formats

This group includes the following formats:

- [The short time \("t"\) format specifier](#)
- [The long time \("T"\) format specifier](#)

The short time ("t") format specifier

The "t" standard format specifier represents a custom date and time format string that is defined by the specified culture. For example, the custom format string for the invariant culture is "HH:mm".

The result string is affected by the formatting information of a specific culture.

The following example uses the "t" format specifier to display a date and time value.

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "t",
Culture = ""]),
        // Displays 06:30

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "t",
Culture = "en-US"]),
        // Displays 6:30 AM

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "t",
Culture = "es-ES"])
        // Displays 6:30
    }
in
    Source
```

[Back to table](#)

The long time ("T") format specifier

The "T" standard format specifier represents a custom date and time format string that is defined by the specific culture. For example, the custom format string for the invariant culture is "HH:mm:ss".

The following example uses the "T" format specifier to display a date and time value.

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "T",
Culture = ""]),
        // Displays 06:30:00

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "T",
Culture = "en-US"]),
        // Displays 6:30:00 AM

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "T",
Culture = "es-ES"])
        // Displays 6:30:00
    }
in
    Source
```

[Back to table](#)

Partial date formats

This group includes the following formats:

- [The month \("M", "m"\) format specifier](#)
- [The year month \("Y", "y"\) format specifier](#)

The month ("M", "m") format specifier

The "M" or "m" standard format specifier represents a custom date and time format string that is defined by the specific culture. For example, the custom format string for the invariant culture is "MMMM dd".

The following example uses the "m" format specifier to display a date and time value.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "m",
        Culture = ""]),
        // Displays April 10

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "m",
        Culture = "en-US"]),
        // Displays April 10

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "m",
        Culture = "ms-MY"])
        // Displays 10 April
    }
in
    Source
```

[Back to table](#)

The year month ("Y", "y") format specifier

The "Y" or "y" standard format specifier represents a custom date format string that is defined by a specific culture. For example, the custom format string for the invariant culture is "yyyy MMMM".

The following example uses the "y" format specifier to display a date and time value.

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "Y",
Culture = ""]),
        // Displays 2024 April

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "Y",
Culture = "en-US"]),
        // Displays April 2024

        DateTime.ToText(#datetime(2024, 4, 10, 6, 30, 0), [Format = "y",
Culture = "af-ZA"])
        // Displays April 2024
    }
in
    Source
```

[Back to table](#)

Related content

- [How culture affects text formatting](#)
- [Date, Time, DateTime, and DateTimeZone type conversion](#)
- [Date functions](#)
- [DateTime functions](#)
- [DateTimeZone functions](#)
- [Time functions](#)
- [Custom Date and Time Format Strings](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Custom date and time format strings

Article • 10/07/2024

A date and time format string defines the text representation of a [Date](#), [DateTime](#), [DateTimeZone](#), or [Time](#) value that results from a formatting operation. It can also define the representation of a date and time value that is required in a parsing operation in order to successfully convert the string to a date and time. A custom format string consists of one or more custom date and time format specifiers. Any string that isn't a [standard date and time format string](#) is interpreted as a custom date and time format string.

In formatting operations, custom date and time format strings can be used with the `ToText` method of a date and time and timezone instance. The following example illustrates its uses.

Power Query M

```
let
    Source =
        {
            Text.From("Today is " & Date.ToText(#date(2011, 6, 10), [Format =
                "MMMM dd yyyy"]]) & "."),
            Text.Format("The current date and time: #{0}", {DateTimeZone.ToText(
                #datetimezone(2011, 6, 10, 15, 24, 16, 0, 0), [Format =
                "MM/dd/yy H:mm:ss zzz"]})}
        )
    in
        Source

// The example displays the following output:
//   Today is June 10, 2011.
//   The current date and time: 06/10/11 15:24:16 +00:00
```

In parsing operations, custom date and time format strings can be used with the **Date**, **DateTime**, **Time**, and **DateTimeZone** functions. These functions require that an input string conforms exactly to a particular pattern for the parse operation to succeed. The following example illustrates a call to the [DateTime.FromText](#) function to parse a date that must include a month, a day, and a two-digit year.

Power Query M

```
let
    dateValues = { "30-12-2011", "12-30-2011", "30-12-11", "12-30-11"},
    pattern = "MM-dd-yy",
    convertedDates = List.Transform(dateValues, (dateValue) =>
```

```

        try Text.Format("Converted '#{0}' to #{1}.", {dateValue,
DateTime.FromText(dateValue, [Format=pattern])})
        otherwise Text.Format("Unable to convert '#{0}' to a date and
time.", {dateValue}))
in
    convertedDates

// The example displays the following output:
//     Unable to convert '30-12-2011' to a date and time.
//     Unable to convert '12-30-2011' to a date and time.
//     Unable to convert '30-12-11' to a date and time.
//     Converted '12-30-11' to 12/30/2011.

```

The following table describes the custom date and time format specifiers and displays a result string produced by each format specifier. By default, result strings reflect the formatting conventions of the en-US culture. If a particular format specifier produces a localized result string, the example also notes the culture to which the result string applies. For more information about using custom date and time format strings, go to the [Notes](#) section.

 Expand table

Format specifier	Description	Examples
"d"	The day of the month, from 1 to 31. More information: The "d" custom format specifier.	2009-06-01T13:45:30 -> 1 2009-06-15T13:45:30 -> 15
"dd"	The day of the month, from 01 to 31. More information: The "dd" custom format specifier.	2009-06-01T13:45:30 -> 01 2009-06-15T13:45:30 -> 15
"ddd"	The abbreviated name of the day of the week. More information: The "ddd" custom format specifier.	2009-06-15T13:45:30 -> Mon (en-US) 2009-06-15T13:45:30 -> Пн (ru-RU) 2009-06-15T13:45:30 -> lun. (fr-FR)
"dddd"	The full name of the day of the week. More information: The "dddd" custom format specifier.	2009-06-15T13:45:30 -> Monday (en-US) 2009-06-15T13:45:30 -> понедельник (ru-RU) 2009-06-15T13:45:30 -> lundi (fr-FR)

Format specifier	Description	Examples
"f"	The tenths of a second in a date and time value. More information: The "f" custom format specifier.	2009-06-15T13:45:30.6170000 -> 6 2009-06-15T13:45:30.05 -> 0
"ff"	The hundredths of a second in a date and time value. More information: The "ff" custom format specifier.	2009-06-15T13:45:30.6170000 -> 61 2009-06-15T13:45:30.0050000 -> 00
"fff"	The milliseconds in a date and time value. More information: The "fff" custom format specifier.	6/15/2009 13:45:30.617 -> 617 6/15/2009 13:45:30.0005 -> 000
"ffff"	The ten thousandths of a second in a date and time value. More information: The "ffff" custom format specifier.	2009-06-15T13:45:30.6175000 -> 6175 2009-06-15T13:45:30.0000500 -> 0000
"fffff"	The hundred thousandths of a second in a date and time value. More information: The "fffff" custom format specifier.	2009-06-15T13:45:30.6175400 -> 61754 6/15/2009 13:45:30.000005 -> 00000
"ffffff"	The millionths of a second in a date and time value. More information: The "ffffff" custom format specifier.	2009-06-15T13:45:30.6175420 -> 617542 2009-06-15T13:45:30.0000005 -> 000000
"fffffff"	The ten millionths of a second in a date and time value. More information: The "fffffff" custom format specifier.	2009-06-15T13:45:30.6175425 -> 6175425 2009-06-15T13:45:30.0001150 -> 0001150
"F"	If non-zero, the tenths of a second in a date and time value. More information: The "F" custom format specifier.	2009-06-15T13:45:30.6170000 -> 6 2009-06-15T13:45:30.0500000 -> (no output)

Format specifier	Description	Examples
"FF"	If non-zero, the hundredths of a second in a date and time value. More information: The "FF" custom format specifier.	2009-06-15T13:45:30.6170000 -> 61 2009-06-15T13:45:30.0050000 -> (no output)
"FFF"	If non-zero, the milliseconds in a date and time value. More information: The "FFF" custom format specifier.	2009-06-15T13:45:30.6170000 -> 617 2009-06-15T13:45:30.0005000 -> (no output)
"FFFF"	If non-zero, the ten thousandths of a second in a date and time value. More information: The "FFFF" custom format specifier.	2009-06-15T13:45:30.5275000 -> 5275 2009-06-15T13:45:30.0000500 -> (no output)
"FFFFF"	If non-zero, the hundred thousandths of a second in a date and time value. More information: The "FFFFF" custom format specifier.	2009-06-15T13:45:30.6175400 -> 61754 2009-06-15T13:45:30.0000050 -> (no output)
"FFFFFF"	If non-zero, the millionths of a second in a date and time value. More information: The "FFFFFF" custom format specifier.	2009-06-15T13:45:30.6175420 -> 617542 2009-06-15T13:45:30.0000005 -> (no output)
"FFFFFFF"	If non-zero, the ten millionths of a second in a date and time value. More information: The "FFFFFFF" custom format specifier.	2009-06-15T13:45:30.6175425 -> 6175425 2009-06-15T13:45:30.0001150 -> 000115
"g", "gg"	The period or era. More information: The "g" or "gg" custom format specifier.	2009-06-15T13:45:30.6170000 -> A.D.
"h"	The hour, using a 12-hour clock from 1 to 12. More information: The "h" custom format specifier.	2009-06-15T01:45:30 -> 1 2009-06-15T13:45:30 -> 1

Format specifier	Description	Examples
"hh"	The hour, using a 12-hour clock from 01 to 12. More information: The "hh" custom format specifier.	2009-06-15T01:45:30 -> 01 2009-06-15T13:45:30 -> 01
"H"	The hour, using a 24-hour clock from 0 to 23. More information: The "H" custom format specifier.	2009-06-15T01:45:30 -> 1 2009-06-15T13:45:30 -> 13
"HH"	The hour, using a 24-hour clock from 00 to 23. More information: The "HH" custom format specifier.	2009-06-15T01:45:30 -> 01 2009-06-15T13:45:30 -> 13
"K"	Time zone information. More information: The "K" custom format specifier.	2009-06-15T13:45:30, Unspecified -> 2009-06-15T13:45:30, Utc -> +00:00 2009-06-15T13:45:30, Local -> -07:00 (depends on local or cloud computer settings)
"m"	The minute, from 0 to 59. More information: The "m" custom format specifier.	2009-06-15T01:09:30 -> 9 2009-06-15T13:29:30 -> 29
"mm"	The minute, from 00 to 59. More information: The "mm" custom format specifier.	2009-06-15T01:09:30 -> 09 2009-06-15T01:45:30 -> 45
"M"	The month, from 1 to 12. More information: The "M" custom format specifier.	2009-06-15T13:45:30 -> 6
"MM"	The month, from 01 to 12. More information: The "MM" custom format specifier.	2009-06-15T13:45:30 -> 06
"MMM"	The abbreviated name of the month.	2009-06-15T13:45:30 -> Jun (en-US)

Format specifier	Description	Examples
	More information: The "MMM" custom format specifier.	2009-06-15T13:45:30 -> juin (fr-FR) 2009-06-15T13:45:30 -> Jun (zu-ZA)
"MMMM"	The full name of the month. More information: The "MMMM" custom format specifier.	2009-06-15T13:45:30 -> June (en-US) 2009-06-15T13:45:30 -> juni (da-DK) 2009-06-15T13:45:30 -> Juni (zu-ZA)
"s"	The second, from 0 to 59. More information: The "s" custom format specifier.	2009-06-15T13:45:09 -> 9
"ss"	The second, from 00 to 59. More information: The "ss" custom format specifier.	2009-06-15T13:45:09 -> 09
"t"	The first character of the AM/PM designator. More information: The "t" custom format specifier.	2009-06-15T13:45:30 -> P (en-US) 2009-06-15T13:45:30 -> 午 (ja-JP) 2009-06-15T13:45:30 -> (fr-FR)
"tt"	The AM/PM designator. More information: The "tt" custom format specifier.	2009-06-15T13:45:30 -> PM (en-US) 2009-06-15T13:45:30 -> 午後 (ja-JP) 2009-06-15T13:45:30 -> (fr-FR)
"y"	The year, from 0 to 99. More information: The "y" custom format specifier.	0001-01-01T00:00:00 -> 1 0900-01-01T00:00:00 -> 0 1900-01-01T00:00:00 -> 0 2009-06-15T13:45:30 -> 9 2019-06-15T13:45:30 -> 19
"yy"	The year, from 00 to 99. More information: The "yy" custom format specifier.	0001-01-01T00:00:00 -> 01 0900-01-01T00:00:00 -> 00 1900-01-01T00:00:00 -> 00

Format specifier	Description	Examples
		2019-06-15T13:45:30 -> 19
"yyy"	The year, with a minimum of three digits. More information: The "yyy" custom format specifier.	0001-01-01T00:00:00 -> 001 0900-01-01T00:00:00 -> 900 1900-01-01T00:00:00 -> 1900 2009-06-15T13:45:30 -> 2009
"yyyy"	The year as a four-digit number. More information: The "yyyy" custom format specifier.	0001-01-01T00:00:00 -> 0001 0900-01-01T00:00:00 -> 0900 1900-01-01T00:00:00 -> 1900 2009-06-15T13:45:30 -> 2009
"yyyyy"	The year as a five-digit number. More information: The "yyyyy" custom format specifier.	0001-01-01T00:00:00 -> 00001 2009-06-15T13:45:30 -> 02009
"z"	Hours offset from UTC, with no leading zeros. More information: The "z" custom format specifier.	2009-06-15T13:45:30-07:00 -> -7
"zz"	Hours offset from UTC, with a leading zero for a single-digit value. More information: The "zz" custom format specifier.	2009-06-15T13:45:30-07:00 -> -07
"zzz"	Hours and minutes offset from UTC. More information: The "zzz" custom format specifier.	2009-06-15T13:45:30-07:00 -> -07:00
"::"	The time separator. More information: The "::" custom format specifier.	2009-06-15T13:45:30 -> : (en-US) 2009-06-15T13:45:30 -> . (it-IT) 2009-06-15T13:45:30 -> : (ja-JP)

Format specifier	Description	Examples
"/"	The date separator. More Information: The "/" custom format specifier .	2009-06-15T13:45:30 -> / (en-US) 2009-06-15T13:45:30 -> - (ar-DZ) 2009-06-15T13:45:30 -> . (tr-TR)
"string" 'string'	Literal string delimiter. More information: Character literals .	2009-06-15T13:45:30 ("arr:" h:m t) -> arr: 1:45 P 2009-06-15T13:45:30 ('arr:' h:m t) -> arr: 1:45 P
%	Defines the following character as a custom format specifier. More information: Using single custom format specifiers .	2009-06-15T13:45:30 (%h) -> 1
\ " ' , '	The escape sequences. More information: Character literals and Using the escape sequences .	2009-06-15T13:45:30 (h \h) -> 1 h 2009-06-15T13:45:30 (h ""h""") -> 1 h 2009-06-15T13:45:30 (h 'h') -> 1 h
Any other character	The character is copied to the result string unchanged. More information: Character literals .	2009-06-15T01:45:30 (arr hh:mm t) -> arr 01:45 A

The following sections provide additional information about each custom date and time format specifier. Unless otherwise noted, each specifier produces an identical string representation regardless of whether it's used with a **Date**, **DateTime**, **DateTimeZone**, or **Time** value.

Day "d" format specifier

The "d" custom format specifier

The "d" custom format specifier represents the day of the month as a number from 1 to 31. A single-digit day is formatted without a leading zero.

If the "d" format specifier is used without other custom format specifiers, it's interpreted as the "d" standard date and time format specifier. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "d" custom format specifier in several format strings.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15), [Format = "d,
M", Culture = ""]),
        // Displays 29, 8

        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15), [Format = "d,
MMMM", Culture = "en-US"]),
        // Displays 29, August

        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15), [Format = "d,
MMMM", Culture = "es-MX"])
        // Displays 29, agosto
    }
in
    Source
```

[Back to table](#)

The "dd" custom format specifier

The "dd" custom format string represents the day of the month as a number from 01 to 31. A single-digit day is formatted with a leading zero.

The following example includes the "dd" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 1, 2, 6, 30, 15), [Format = "dd,
MM", Culture = ""])
        // Displays 02, 01
    }
in
    Source
```

[Back to table](#)

The "ddd" custom format specifier

The "ddd" custom format specifier represents the abbreviated name of the day of the week. The localized abbreviated name of the day of the week is retrieved from the current or specified culture.

The following example includes the "ddd" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15), [Format = "ddd d
MMM", Culture = "en-US"]),
        // Displays Thu 29 Aug

        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15), [Format = "ddd d
MMM", Culture = "fr-FR"])
        // Displays jeu. 29 août
    }
in
    Source
```

[Back to table](#)

The "dddd" custom format specifier

The "dddd" custom format specifier (plus any number of additional "d" specifiers) represents the full name of the day of the week. The localized name of the day of the week is retrieved from the current or specified culture.

The following example includes the "dddd" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15), [Format = "dddd
dd MMMM", Culture = "en-US"]),
        // Displays Thursday 29 August
    }
```

```

        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15), [Format = "dddd
dd MMMM", Culture = "it-IT"])
        // Displays giovedì 29 agosto
    }
in
    Source

```

[Back to table](#)

Lowercase seconds "f" fraction specifier

The "f" custom format specifier

The "f" custom format specifier represents the most significant digit of the seconds fraction; that is, it represents the tenths of a second in a date and time value.

If the "f" format specifier is used without other format specifiers, it's interpreted as the "f" standard date and time format specifier. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

When you use "f" format specifiers as part of a format string supplied to parse the number of fractional seconds, the number of "f" format specifiers indicates the number of most significant digits of the seconds fraction that must be present to successfully parse the string.

The following example includes the "f" custom format specifier in a custom format string.

Power Query M

```

let
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =
"hh:mm:ss:f", Culture = ""]),
        // Displays 07:27:15:0

        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =
"hh:mm:ss:F", Culture = ""])
        // Displays 07:27:15:
    }
in
    Source

```

[Back to table](#)

The "ff" custom format specifier

The "ff" custom format specifier represents the two most significant digits of the seconds fraction; that is, it represents the hundredths of a second in a date and time value.

following example includes the "ff" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =
        "hh:mm:ss:ff", Culture = ""]),
        // Displays 07:27:15:01

        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =
        "hh:mm:ss:FF", Culture = ""])
        // Displays 07:27:15:01
    }
in
    Source
```

[Back to table](#)

The "fff" custom format specifier

The "fff" custom format specifier represents the three most significant digits of the seconds fraction; that is, it represents the milliseconds in a date and time value.

The following example includes the "fff" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =
        "hh:mm:ss:fff", Culture = ""]),
        // Displays 07:27:15:018

        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =
        "hh:mm:ss:FFF", Culture = ""])
        // Displays 07:27:15:018
    }
}
```

[Back to table](#)

The "ffff" custom format specifier

The "ffff" custom format specifier represents the four most significant digits of the seconds fraction; that is, it represents the ten thousandths of a second in a date and time value.

Although it's possible to display the ten thousandths of a second component of a time value, that value might not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows Server 2019 and Windows 11 operating systems, the clock's resolution is approximately 1 millisecond.

[Back to table](#)

The "fffff" custom format specifier

The "fffff" custom format specifier represents the five most significant digits of the seconds fraction; that is, it represents the hundred thousandths of a second in a date and time value.

Although it's possible to display the hundred thousandths of a second component of a time value, that value might not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows Server 2019 and Windows 11 operating systems, the clock's resolution is approximately 1 millisecond.

[Back to table](#)

The "ffffff" custom format specifier

The "ffffff" custom format specifier represents the six most significant digits of the seconds fraction; that is, it represents the millionths of a second in a date and time value.

Although it's possible to display the millionths of a second component of a time value, that value might not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows Server 2019 and Windows 11 operating systems, the clock's resolution is approximately 1 millisecond.

[Back to table](#)

The "ffffff" custom format specifier

The "ffffff" custom format specifier represents the seven most significant digits of the seconds fraction; that is, it represents the ten millionths of a second in a date and time value.

Although it's possible to display the ten millionths of a second component of a time value, that value might not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows Server 2019 and Windows 11 operating systems, the clock's resolution is approximately 1 millisecond.

[Back to table](#)

Uppercase seconds "F" fraction specifier

The "F" custom format specifier

The "F" custom format specifier represents the most significant digit of the seconds fraction; that is, it represents the tenths of a second in a date and time value. Nothing is displayed if the digit is zero, and the decimal point that follows the number of seconds is also not displayed.

If the "F" format specifier is used without other format specifiers, it's interpreted as the "F" standard date and time format specifier. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The number of "F" format specifiers used when parsing indicates the maximum number of most significant digits of the seconds fraction that can be present to successfully parse the string.

The following example includes the "F" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =
        "hh:mm:ss:f", Culture = ""]),
        // Displays 07:27:15:0

        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =
        "hh:mm:ss:F", Culture = ""])
        // Displays 07:27:15:
```

```
}  
in  
    Source
```

[Back to table](#)

The "FF" custom format specifier

The "FF" custom format specifier represents the two most significant digits of the seconds fraction; that is, it represents the hundredths of a second in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the two significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

The following example includes the "FF" custom format specifier in a custom format string.

Power Query M

```
let  
    Source =  
    {  
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =  
            "hh:mm:ss:ff", Culture = ""]),  
        // Displays 07:27:15:01  
  
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =  
            "hh:mm:ss:FF", Culture = ""]),  
        // Displays 07:27:15:01  
    }  
in  
    Source
```

[Back to table](#)

The "FFF" custom format specifier

The "FFF" custom format specifier represents the three most significant digits of the seconds fraction; that is, it represents the milliseconds in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the three significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

The following example includes the "FFF" custom format specifier in a custom format string.

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =
"hh:mm:ss:fff", Culture = ""]),
        // Displays 07:27:15:018

        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15.018), [Format =
"hh:mm:ss:FFF", Culture = ""])
        // Displays 07:27:15:018
    }
in
    Source
```

[Back to table](#)

The "FFFF" custom format specifier

The "FFFF" custom format specifier represents the four most significant digits of the seconds fraction; that is, it represents the ten thousandths of a second in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the four significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

Although it's possible to display the ten thousandths of a second component of a time value, that value might not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows Server 2019 and Windows 11 operating systems, the clock's resolution is approximately 1 millisecond.

[Back to table](#)

The "FFFFF" custom format specifier

The "FFFFF" custom format specifier represents the five most significant digits of the seconds fraction; that is, it represents the hundred thousandths of a second in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the five significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

Although it's possible to display the hundred thousandths of a second component of a time value, that value might not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows Server 2019 and Windows 11 operating systems, the clock's resolution is approximately 1 millisecond.

[Back to table](#)

The "FFFFFF" custom format specifier

The "FFFFFF" custom format specifier represents the six most significant digits of the seconds fraction; that is, it represents the millionths of a second in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the six significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

Although it's possible to display the millionths of a second component of a time value, that value might not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows Server 2019 and Windows 11 operating systems, the clock's resolution is approximately 1 millisecond.

[Back to table](#)

The "FFFFFFFF" custom format specifier

The "FFFFFFFF" custom format specifier represents the seven most significant digits of the seconds fraction; that is, it represents the ten millionths of a second in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the seven significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

Although it's possible to display the ten millionths of a second component of a time value, that value might not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows Server 2019 and Windows 11 operating systems, the clock's resolution is approximately 1 millisecond.

[Back to table](#)

Era "g" format specifier

The "g" or "gg" custom format specifier

The "g" or "gg" custom format specifiers (plus any number of additional "g" specifiers) represents the period or era, such as A.D. The formatting operation ignores this specifier if the date to be formatted doesn't have an associated period or era string.

If the "g" format specifier is used without other custom format specifiers, it's interpreted as the "g" standard date and time format specifier. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "g" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        Date.ToText(#date(70, 08, 04), [Format = "MM/dd/yyyy g", Culture =
        ""]),
        // Displays 08/04/0070 A.D.

        Date.ToText(#date(70, 08, 04), [Format = "MM/dd/yyyy g", Culture =
        "fr-FR"])
        // Displays 08/04/0070 ap. J.-C.
    }
in
    Source
```

[Back to table](#)

Lowercase hour "h" format specifier

The "h" custom format specifier

The "h" custom format specifier represents the hour as a number from 1 to 12; that is, the hour is represented by a 12-hour clock that counts the whole hours since midnight or noon. A particular hour after midnight is indistinguishable from the same hour after noon. The hour isn't rounded, and a single-digit hour is formatted without a leading zero. For example, given a time of 5:43 in the morning or afternoon, this custom format specifier displays "5".

If the "h" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws an expression error. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "h" custom format specifier in a custom format string.

Power Query M

```

let
    Source =
    {
        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "h:m:s.F t", Culture = ""]),
        // Displays 6:9:1 P

        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "h:m:s.F t", Culture = "el-GR"]),
        // Displays 6:9:1 μ

        DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format = "h:m:s.F t", Culture = ""]),
        // Displays 9:18:1.5 A

        DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format = "h:m:s.F t", Culture = "el-GR"]),
        // Displays 9:18:1.5 π
    }
in
    Source

```

[Back to table](#)

The "hh" custom format specifier

The "hh" custom format specifier (plus any number of additional "h" specifiers) represents the hour as a number from 01 to 12; that is, the hour is represented by a 12-hour clock that counts the whole hours since midnight or noon. A particular hour after midnight is indistinguishable from the same hour after noon. The hour isn't rounded, and a single-digit hour is formatted with a leading zero. For example, given a time of 5:43 in the morning or afternoon, this format specifier displays "05".

The following example includes the "hh" custom format specifier in a custom format string.

Power Query M

```

let
    Source =
    {
        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "hh:mm:ss tt", Culture = ""]),
        // Displays 06:09:01 PM

        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "hh:mm:ss tt", Culture = "hu-HU"]),
        // Displays 06:09:01 du.
    }

```



```

        DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format =
"hh:mm:ss.ff tt", Culture = ""]),
        // Displays 09:18:01.50 AM

        DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format =
"hh:mm:ss.ff tt", Culture = "hu-HU"])
        // Displays 09:18:01.50 de.
    }
in
    Source

```

[Back to table](#)

Uppercase hour "H" format specifier

The "H" custom format specifier

The "H" custom format specifier represents the hour as a number from 0 to 23; that is, the hour is represented by a zero-based 24-hour clock that counts the hours since midnight. A single-digit hour is formatted without a leading zero.

If the "H" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws an expression error. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "H" custom format specifier in a custom format string.

Power Query M

```

let
    Source =
    {
        DateTime.ToText(#datetime(2024, 1, 1, 6, 9, 1), [Format = "H:mm:ss",
Culture = ""])
        // Displays 6:09:01
    }
in
    Source

```

[Back to table](#)

The "HH" custom format specifier

The "HH" custom format specifier (plus any number of additional "H" specifiers) represents the hour as a number from 00 to 23; that is, the hour is represented by a zero-based 24-hour clock that counts the hours since midnight. A single-digit hour is formatted with a leading zero.

The following example includes the "HH" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 1, 1, 6, 9, 1), [Format =
"HH:mm:ss", Culture = ""])
        // Displays 06:09:01
    }
in
    Source
```

[Back to table](#)

Time zone "K" format specifier

The "K" custom format specifier

The "K" custom format specifier represents the time zone information of a date and time value. When this format specifier is used with **DateTimeZone** values, the result string is defined as:

- For the local time zone, this specifier produces a result string containing the local offset from Coordinated Universal Time (UTC), for example, "-07:00", if your query runs in Power Query Desktop. If your query runs in Power Query Online, the result string produces no offset from UTC time, that is, "+00:00".
- For a UTC time, the result string produces no offset from UTC time; that is, "+00:00" to represent a UTC date.
- For a time from an unspecified time zone, the result is empty.

If the "K" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws an expression error. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The following example displays the string that results from using the "K" custom format specifier with various values on a system in the U.S. Pacific Time zone.

Power Query M

```
let
    Source =
    {
        DateTimeZone.ToText(DateTimeZone.LocalNow(), [Format="%K"]),
        // Displays -07:00 (Desktop) or +00:00 (Online)

        DateTimeZone.ToText(DateTimeZone.UtcNow(), [Format="%K"]),
        // Displays +00:00

        Text.Format("'#{0}'", {DateTime.ToText(DateTime.LocalNow(),
[Format="%K"])}))
        // Displays ''
    }
in
    Source
```

ⓘ Note

The value returned by [DateTimeZone.LocalNow](#) depends on whether you're running Power Query on a local machine or online. For example, in the sample above on a system in the U.S. Pacific Time zone, Power Query Desktop returns `-07:00` because it's reading the time set on your local machine. However, Power Query Online returns `+00:00` because it's reading the time set on the cloud virtual machines, which are set to UTC.

[Back to table](#)

Minute "m" format specifier

The "m" custom format specifier

The "m" custom format specifier represents the minute as a number from 0 to 59. The minute represents whole minutes that have passed since the last hour. A single-digit minute is formatted without a leading zero.

If the "m" format specifier is used without other custom format specifiers, it's interpreted as the "m" standard date and time format specifier. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "m" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "h:m:s.F t", Culture = ""]),
        // Displays 6:9:1 P

        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "h:m:s.F t", Culture = "el-GR"]),
        // Displays 6:9:1 μ

        DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format = "h:m:s.F t", Culture = ""]),
        // Displays 9:18:1.5 A

        DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format = "h:m:s.F t", Culture = "el-GR"]),
        // Displays 9:18:1.5 π
    }
in
    Source
```

[Back to table](#)

The "mm" custom format specifier

The "mm" custom format specifier (plus any number of additional "m" specifiers) represents the minute as a number from 00 to 59. The minute represents whole minutes that have passed since the last hour. A single-digit minute is formatted with a leading zero.

The following example includes the "mm" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "hh:mm:ss tt", Culture = ""]),
        // Displays 06:09:01 PM

        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "hh:mm:ss tt", Culture = "el-GR"]),
        // Displays 06:09:01 PM
    }
in
    Source
```

```

tt", Culture = "hu-HU"]]),
    // Displays 06:09:01 du.

    DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format =
"hh:mm:ss.ff tt", Culture = ""]),
    // Displays 09:18:01.50 AM

    DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format =
"hh:mm:ss.ff tt", Culture = "hu-HU"])
    // Displays 09:18:01.50 de.
}
in
    Source

```

[Back to table](#)

Month "M" format specifier

The "M" custom format specifier

The "M" custom format specifier represents the month as a number from 1 to 12 (or from 1 to 13 for calendars that have 13 months). A single-digit month is formatted without a leading zero.

If the "M" format specifier is used without other custom format specifiers, it's interpreted as the "M" standard date and time format specifier. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "M" custom format specifier in a custom format string.

Power Query M

```

let
    Source =
    {
        Date.ToText(#date(2024, 8, 18), [Format = "(M) MMM, MMMM", Culture =
"en-US"]),
        // Displays (8) Aug, August

        Date.ToText(#date(2024, 8, 18), [Format = "(M) MMM, MMMM", Culture =
"nl-NL"]),
        // Displays (8) aug, augustus

        Date.ToText(#date(2024, 8, 18), [Format = "(M) MMM, MMMM", Culture =
"lv-LV"])
        // Displays (8) aug., augusts
    }

```

```
}  
in  
    Source
```

[Back to table](#)

The "MM" custom format specifier

The "MM" custom format specifier represents the month as a number from 01 to 12 (or from 1 to 13 for calendars that have 13 months). A single-digit month is formatted with a leading zero.

The following example includes the "MM" custom format specifier in a custom format string.

Power Query M

```
let  
    Source =  
    {  
        DateTime.ToText(#datetime(2024, 1, 2, 6, 30, 15), [Format = "dd,  
MM", Culture = ""])  
        // Displays 02, 01  
    }  
in  
    Source
```

[Back to table](#)

The "MMM" custom format specifier

The "MMM" custom format specifier represents the abbreviated name of the month. The localized abbreviated name of the month is retrieved from the abbreviated month names of the current or specified culture. If there is a "d" or "dd" custom format specifier in the custom format string, the month name is retrieved from the abbreviated genitive names instead.

The following example includes the "MMM" custom format specifier in a custom format string.

Power Query M

```
let  
    Source =  
    {  
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15), [Format = "ddd d
```

```

MMM", Culture = "en-US"])),
    // Displays Thu 29 Aug

    DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15), [Format = "ddd d
MMM", Culture = "fr-FR"]])
    // Displays jeu. 29 août
}
in
Source

```

[Back to table](#)

The "MMMM" custom format specifier

The "MMMM" custom format specifier represents the full name of the month. The localized name of the month is retrieved from the current or specified culture. If there is a "d" or "dd" custom format specifier in the custom format string, the month name is retrieved from the abbreviated genitive names instead.

The following example includes the "MMMM" custom format specifier in a custom format string.

Power Query M

```

let
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15), [Format = "dddd
dd MMMM", Culture = "en-US"])),
        // Displays Thursday 29 August

        DateTime.ToText(#datetime(2024, 8, 29, 19, 27, 15), [Format = "dddd
dd MMMM", Culture = "it-IT"]])
        // Displays giovedì 29 agosto
    }
in
    Source

```

[Back to table](#)

Seconds "s" format specifier

The "s" custom format specifier

The "s" custom format specifier represents the seconds as a number from 0 to 59. The result represents whole seconds that have passed since the last minute. A single-digit

second is formatted without a leading zero.

If the "s" format specifier is used without other custom format specifiers, it's interpreted as the "s" standard date and time format specifier. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "s" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "h:m:s.F t", Culture = ""]),
        // Displays 6:9:1 P

        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "h:m:s.F t", Culture = "el-GR"]),
        // Displays 6:9:1 μ

        DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format = "h:m:s.F t", Culture = ""]),
        // Displays 9:18:1.5 A

        DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format = "h:m:s.F t", Culture = "el-GR"]),
        // Displays 9:18:1.5 π
    }
in
    Source
```

[Back to table](#)

The "ss" custom format specifier

The "ss" custom format specifier (plus any number of additional "s" specifiers) represents the seconds as a number from 00 to 59. The result represents whole seconds that have passed since the last minute. A single-digit second is formatted with a leading zero.

The following example includes the "ss" custom format specifier in a custom format string.

Power Query M


```

let
    Source =
    {
        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "hh:mm:ss
tt", Culture = ""]),
        // Displays 06:09:01 PM

        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "hh:mm:ss
tt", Culture = "hu-HU"]),
        // Displays 06:09:01 du.

        DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format =
"hh:mm:ss.ff tt", Culture = ""]),
        // Displays 09:18:01.50 AM

        DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format =
"hh:mm:ss.ff tt", Culture = "hu-HU"])
        // Displays 09:18:01.50 de.
    }
in
    Source

```

[Back to table](#)

Meridiem "t" format specifier

The "t" custom format specifier

The "t" custom format specifier represents the first character of the AM/PM designator. The appropriate localized designator is retrieved from the current or specific culture. The AM designator is used for all times from 0:00:00 (midnight) to 11:59:59.999. The PM designator is used for all times from 12:00:00 (noon) to 23:59:59.999.

If the "t" format specifier is used without other custom format specifiers, it's interpreted as the "t" standard date and time format specifier. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "t" custom format specifier in a custom format string.

Power Query M

```

let
    Source =
    {
        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "h:m:s.F
t", Culture = ""]),

```

```

// Displays 6:9:1 P

DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "h:m:s.F
t", Culture = "el-GR"]),
// Displays 6:9:1 μ

DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format =
"h:m:s.F t", Culture = ""]),
// Displays 9:18:1.5 A

DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format =
"h:m:s.F t", Culture = "el-GR"])
// Displays 9:18:1.5 π
}
in
Source

```

[Back to table](#)

The "tt" custom format specifier

The "tt" custom format specifier (plus any number of additional "t" specifiers) represents the entire AM/PM designator. The appropriate localized designator is retrieved from the current or specific culture. The AM designator is used for all times from 0:00:00 (midnight) to 11:59:59.999. The PM designator is used for all times from 12:00:00 (noon) to 23:59:59.999.

Make sure to use the "tt" specifier for languages for which it's necessary to maintain the distinction between AM and PM. An example is Japanese, for which the AM and PM designators differ in the second character instead of the first character.

The following example includes the "tt" custom format specifier in a custom format string.

Power Query M

```

let
    Source =
    {
        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "hh:mm:ss
tt", Culture = ""]),
        // Displays 06:09:01 PM

        DateTime.ToText(#datetime(2024, 1, 1, 18, 9, 1), [Format = "hh:mm:ss
tt", Culture = "hu-HU"]),
        // Displays 06:09:01 du.

        DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format =
"hh:mm:ss.ff tt", Culture = ""]),
    }
in
Source

```

```
// Displays 09:18:01.50 AM

DateTime.ToText(#datetime(2024, 1, 1, 9, 18, 1.500), [Format =
"hh:mm:ss.ff tt", Culture = "hu-HU"])
// Displays 09:18:01.50 de.
}
in
Source
```

[Back to table](#)

Year "y" format specifier

The "y" custom format specifier

The "y" custom format specifier represents the year as a one-digit or two-digit number. If the year has more than two digits, only the two low-order digits appear in the result. If the first digit of a two-digit year begins with a zero (for example, 2008), the number is formatted without a leading zero.

If the "y" format specifier is used without other custom format specifiers, it's interpreted as the "y" standard date and time format specifier. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "y" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        Date.ToText(#date(1, 12, 1), [Format = "%y"]),
        // Displays 1

        Date.ToText(#date(2024, 1, 1), [Format = "%y"])
        // Displays 24
    }
in
    Source
```

[Back to table](#)

The "yy" custom format specifier

The "yy" custom format specifier represents the year as a two-digit number. If the year has more than two digits, only the two low-order digits appear in the result. If the two-digit year has fewer than two significant digits, the number is padded with leading zeros to produce two digits.

In a parsing operation, a two-digit year that is parsed using the "yy" custom format specifier is interpreted based on the format provider's current calendar. The following example parses the string representation of a date that has a two-digit year by using the default Gregorian calendar of the en-US culture, which, in this case, is the current culture. The values returned for the four-digit date depend on the 100 year range set by the operating system.

Power Query M

```
let
    // Define the date format and value
    fmt = "dd-MMM-yy",

    // Convert year 49 to a 4-digit year
    firstDate = Text.Format("#{0}", { Date.FromText("24-Jan-49", [Format =
    fmt]) }),

    // Convert year 50 to a 4-digit year
    finalDate = Text.Format("#{0}", { Date.FromText("24-Jan-50", [Format =
    fmt]) }),
    Heading = "Default Two Digit Year Range: 1950 - 2049",
    result = {Heading, firstDate, finalDate}
in
    result

// The example displays the following output:
//      Default Two Digit Year Range: 1950 - 2049
//      1/24/2049
//      1/24/1950
```

The following example includes the "yy" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        Date.ToText(#date(1, 12, 1), [Format = "yy"]),
        // Displays 01

        Date.ToText(#date(2024, 1, 1), [Format = "yy"])
        // Displays 24
    }

```

in
Source

[Back to table](#)

The "yyy" custom format specifier

The "yyy" custom format specifier represents the year with a minimum of three digits. If the year has more than three significant digits, they are included in the result string. If the year has fewer than three digits, the number is padded with leading zeros to produce three digits.

ⓘ Note

For the Thai Buddhist calendar, which can have five-digit years, this format specifier displays all significant digits.

The following example includes the "yyy" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        Date.ToText(#date(1, 12, 1), [Format = "yyy"]),
        // Displays 001

        Date.ToText(#date(2024, 1, 1), [Format = "yyy"])
        // Displays 2024
    }
in
    Source
```

[Back to table](#)

The "yyyy" custom format specifier

The "yyyy" custom format specifier represents the year with a minimum of four digits. If the year has more than four significant digits, they are included in the result string. If the year has fewer than four digits, the number is padded with leading zeros to produce four digits.

ⓘ Note

For the Thai Buddhist calendar, which can have five-digit years, this format specifier displays a minimum of four digits.

The following example includes the "yyyy" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        Date.ToText(#date(1, 12, 1), [Format = "yyyy"]),
        // Displays 0001

        Date.ToText(#date(2024, 1, 1), [Format = "yyyy"])
        // Displays 2024
    }
in
    Source
```

[Back to table](#)

The "yyyyy" custom format specifier

The "yyyyy" custom format specifier (plus any number of additional "y" specifiers) represents the year with a minimum of five digits. If the year has more than five significant digits, they are included in the result string. If the year has fewer than five digits, the number is padded with leading zeros to produce five digits.

If there are additional "y" specifiers, the number is padded with as many leading zeros as necessary to produce the number of "y" specifiers.

The following example includes the "yyyyy" custom format specifier in a custom format string.

Power Query M

```
let
    Source =
    {
        Date.ToText(#date(1, 12, 1), [Format = "yyyyy"]),
        // Displays 00001

        Date.ToText(#date(2024, 1, 1), [Format = "yyyyy"])
    }
```

```
// Displays 02024
}
in
Source
```

[Back to table](#)

Offset "z" format specifier

The "z" custom format specifier

With **DateTimeZone** values, the "z" custom format specifier represents the signed offset of the specified time zone from Coordinated Universal Time (UTC), measured in hours. The offset is always displayed with a leading sign. A plus sign (+) indicates hours ahead of UTC, and a minus sign (-) indicates hours behind UTC. A single-digit offset is formatted *without* a leading zero.

The following table shows how the offset value changes depending on the **DateTimeZone** function.

[Expand table](#)

DateTimeZone value	Offset value
DateTimeZone.LocalNow	On Power Query Desktop, the signed offset of the local operating system's time zone from UTC. On Power Query Online, returns <code>+00</code> .
DateTimeZone.UtcNow	Returns <code>+0</code> .

If the "z" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws an expression error. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "z" custom format specifier in a custom format string on a system in the U.S. Pacific Time zone.

Power Query M

```
let
    Source =
    {
        DateTimeZone.ToText(DateTimeZone.LocalNow(), [Format="{0:%z}"]),
        // Displays {0:-7} on Power Query Desktop
        // Displays {0:+0} on Power Query Online
    }
```

```

DateTimeZone.ToText(DateTimeZone.UtcNow(), [Format="{0:%z}"]),
// Displays {0:+0}

DateTimeZone.ToText(DateTimeZone.SwitchZone(
    #datetimezone(2024, 8, 1, 0, 0, 0, 0, 0), 6),
    [Format = "{0:%z}"])
)
// Displays {0:+6}
}
in
Source

```

ⓘ Note

The value returned by [DateTimeZone.LocalNow](#) depends on whether you're running Power Query on a local machine or online. For example, in the sample above on a system in the U.S. Pacific Time zone, Power Query Desktop returns `{0:-7}` because it's reading the time set on your local machine. However, Power Query Online returns `{0:+0}` because it's reading the time set on the cloud virtual machines, which are set to UTC.

[Back to table](#)

The "zz" custom format specifier

With **DateTimeZone** values, the "zz" custom format specifier represents the signed offset of the specified time zone from UTC, measured in hours. The offset is always displayed with a leading sign. A plus sign (+) indicates hours ahead of UTC, and a minus sign (-) indicates hours behind UTC. A single-digit offset is formatted *with* a leading zero.

The following table shows how the offset value changes depending on the **DateTimeZone** function.

[Expand table](#)

DateTimeZone value	Offset value
DateTimeZone.LocalNow	On Power Query Desktop, the signed offset of the local operating system's time zone from UTC. On Power Query Online, returns <code>+00</code> .
DateTimeZone.UtcNow	Returns <code>+00</code> .

The following example includes the "zz" custom format specifier in a custom format string on a system in the U.S. Pacific Time zone.

Power Query M

```
let
    Source =
    {
        DateTimeZone.ToText(DateTimeZone.LocalNow(), [Format="{0:zz}"]),
        // Displays {-07} on Power Query Desktop
        // Displays {0:+00} on Power Query Online

        DateTimeZone.ToText(DateTimeZone.UtcNow(), [Format="{0:zz}"]),
        // Displays {0:+00}

        DateTimeZone.ToText(DateTimeZone.SwitchZone(
            #datetimezone(2024, 8, 1, 0, 0, 0, 0, 6),
            [Format = "{0:zz}"]
        ),
        // Displays {0:+06}
    }
in
    Source
```

ⓘ Note

The value returned by [DateTimeZone.LocalNow](#) depends on whether you're running Power Query on a local machine or online. For example, in the sample above on a system in the U.S. Pacific Time zone, Power Query Desktop returns `{0:-07}` because it's reading the time set on your local machine. However, Power Query Online returns `{0:+00}` because it's reading the time set on the cloud virtual machines, which are set to UTC.

[Back to table](#)

The "zzz" custom format specifier

With **DateTimeZone** values, the "zzz" custom format specifier represents the signed offset of the specified time zone from UTC, measured in hours and minutes. The offset is always displayed with a leading sign. A plus sign (+) indicates hours ahead of UTC, and a minus sign (-) indicates hours behind UTC. A single-digit offset is formatted with a leading zero.

The following table shows how the offset value changes depending on the **DateTimeZone** function.

DateTimeZoneValue value	Offset value
DateTimeZone.LocalNow	On Power Query Desktop, the signed offset of the local operating system's time zone from UTC. On Power Query Online, returns <code>+00</code> .
DateTimeZone.UtcNow	Returns <code>+00:00</code> .

The following example includes the "zzz" custom format specifier in a custom format string on a system in the U.S. Pacific Time zone.

Power Query M

```
let
    Source =
    {
        DateTimeZone.ToText(DateTimeZone.LocalNow(), [Format="{0:zzz}"]),
        // Displays {0:-07:00} on Power Query Desktop
        // Displays {0:+00:00} on Power Query Online

        DateTimeZone.ToText(DateTimeZone.UtcNow(), [Format="{0:zzz}"]),
        // Displays {0:+00:00}

        DateTimeZone.ToText(DateTimeZone.SwitchZone(
            #datetimezone(2024, 8, 1, 0, 0, 0, 0, 0), 6),
            [Format = "{0:zzz}"])
        )
        // Displays {0:+06:00}
    }
in
    Source
```

ⓘ Note

The value returned by [DateTimeZone.LocalNow](#) depends on whether you're running Power Query on a local machine or online. For example, in the sample above on a system in the U.S. Pacific Time zone, Power Query Desktop returns `{0:-07:00}` because it's reading the time set on your local machine. However, Power Query Online returns `{0:+00:00}` because it's reading the time set on the cloud virtual machines, which are set to UTC.

[Back to table](#)

Date and time separator specifiers

The ":" custom format specifier

The ":" custom format specifier represents the time separator, which is used to differentiate hours, minutes, and seconds. The appropriate localized time separator is retrieved from the current or specified culture.

ⓘ Note

To change the time separator for a particular date and time string, specify the separator character within a literal string delimiter. For example, the custom format string `hh_dd_ss` produces a result string in which "_" (an underscore) is always used as the time separator.

If the ":" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws an expression error. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

[Back to table](#)

The "/" custom format specifier

The "/" custom format specifier represents the date separator, which is used to differentiate years, months, and days. The appropriate localized date separator is retrieved from the current or specified culture.

ⓘ Note

To change the date separator for a particular date and time string, specify the separator character within a literal string delimiter. For example, the custom format string `mm/dd/yyyy` produces a result string in which "/" is always used as the date separator.

If the "/" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws an expression error. For more information about using a single format specifier, go to [Using Single Custom Format Specifiers](#) later in this article.

Character literals

The following characters in a custom date and time format string are reserved and are always interpreted as formatting characters or, in the case of `"`, `'`, `/`, and `\`, as special characters.

- `F`
- `H`
- `K`
- `M`
- `d`
- `f`
- `g`
- `h`
- `m`
- `s`
- `t`
- `y`
- `z`
- `%`
- `:`
- `/`
- `"`
- `'`
- `\`

All other characters are always interpreted as character literals and, in a formatting operation, are included in the result string unchanged. In a parsing operation, they must match the characters in the input string exactly; the comparison is case-sensitive.

The following example includes the literal characters "PST" (for Pacific Standard Time) and "PDT" (for Pacific Daylight Time) to represent the local time zone in a format string. Note that the string is included in the result string, and that a string that includes the local time zone string also parses successfully.

Power Query M

```
let
    #"Date Formats" = {"dd MMM yyyy hh:mm tt PST", "dd MMM yyyy hh:mm tt
```

```

PDT"}},
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 18, 16, 50, 0), [Format = #"Date
Formats"{1}]),
        try DateTime.ToText(DateTime.FromText(
            "25 Dec 2023 12:00 pm PST", [Format = #"Date Formats"{0}]))
            otherwise "Unable to parse '" & "25 Dec 2023 12:00 pm PST" & "'"
    }
in
    Source

// The example displays the following output text:
//      18 Aug 2024 04:50 PM PDT
//      12/25/2023 12:00:00 PM

```

There are two ways to indicate that characters are to be interpreted as literal characters and not as reserve characters, so that they can be included in a result string or successfully parsed in an input string:

- By escaping each reserved character. For more information, go to [Using the escape sequences](#).

The following example includes the literal characters "pst" (for Pacific Standard time) to represent the local time zone in a format string. Because both "s" and "t" are custom format strings, both characters must be escaped to be interpreted as character literals.

Power Query M

```

let
    #"Date Format" = "dd MMM yyyy hh:mm tt p's''t'",
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 18, 16, 50, 0), [Format =
#"Date Format"]),
        try DateTime.ToText(DateTime.FromText(
            "25 Dec 2023 12:00 pm pst", [Format = #"Date Format"]))
            otherwise "Unable to parse '" & "25 Dec 2023 12:00 pm
pst" & "'"
    }
in
    Source

// The example displays the following output text:
//      18 Aug 2024 04:50 PM pst
//      12/25/2016 12:00:00 PM

```

- By enclosing the entire literal string in apostrophes. The following example is like the previous one, except that "pst" is enclosed in apostrophes to indicate that the entire delimited string should be interpreted as character literals.

Power Query M

```
let
    #"Date Format" = "dd MMM yyyy hh:mm tt 'pst'",
    Source =
    {
        DateTime.ToText(#datetime(2024, 8, 18, 16, 50, 0), [Format =
        #"Date Format"]),
        try DateTime.ToText(DateTime.FromText(
            "25 Dec 2023 12:00 pm pst", [Format = #"Date Format"]))
            otherwise "Unable to parse '" & "25 Dec 2023 12:00 pm
        pst" & "'"
    }
in
    Source

// The example displays the following output text:
//      18 Aug 2024 04:50 PM pst
//      12/25/2016 12:00:00 PM
```

Notes

Using single custom format specifiers

A custom date and time format string consists of two or more characters. Date and time formatting methods interpret any single-character string as a standard date and time format string. If they don't recognize the character as a valid format specifier, they throw an expression error. For example, a format string that consists only of the specifier "h" is interpreted as a standard date and time format string. However, in this particular case, an exception is thrown because there is no "h" standard date and time format specifier.

To use any of the custom date and time format specifiers as the only specifier in a format string (that is, to use the "d", "f", "F", "g", "h", "H", "K", "m", "M", "s", "t", "y", "z", ":", or "/" custom format specifier by itself), include a space before or after the specifier, or include a percent ("%") format specifier before the single custom date and time specifier.

For example, "%h" is interpreted as a custom date and time format string that displays the hour represented by the current date and time value. You can also use the " h" or "h " format string, although this includes a space in the result string along with the hour. The following example illustrates these three format strings.

```

let
    date = #datetime(2024, 6, 15, 13, 45, 0),
    Source =
    {
        Text.Format("'#{0}'", {DateTime.ToText(date, [Format = "%h"])}),
        Text.Format("'#{0}'", {DateTime.ToText(date, [Format = " h"])}),
        Text.Format("'#{0}'", {DateTime.ToText(date, [Format = "h "])})
    }
in
    Source

// The example displays a list with the following output text,
// with <sp> representing a space:
//      '1'
//      ' 1'
//      '1 '

```

Using the escape sequences

The "d", "f", "F", "g", "h", "H", "K", "m", "M", "s", "t", "y", "z", ":", or "/" characters in a format string are interpreted as custom format specifiers rather than as literal characters.

To prevent a character from being interpreted as a format specifier, you can:

- Precede it with a backslash.
- Surround it with a single quote.
- Surround it with two double quotes.

Each of these characters acts as an escape sequence. The escape sequence signifies that the following character or surrounded character is a text literal that should be included in the result string unchanged.

To include a double quote in a result string, you must escape it with another double quote ("").

The following example uses different escape sequences to prevent the formatting operation from interpreting the "h" and "m" characters as format specifiers.

```

let
    date = #datetime(2024, 6, 15, 13, 45, 30.90),
    format1 = "h \h m \m",
    format2 = "h ""h"" m ""m""",
    format3 = "h 'h' m 'm'",
    Source =

```

```
{
    Text.Format("#{0} ({1}) -> #{2}", {DateTime.ToText(date), format1,
DateTime.ToText(date, format1)}),
    Text.Format("#{0} ({1}) -> #{2}", {DateTime.ToText(date), format2,
DateTime.ToText(date, format2)}),
    Text.Format("#{0} ({1}) -> #{2}", {DateTime.ToText(date), format3,
DateTime.ToText(date, format3)})
}
in
    Source

// The example displays the following output text:
//      6/15/2024 1:45:30 PM (h \h m \m) -> 1 h 45 m
//      6/15/2024 1:45:30 PM (h "h" m "m") -> 1 h 45 m
//      6/15/2024 1:45:30 PM (h 'h' m 'm') -> 1 h 45 m
```

Related content

- [How culture affects text formatting](#)
- [Date, Time, DateTime, and DateTimeZone type conversion](#)
- [Date functions](#)
- [DateTime functions](#)
- [DateTimeZone functions](#)
- [Time functions](#)
- [Standard Date and Time format strings](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

How culture affects text formatting

Article • 10/09/2024

Different countries and language groups have different conventions for formatting different kinds of text, such as numbers, dates, and time. In Power Query, *culture* refers to the locale whose conventions are used to format this type of text in Power Query M. This locale is generally composed of the language and the country where the language is spoken, for example "en-US" for the English language spoken in the United States of America.

Default culture

The default culture is set to the system locale (Windows, MacOS) of a particular document where your queries are first authored. For example, if you author your queries in Power Query Desktop, the default culture is defined by the locale set on your local computer. However, if you author your queries in the Power Query Online, the default culture is defined by the locale set on the online service. No matter where you author your query, if you move your query to a different location that uses a different default culture, your query still uses the culture of the original location.

To use the current default culture, no culture setting is required in your Power Query M code.

However, you can change the default culture in the Power Query settings dialog where you create the query. For example, if you are running Power Query from Excel:

1. In Power Query, select **File > Options and settings > Query options**.
2. Under **Current Workbook**, select **Regional Settings**.
3. Select the locale you want to use.

Other versions of Power Query work similarly. In general, within Power Query you select **Options**, which opens the **Options** dialog. Then select **Regional Settings** and select the locale you want to use.

Invariant culture

The invariant culture is culture-insensitive; it's associated with the English language but not with any country or region. You specify the invariant culture by name by using an empty string ("") in functions that include the culture parameter.

Unlike culture-sensitive data, which is subject to change by user customization or by updates to the operating system, invariant culture data is stable over time and across installed cultures and can't be customized by users. This makes the invariant culture particularly useful for operations that require culture-independent results, such as formatting and parsing operations that persist formatted data, or sorting and ordering operations that require that data be displayed in a fixed order regardless of culture.

To use the invariant culture in Power Query M, use the blank text value ("") in the numeric functions that support culture, or (Culture = "") in date and time functions that support culture.

Related content

- [Standard date and time format strings](#)
- [Custom date and time format strings](#)
- [Standard numeric format strings](#)
- [Custom numeric format strings](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Capitalization in Power Query M

Article • 05/30/2025

Working with text data can sometimes be messy. For example, the name of the city Redmond might be represented in a database using different casings ("Redmond", "redmond", and "REDMOND"). This could cause a problem when transforming the data in Power Query because the Power Query M formula language is case sensitive.

Thankfully, Power Query M provides functions to clean and normalize the case of text data. There are functions to convert text to lower case (abc), upper case (ABC), or proper case (Abc). In addition, Power Query M also provides several ways to ignore case altogether.

This article shows you how to change the capitalization of words in text, lists, and tables. It also describes various ways to ignore case while manipulating data in text, lists, and tables. In addition, this article discusses how to sort according to case.

Changing case in text

There are three functions that convert text to lower case, upper case, and proper case. The functions are [Text.Lower](#), [Text.Upper](#), and [Text.Proper](#). The following simple examples demonstrate how these functions can be used in text.

Convert all characters in text to lower case

The following example demonstrates how to convert all characters in a string to lower case.

Power Query M

```
let
    Source = Text.Lower("The quick brown fox jumps over the lazy dog.")
in
    Source
```

This code produces the following output:

```
the quick brown fox jumps over the lazy dog.
```

Convert all characters in text to upper case

The following example demonstrates how to convert all characters in a text string to upper case.

Power Query M

```
let
    Source = Text.Upper("The quick brown fox jumps over the lazy dog.")
in
    Source
```

This code produces the following output:

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Convert all words to initial caps

The following example demonstrates how to convert all words in the sentence to initial capitalization.

Power Query M

```
let
    Source = Text.Proper("The quick brown fox jumps over the lazy dog.")
in
    Source
```

This code produces the following output:

```
The Quick Brown Fox Jumps Over The Lazy Dog.
```

Changing case in lists

When changing case in lists, the most common function to use is [List.Transform](#). The following simple examples demonstrate how this function can be used in lists.

Convert all items to lower case

The following example shows how to change all items in a list to lower case.

Power Query M

```
let
    Source = {"Squash", "Pumpkin", "ApPle", "pear", "orange", "APPLE", "Pear", "pear"},
    #"Lower Case" = List.Transform(Source, Text.Lower)
in
    #"Lower Case"
```

This code produces the following output:

	List
1	squash
2	pumpkin
3	apple
4	pear
5	orange
6	apple
7	pear
8	pear

Convert all items to upper case

The following example demonstrates how to change all items in a list to upper case.

Power Query M

```
let
    Source = {"Squash", "Pumpkin", "ApPlE", "pear", "orange", "APPLE", "Pear", "pear"},
    #"Upper Case" = List.Transform(Source, Text.Upper)
in
    #"Upper Case"
```

This code produces the following output:

	List
1	SQUASH
2	PUMPKIN
3	APPLE
4	PEAR
5	ORANGE
6	APPLE
7	PEAR
8	PEAR

Convert all items to proper case

The following example demonstrates how to change all items in a list to proper case.

Power Query M

```
let
    Source = {"Squash", "Pumpkin", "ApPlE", "pear", "orange", "APPLE", "Pear", "pear"},
    #"Proper Case" = List.Transform(Source, Text.Proper)
in
    #"Proper Case"
```

This code produces the following output:

	List
1	Squash
2	Pumpkin
3	Apple
4	Pear
5	Orange
6	Apple
7	Pear
8	Pear

Changing case in tables

When changing case in tables, the most common function to use is [Table.TransformColumns](#). There's also a function you can use to change the case of text that's contained in a row, called [Table.TransformRows](#). However, this function isn't used as often.

The following simple examples demonstrate how the `Table.TransformColumns` function can be used to change the case in tables.

Convert all items in a table column to lower case

The following example demonstrates how to change all items in a table column to lower case, in this case, the customer names.

Power Query M

```

let
    Source = #table(type table [CUSTOMER = text, FRUIT = text],
    {
        {"Tulga", "Squash"},
        {"suSanna", "Pumpkin"},
        {"LESLIE", "ApPlE"},
        {"Willis", "pear"},
        {"Dilbar", "orange"},
        {"ClaudiA", "APPLE"},
        {"afonso", "Pear"},
        {"SErgio", "pear"}
    }),
    #"Lower Case" = Table.TransformColumns(Source, {"CUSTOMER", Text.Lower})
in
    #"Lower Case"

```

This code produces the following output:

	CUSTOMER	FRUIT
1	tulga	Squash
2	susanna	Pumpkin
3	leslie	ApPlE
4	willis	pear
5	dilbar	orange
6	claudia	APPLE
7	afonso	Pear
8	sergio	pear

Convert all items in a table column to upper case

The following example demonstrates how to change all items in a table column to upper case, in this case, the fruit names.

Power Query M

```

let
    Source = #table(type table [CUSTOMER = text, FRUIT = text],
    {
        {"Tulga", "Squash"},
        {"suSanna", "Pumpkin"},
        {"LESLIE", "ApPlE"},
        {"Willis", "pear"},
        {"Dilbar", "orange"},
        {"ClaudiA", "APPLE"},
        {"afonso", "Pear"},
        {"SErgio", "pear"}
    })

```

```

        {"afonso", "Pear"},
        {"SErgio", "pear"}
    }},
    #"Upper Case" = Table.TransformColumns(Source, {"FRUIT", Text.Upper})
in
    #"Upper Case"

```

This code produces the following output:

	A _C CUSTOMER ▾	A _C FRUIT ▾
1	Tulga	SQUASH
2	suSanna	PUMPKIN
3	LESLIE	APPLE
4	Willis	PEAR
5	Dilbar	ORANGE
6	Claudia	APPLE
7	afonso	PEAR
8	SErgio	PEAR

Convert all items in a table to proper case

The following example demonstrates how to change all items in both of the table columns to proper case.

Power Query M

```

let
    Source = #table(type table [CUSTOMER = text, FRUIT = text],
        {
            {"Tulga", "Squash"},
            {"suSanna", "Pumpkin"},
            {"LESLIE", "ApPlE"},
            {"Willis", "pear"},
            {"Dilbar", "orange"},
            {"Claudia", "APPLE"},
            {"afonso", "Pear"},
            {"SErgio", "pear"}
        }
    ),
    #"Customer Case" = Table.TransformColumns(Source, {"CUSTOMER", Text.Proper}),
    #"Proper Case" = Table.TransformColumns(#"Customer Case", {"FRUIT",
Text.Proper})
in
    #"Proper Case"

```


This code produces the following output:

	 A ^B C CUSTOMER ▼	A ^B C FRUIT ▼
1	Tulga	Squash
2	Susanna	Pumpkin
3	Leslie	Apple
4	Willis	Pear
5	Dilbar	Orange
6	Claudia	Apple
7	Afonso	Pear
8	Sergio	Pear

Ignoring case

In many cases when searching or replacing items, you might need to ignore the case of the item you're looking for. Because the Power Query M formula language is case sensitive, comparisons between items that are identical but have different cases results in identifying the items as being different, not identical. One method of ignoring case involves using the [Comparer.OrdinalIgnoreCase](#) function in functions that include either an `equationCriteria` parameter or a `comparer` parameter. Another method of ignoring case involves using the `IgnoreCase` option (if available) in functions that include an `options` parameter.

Ignoring case in text

Searches in text sometimes require that you ignore case to be able to find all the instances of the searched for text. Text functions generally use the `Comparer.OrdinalIgnoreCase` function in the `comparer` parameter to ignore case when testing for equality.

The following example shows how to ignore case when determining if a sentence contains a specific word, regardless of case.

Power Query M

```
let
    Source = Text.Contains(
        "The rain in spain falls mainly on the plain.",
        "Spain",
        Comparer.OrdinalIgnoreCase
    )
```

```
in
    Source
```

This code produces the following output:

```
true
```

The following example shows how to retrieve the initial position of the last occurrence of the word "the" in the sentence, regardless of case.

Power Query M

```
let
    Source = Text.PositionOf(
        "THE RAIN IN SPAIN FALLS MAINLY ON THE PLAIN.",
        "the",
        Occurrence.Last,
        Comparer.OrdinalIgnoreCase
    )
in
    Source
```

This code produces the following output:

```
34
```

Ignoring case in lists

Any list function that contains an optional `equationCriteria` parameter can use the [Comparer.OrdinalIgnoreCase](#) function to ignore case in the list.

The following example checks whether a list contains a specific item, while ignoring case. In this example, [List.Contains](#) can only compare one item in the list, you can't compare a list to a list. For that, you need to use [List.ContainsAny](#).

Power Query M

```
let
    Source = List.Contains(
        {"Squash", "Pumpkin", "ApPlE", "pear", "orange", "APPLE", "Pear", "pear"},
        "apple",
        Comparer.OrdinalIgnoreCase
    )
in
    Source
```

This code produces the following output:

true

The following examples check whether a list contains all the specified items in the second parameter (`value`), while ignoring case. If any one item isn't contained in the list, such as `cucumber` in the second example, the function returns FALSE.

Power Query M

```
let
    Source = List.ContainsAll(
        {"Squash", "Pumpkin", "ApPlE", "pear", "orange", "APPLE", "Pear", "pear"},
        {"apple", "pear", "squash", "pumpkin"},
        Comparer.OrdinalIgnoreCase
    )
in
    Source
```

This code produces the following output:

true

Power Query M

```
let
    Source = List.ContainsAll(
        {"Squash", "Pumpkin", "ApPlE", "pear", "orange", "APPLE", "Pear", "pear"},
        {"apple", "pear", "squash", "pumpkin", "cucumber"},
        Comparer.OrdinalIgnoreCase
    )
in
    Source
```

This code produces the following output:

false

The following example checks whether any of the items in the list are apples or pears, while ignoring case.

Power Query M

```
let
    Source = List.ContainsAny(
        {"Squash", "Pumpkin", "ApPlE", "PEAR", "orange", "APPLE", "Pear", "peaR"},
        {"apple", "pear"},
        Comparer.OrdinalIgnoreCase
    )
```

```
in
    Source
```

This code produces the following output:

```
true
```

The following example keeps only unique items, while ignoring case.

Power Query M

```
let
    Source = List.Distinct(
        {"Squash", "Pumpkin", "ApPlE", "PEAR", "orange", "APPLE", "Pear", "peaR"},
        Comparer.OrdinalIgnoreCase
    )
in
    Source
```

This code produces the following output:

	List
1	Squash
2	Pumpkin
3	ApPlE
4	PEAR
5	orange

In the previous example, the output displays the case of the first unique item found in the list. So, although there are two apples (`ApPlE` and `APPLE`), only the first example found is displayed.

The following example keeps only unique items while ignoring case, but also returns all lower case results.

Power Query M

```
let
    Source = List.Distinct(
        {"Squash", "Pumpkin", "ApPlE", "PEAR", "orange", "APPLE", "Pear", "peaR"},
        Comparer.OrdinalIgnoreCase
    ),
    #"Lower Case" = List.Transform(Source, Text.Lower)
in
    #"Lower Case"
```

This code produces the following output:

	List
1	squash
2	pumpkin
3	apple
4	pear
5	orange

Ignoring case in tables

Tables have several ways to ignore case. Table functions such as [Table.Contains](#), [Table.Distinct](#), and [Table.PositionOf](#) all contain `equationCriteria` parameters. These parameters can use the [Comparer.OrdinalIgnoreCase](#) function to ignore case in tables, in much the same way as the lists in the previous sections. Table functions, such as [Table.MatchesAnyRows](#) that contain a `condition` parameter can also use [Comparer.OrdinalIgnoreCase](#) wrapped in other table functions to ignore case. Other table functions, specifically for fuzzy matching, can use the `IgnoreCase` option.

The following example demonstrates how to select specific rows that contain the word "pear" while ignoring case. This example uses the `condition` parameter of [Table.SelectRows](#) with [Text.Contains](#) as the conditional to make the comparisons while ignoring case.

Power Query M

```
let
    Source = #table(type table[CUSTOMER = text, FRUIT = text],
        {
            {"Tulga", "Squash"},
            {"suSanna", "Pumpkin"},
            {"LESLIE", "ApPlE"},
            {"Willis", "pear"},
            {"Dilbar", "orange"},
            {"Claudia", "APPLE"},
            {"afonso", "Pear"},
            {"SErgio", "pear"}
        }
    ),
    #"Select Rows" = Table.SelectRows(
        Source, each Text.Contains([FRUIT], "pear", Comparer.OrdinalIgnoreCase))
in
    #"Select Rows"
```

This code produces the following output:

	A ^B _C CUSTOMER ▾	A ^B _C FRUIT ▾
1	Willis	pear
2	afonso	Pear
3	SErgio	pear

The following sample shows how to determine if any of the rows in a table contain a `pear` in the `FRUIT` column. This example uses `Comparer.OrdinalIgnoreCase` in a `Text.Contains` function using the `condition` parameter of the `Table.MatchesAnyRows` function.

Power Query M

```
let
    Source = #table(type table [CUSTOMER = text, FRUIT = text],
        {
            {"Tulga", "Squash"},
            {"suSanna", "Pumpkin"},
            {"LESLIE", "ApPle"},
            {"Willis", "PEAR"},
            {"Dilbar", "orange"},
            {"Claudia", "APPLE"},
            {"afonso", "Pear"},
            {"SErgio", "peAR"}
        }
    ),
    #"Select Rows" = Table.MatchesAnyRows(Source,
        each Text.Contains([FRUIT], "pear", Comparer.OrdinalIgnoreCase))
in
    #"Select Rows"
```

This code produces the following output:

true

The following example demonstrates how to take a table with values entered by users that contains a column listing their favorite fruits, using no set format. This column is first fuzzy matched to extract the names of their favorite fruit, which is then displayed in its own column, named **Cluster**. Then the **Cluster** column is examined to determine the different distinct fruits that are in the column. Once the unique fruits are determined, a final step is to change all of the fruit names to lower case.

Power Query M

```
let
    // Load a table of user's favorite fruits into Source
    Source = #table(type table [Fruit = text], {"blueberries"},
        {"Blue berries are simply the best"}, {"strawberries"}, {"Strawberries =
    <3"},
```

```

{"Apples"}, {"'sples"}, {""4ppl3s"}, {"Bananas"}, {"fav fruit is bananas"},
{"Banas"}, {"My favorite fruit, by far, is Apples. I simply love them!"}}
),
// Create a Cluster column and fuzzy match the fruits into that column
#"Cluster fuzzy match" = Table.AddFuzzyClusterColumn(
    Source, "Fruit", "Cluster",
    [IgnoreCase = true, IgnoreSpace = true, Threshold = 0.5]
),
// Find the distinct fruits from the Cluster column
#"Ignore cluster case" = Table.Distinct(
    Table.SelectColumns(#"Cluster fuzzy match", "Cluster"),
    Comparer.OrdinalIgnoreCase
),
// Set all of the distinct fruit names to lower case
#"Set lower case" = Table.TransformColumns(#"Ignore cluster case",
    {"Cluster", Text.Lower}
)
in
#"Set lower case"

```

This code produces the following output:

	Cluster
1	blueberries
2	strawberries
3	apples
4	bananas

Case and sorting

Lists and tables can both be sorted using either [List.Sort](#) or [Table.Sort](#), respectively. However, sorting text depends on the case of the associated items in the list or table to determine the actual sort order (either ascending or descending).

The most common form of sorting uses text that is either all lower case, all upper case, or proper case. If there is a mix of these cases, the ascending sort order is as follows:

1. Any text in the list or table column that begins with a capital letter is first.
2. If there is matching text, but one is proper case and another is all upper case, the upper case version is first.
3. Lower case is then sorted.

For descending order, the previously listed steps are processed in reverse.

For example, the following sample has a mixture of all lower case, all upper case, and proper case text to be sorted in ascending order.

Power Query M

```
let
    Source = { "Alpha", "Beta", "Zulu", "ALPHA", "gamma", "alpha",
               "beta", "Gamma", "Sierra", "zulu", "GAMMA", "ZULU" },
    SortedList = List.Sort(Source, Order.Ascending)
in
    SortedList
```

This code produces the following output:

	List
1	ALPHA
2	Alpha
3	Beta
4	GAMMA
5	Gamma
6	Sierra
7	ZULU
8	Zulu
9	alpha
10	beta
11	gamma
12	zulu

Although not common, you might have a mix of upper and lower case letters in text to sort. The ascending sort order in this case is:

1. Any text in the list or table column that begins with a capital letter is first.
2. If there is matching text, the text with the maximum number of upper case letters to the left is done next.
3. Lower case is then sorted, with the maximum number of upper case letters to the right done first.

In any case, it might be more convenient to convert the text to a consistent case before sorting.

Power BI Desktop normalization

Power Query M is case sensitive and distinguishes between different capitalizations of the same text. For example, "Foo", "foo", and "FOO" are treated as different. However, when the data is loaded into Power BI Desktop, the text values are normalized, meaning Power BI Desktop treats them as the same value regardless of their capitalization. Therefore, if you need to transform data while maintaining case sensitivity in your data, you should handle data transformation in Power Query before loading the data to Power BI Desktop.

For example, the following table in Power Query shows different cases in each row of the table.

	ABC Column1
1	Foo
2	TOO
3	foo
4	tOo
5	FOO
6	Too
7	fOo
8	too

When this table is loaded into Power BI Desktop, the text values become normalized, resulting in the following table.

Clipboard	
	✕ ✓
	Column1
	Foo
	TOO
	Foo
	TOO
	Foo
	TOO
	Foo
	TOO

The first instance of "foo" and the first instance of "too" determine the case of "foo" and "too" throughout the rest of the rows in the Power BI Desktop table. In this example, all instances of "foo" are normalized to the value "Foo" and all instances of "too" are normalized to the value "TOO".

Power Query M function reference

06/09/2025

The Power Query M function reference includes articles for each of the over 700 functions. These reference articles are auto-generated from in-product help. To learn more about functions and how they work in an expression, go to [Understanding Power Query M functions](#).

Functions by category

- [Accessing data functions](#)
- [Binary functions](#)
- [Combiner functions](#)
- [Comparer functions](#)
- [Date functions](#)
- [DateTime functions](#)
- [DateTimeZone functions](#)
- [Duration functions](#)
- [Error handling](#)
- [Expression functions](#)
- [Function values](#)
- [List functions](#)
- [Lines functions](#)
- [Logical functions](#)
- [Number functions](#)
- [Record functions](#)
- [Replacer functions](#)
- [Splitter functions](#)
- [Table functions](#)
- [Text functions](#)
- [Time functions](#)
- [Type functions](#)
- [Uri functions](#)
- [Value functions](#)

Understanding Power Query M functions

Article • 02/14/2025

In the Power Query M formula language, a *function* is a mapping from a set of input values to a single output value. A function is written by first naming the function parameters, and then providing an expression to compute the result of the function. The body of the function follows the goes-to ($\Rightarrow$) symbol. Optionally, type information can be included on parameters and the function return value. A function is defined and invoked in the body of a **let** statement. Parameters and/or return value can be implicit or explicit. Implicit parameters and/or return value are of type **any**. Type **any** is similar to an object type in other languages. All types in M derive from type **any**.

A function is a value just like a number or a text value, and can be included in-line just like any other expression. The following example shows a function that is the value of an **Add** variable, which is then invoked, or executed, from several other variables. When a function is invoked, a set of values are specified that logically substitute for the required set of input values within the function body expression.

Example - Explicit parameters and return value

Power Query M

```
let
    AddOne = (x as number) as number => x + 1,
    //additional expression steps
    CalcAddOne = AddOne(5)
in
    CalcAddOne
```

Example - Implicit parameters and return value

Power Query M

```
let
    Add = (x, y) => x + y,
    AddResults =
        [
            OnePlusOne = Add(1, 1),      // equals 2
            OnePlusTwo = Add(1, 2)      // equals 3
        ]
in
    AddResults
```

Find the first element of a list greater than 5, or null otherwise

Power Query M

```
let
    FirstGreaterThan5 = (list) =>
        let
            GreaterThan5 = List.Select(list, (n) => n > 5),
            First = List.First(GreaterThan5)
        in
            First,
    Results =
    [
        Found      = FirstGreaterThan5({3,7,9}), // equals 7
        NotFound   = FirstGreaterThan5({1,3,4})  // equals null
    ]
in
    Results
```

Functions can be used recursively. In order to recursively reference the function, prefix the identifier with @.

Power Query M

```
let
    fact = (num) => if num = 0 then 1 else num * @fact (num-1)
in
    fact(5) // equals 120
```

Each keyword

The **each** keyword is used to easily create simple functions. `each ...` is syntactic sugar for a function signature that takes the `_` parameter `(_) => ...`.

The **each** keyword is useful when combined with the lookup operator, which is applied by default to `_`. For example, `each [CustomerID]` is the same as `each _[CustomerID]`, which is the same as `(_) => _[CustomerID]`.

Example - Using each in table row filter

Power Query M

```
Table.SelectRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"] ,
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"] ,
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    })),
```

```
        each [CustomerID] = 2
    )[Name]

    // equals "Jim"
```

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Accessing data functions

Article • 01/17/2025

These functions access data and return table values. Most of these functions return a table value called a **navigation table**. Navigation tables are primarily used by the Power Query user interface to provide a navigation experience over the potentially large hierarchical data sets returned.

 Expand table

Name	Description
AccessControlEntry.ConditionTolIdentities	Returns a list of identities that the condition will accept.
Access.Database	Returns a structural representation of a Microsoft Access database.
ActiveDirectory.Domains	Returns a list of Active Directory domains in the same forest as the specified domain or of the current machine's domain if none is specified.
AdobeAnalytics.Cubes	Returns the report suites in Adobe Analytics.
AdoDotNet.DataSource	Returns the schema collection for an ADO.NET data source.
AdoDotNet.Query	Returns the result of running a native query on an ADO.NET data source.
AnalysisServices.Database	Returns a table of multidimensional cubes or tabular models from the Analysis Services database.
AnalysisServices.Databases	Returns the Analysis Services databases on a particular host.
AzureStorage.BlobContents	Returns the content of the specified blob from an Azure storage vault.
AzureStorage.Blobs	Returns a navigational table containing the containers found in the specified account from an Azure storage vault. Each row has the container name and a link to the container blobs.
AzureStorage.DataLake	Returns a navigational table containing the documents found in the specified container and its subfolders from Azure Data Lake Storage.

Name	Description
AzureStorage.DataLakeContents	Returns the content of the specified file from an Azure Data Lake Storage filesystem.
AzureStorage.Tables	Returns a navigational table containing the tables found in the specified account from an Azure storage vault. Each row contains a link to the azure table.
Cdm.Contents	This function is unavailable because it requires .NET 4.5.
Csv.Document	Returns the contents of the CSV document as a table using the specified encoding.
Cube.AddAndExpandDimensionColumn	Merges the specified dimension table into the cube's filter context and changes the dimensional granularity of the filter context by expanding the specified set of dimension attributes.
Cube.AddMeasureColumn	AAAdds a column to the cube that contains the results of the measure applied in the row context of each row.
Cube.ApplyParameter	Returns a cube after applying a parameter to it.
Cube.AttributeMemberId	Returns the unique member identifier from members property value.
Cube.AttributeMemberProperty	Returns a property of a dimension attribute.
Cube.CollapseAndRemoveColumns	Changes the dimensional granularity of the filter context for the cube by collapsing the attributes mapped to the specified columns.
Cube.Dimensions	Returns a table containing the set of available dimensions.
Cube.DisplayFolders	Returns a nested tree of tables representing the display folder hierarchy of the objects (for example, dimensions and measures).
Cube.MeasureProperties	Returns a table containing the set of available measure properties that are expanded in the cube.
Cube.MeasureProperty	Returns a property of a measure (cell property).
Cube.Measures	Returns a table containing the set of available measures.
Cube.Parameters	Returns a table containing the set of parameters that can be applied to the cube.

Name	Description
Cube.Properties	Returns a table containing the set of available properties for dimensions that are expanded in the cube.
Cube.PropertyKey	Returns the key of a property.
Cube.ReplaceDimensions	Replaces the set of dimensions returned by Cube.Dimensions .
Cube.Transform	Applies a list of cube functions.
DB2.Database	Returns a table of SQL tables and views available in a Db2 database.
DeltaLake.Metadata	Given a Delta Lake table, returns the log entries for that table.
DeltaLake.Table	Returns the contents of the Delta Lake table.
Essbase.Cubes	Returns the cubes in an Essbase instance grouped by Essbase server.
Excel.CurrentWorkbook	Returns the contents of the current Excel workbook.
Excel.Workbook	Returns the contents of the Excel workbook.
Exchange.Contents	Returns a table of contents from a Microsoft Exchange account.
File.Contents	Returns the contents of the specified file as binary.
Folder.Contents	Returns a table containing the properties and contents of the files and folders found in the specified folder.
Folder.Files	Returns a table containing the properties and contents of the files found in the specified folder and subfolders. Each row contains properties of the folder or file and a link to its content.
GoogleAnalytics.Accounts	Returns the Google Analytics accounts for the current credential.
Hdfs.Contents	Returns a table containing the properties and contents of the files and folders found in the specified folder from a Hadoop file system. Each row contains properties of the folder or file and a link to its content.
Hdfs.Files	Returns a table containing the properties and contents of the files found in the specified folder and

Name	Description
	subfolders from a Hadoop file system. Each row contains properties of the file and a link to its content.
HdInsight.Containers	Returns a navigational table containing the containers found in the specified account from an Azure storage vault. Each row has the container name and table containing its files.
HdInsight.Contents	Returns a navigational table containing the containers found in the specified account from an Azure storage vault. Each row has the container name and table containing its files.
HdInsight.Files	Returns a table containing the properties and contents of the blobs found in the specified container from an Azure storage vault. Each row contains properties of the file/folder and a link to its content.
Html.Table	Returns a table containing the results of running the specified CSS selectors against the provided HTML.
Identity.From	Creates an identity.
Identity.IsMemberOf	Determines whether an identity is a member of an identity collection.
IdentityProvider.Default	The default identity provider for the current host.
Informix.Database	Returns a table of SQL tables and views available in an Informix database.
Json.Document	Returns the content of the JSON document. The contents can be directly passed to the function as text, or it can be the binary value returned by a function like File.Contents .
Json.FromValue	Produces a JSON representation of a given value value with a text encoding specified by encoding.
MySQL.Database	Returns a table of SQL tables, views, and stored scalar functions available in a MySQL database.
OData.Feed	Returns a table of OData feeds offered by an OData service.
Odbc.DataSource	Returns a table of SQL tables and views from the ODBC data source.
Odbc.InferOptions	Returns the result of trying to infer SQL capabilities for an ODBC driver.

Name	Description
Odbc.Query	Returns the result of running a native query on an ODBC data source.
OleDb.DataSource	Returns a table of SQL tables and views from the OLE DB data source.
OleDb.Query	Returns the result of running a native query on an OLE DB data source.
Oracle.Database	Returns a table of SQL tables and views from the Oracle database.
Pdf.Tables	Returns any tables found in a PDF file.
PostgreSQL.Database	Returns a table of SQL tables and views available in a PostgreSQL database.
RData.FromBinary	Returns a record of data frames from the RData file.
Salesforce.Data	Returns the objects from the Salesforce account.
Salesforce.Reports	Returns the reports from the Salesforce account.
SapBusinessWarehouse.Cubes	Returns the InfoCubes and queries in an SAP Business Warehouse system grouped by InfoArea.
SapHana.Database	Returns the packages in an SAP HANA database.
SharePoint.Contents	Returns a table containing content from a SharePoint site. Each row contains properties of the folder or file and a link to its content.
SharePoint.Files	Returns a table containing documents from a SharePoint site. Each row contains properties of the folder or file and a link to its content.
SharePoint.Tables	Returns a table containing content from a SharePoint List.
Soda.Feed	Returns a table from the contents at the specified URL formatted according to the SODA 2.0 API. The URL must point to a valid SODA-compliant source that ends in a .csv extension.
Sql.Database	Returns a table of SQL tables, views, and stored functions from the SQL Server database.
Sql.Databases	Returns a table of databases on a SQL Server.
Sybase.Database	Returns a table of SQL tables and views available in a

Name	Description
	Sybase database.
Teradata.Database	Returns a table of SQL tables and views from the Teradata database.
WebAction.Request	Creates an action that, when executed, will return the results of performing a method request against url using HTTP as a binary value.
Web.BrowserContents	Returns the HTML for the specified URL, as viewed by a web browser.
Web.Contents	Returns the contents downloaded from the URL as binary.
Web.Headers	Returns the HTTP headers downloaded from the URL as a record value.
Web.Page	Returns the contents of the HTML document broken into its constituent structures, as well as a representation of the full document and its text after removing tags.
Xml.Document	Returns the contents of the XML document as a hierarchical table (list of records).
Xml.Tables	Returns the contents of an XML document as a nested collection of flattened tables.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

AccessControlEntry.ConditionToIdentities

06/19/2025

Syntax

```
AccessControlEntry.ConditionToIdentities(identityProvider as function, condition as function) as list
```

About

Using the specified `identityProvider`, converts the `condition` into the list of identities for which `condition` would return `true` in all authorization contexts with `identityProvider` as the identity provider. An error is raised if it is not possible to convert `condition` into a list of identities, for example if `condition` consults attributes other than user or group identities to make a decision.

Note that the list of identities represents the identities as they appear in `condition` and no normalization (such as group expansion) is performed on them.

Access.Database

06/19/2025

Syntax

```
Access.Database(database as binary, optional options as nullable record) as table
```

About

Returns a structural representation of an Access database, `database`. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is false).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.

The record parameter is specified as [option1 = value1, option2 = value2...], for example.

ActiveDirectory.Domains

06/19/2025

Syntax

```
ActiveDirectory.Domains(optional forestRootDomainName as nullable text) as table
```

About

Returns a list of Active Directory domains in the same forest as the specified domain or of the current machine's domain if none is specified.

AdobeAnalytics.Cubes

06/19/2025

Syntax

```
AdobeAnalytics.Cubes(optional options as nullable record) as table
```

About

Returns a table of multidimensional packages from Adobe Analytics. An optional record parameter, `options`, may be specified to control the following options:

- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).
- `MaxRetryCount`: The number of retries to perform when polling for the result of the query. The default value is 120.
- `RetryInterval`: The duration of time between retry attempts. The default value is 1 second.
- `Implementation`: Specifies Adobe Analytics API version. Valid values are: "2.0". Default uses API version 1.4.

AdoDotNet.DataSource

06/19/2025

Syntax

```
AdoDotNet.DataSource(providerName as text, connectionString as any, optional options as nullable record) as table
```

About

Returns the schema collection for the ADO.NET data source with provider name `providerName` and connection string `connectionString`. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth`: A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.
- `TypeMap`

AdoDotNet.Query

06/19/2025

Syntax

```
AdoDotNet.Query(providerName as text, connectionString as any, query as text,  
optional options as nullable record) as table
```

About

Returns the result of running `query` with the connection string `connectionString` using the ADO.NET provider `providerName`. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth`: A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.

AnalysisServices.Database

06/19/2025

Syntax

```
AnalysisServices.Database(server as text, database as text, optional options as nullable record) as table
```

About

Returns a table of multidimensional cubes or tabular models from the Analysis Services database `database` on server `server`. An optional record parameter, `options`, may be specified to control the following options:

- `Query`: A native MDX query used to retrieve data.
- `TypedMeasureColumns`: A logical value indicating if the types specified in the multidimensional or tabular model will be used for the types of the added measure columns. When set to false, the type "number" will be used for all measure columns. The default value for this option is false.
- `Culture`: A culture name specifying the culture for the data. This corresponds to the 'Locale Identifier' connection string property.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is driver-dependent.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `SubQueries`: A number (0, 1 or 2) that sets the value of the "SubQueries" property in the connection string. This controls the behavior of calculated members on subselects or subcubes. (The default value is 2).
- `Implementation`

Related content

- [How culture affects text formatting](#)

AnalysisServices.Databases

06/19/2025

Syntax

```
AnalysisServices.Databases(server as text, optional options as nullable record) as table
```

About

Returns databases on an Analysis Services instance, **server**. An optional record parameter, **options**, may be provided to specify additional properties. The record can contain the following fields:

- **TypedMeasureColumns**: A logical value indicating if the types specified in the multidimensional or tabular model will be used for the types of the added measure columns. When set to false, the type "number" will be used for all measure columns. The default value for this option is false.
- **Culture**: A culture name specifying the culture for the data. This corresponds to the 'Locale Identifier' connection string property.
- **CommandTimeout**: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is driver-dependent.
- **ConnectionTimeout**: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- **SubQueries**: A number (0, 1 or 2) that sets the value of the "SubQueries" property in the connection string. This controls the behavior of calculated members on subselects or subcubes. (The default value is 2).
- **Implementation**

Related content

- [How culture affects text formatting](#)

AzureStorage.BlobContents

06/19/2025

Syntax

```
AzureStorage.BlobContents(url as text, optional options as nullable record) as binary
```

About

Returns the content of the blob at the URL, `url`, from an Azure storage vault. `options` may be specified to control the following options:

- `BlockSize`: The number of bytes to read before waiting on the data consumer. The default value is 4 MB.
- `RequestSize`: The number of bytes to try to read in a single HTTP request to the server. The default value is 4 MB.
- `ConcurrentRequests`: The `ConcurrentRequests` option supports faster download of data by specifying the number of requests to be made in parallel, at the cost of memory utilization. The memory required is $(\text{ConcurrentRequest} * \text{RequestSize})$. The default value is 16.

AzureStorage.Blobs

06/19/2025

Syntax

```
AzureStorage.Blobs(account as text, optional options as nullable record) as table
```

About

Returns a navigational table containing a row for each container found at the account URL, `account`, from an Azure storage vault. Each row contains a link to the container blobs. `options` may be specified to control the following options:

- `BlockSize`: The number of bytes to read before waiting on the data consumer. The default value is 4 MB.
- `RequestSize`: The number of bytes to try to read in a single HTTP request to the server. The default value is 4 MB.
- `ConcurrentRequests`: The ConcurrentRequests option supports faster download of data by specifying the number of requests to be made in parallel, at the cost of memory utilization. The memory required is (ConcurrentRequest * RequestSize). The default value is 16.

AzureStorage.DataLake

06/19/2025

Syntax

```
AzureStorage.DataLake(endpoint as text, optional options as nullable record) as table
```

About

Returns a navigational table containing the documents found in the specified container and its subfolders at the account URL, `endpoint`, from an Azure Data Lake Storage filesystem. `options` may be specified to control the following options:

- `BlockSize`: The number of bytes to read before waiting on the data consumer. The default value is 4 MB.
- `RequestSize`: The number of bytes to try to read in a single HTTP request to the server. The default value is 4 MB.
- `ConcurrentRequests`: The `ConcurrentRequests` option supports faster download of data by specifying the number of requests to be made in parallel, at the cost of memory utilization. The memory required is $(\text{ConcurrentRequest} * \text{RequestSize})$. The default value is 16.
- `HierarchicalNavigation`: A logical (true/false) that controls whether the files are returned in a tree-like directory view or in a flat list. The default value is false.

AzureStorage.DataLakeContents

06/19/2025

Syntax

```
AzureStorage.DataLakeContents(url as text, optional options as nullable record) as binary
```

About

Returns the content of the file at the URL, `url`, from an Azure Data Lake Storage filesystem.

`options` may be specified to control the following options:

- `BlockSize`: The number of bytes to read before waiting on the data consumer. The default value is 4 MB.
- `RequestSize`: The number of bytes to try to read in a single HTTP request to the server. The default value is 4 MB.
- `ConcurrentRequests`: The `ConcurrentRequests` option supports faster download of data by specifying the number of requests to be made in parallel, at the cost of memory utilization. The memory required is (`ConcurrentRequest * RequestSize`). The default value is 16.

AzureStorage.Tables

06/19/2025

Syntax

```
AzureStorage.Tables(account as text, optional options as nullable record) as table
```

About

Returns a navigational table containing a row for each table found at the account URL, `account`, from an Azure storage vault. Each row contains a link to the azure table. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `Timeout`: A duration that controls how long to wait before abandoning the request to the server. The default value is source-specific.

Cdm.Contents

06/19/2025

Syntax

```
Cdm.Contents(table as table) as table
```

About

This function is unavailable in the current context.

Csv.Document

06/19/2025

Syntax

```
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
```

About

Returns the contents of the CSV document as a table.

- `columns` can be null, the number of columns, a list of column names, a table type, or an options record.
- `delimiter` can be a single character, a list of characters, or the value `""`, which indicates rows should be split by consecutive whitespace characters. Default: `","`.
- Refer to [ExtraValues.Type](#) for the supported values of `extraValues`.
- `encoding` specifies the [text encoding](#) type.

If a record is specified for `columns` (and `delimiter`, `extraValues`, and `encoding` are null), the following record fields may be provided:

- `Delimiter`: A single character column delimiter. Default: `","`.
- `Columns`: Can be null, the number of columns, a list of column names, or a table type. If the number of columns is lower than the number found in the input, the additional columns will be ignored. If the number of columns is higher than the number found in the input, the additional columns will be null. When not specified, the number of columns will be determined by what is found in the input.
- `Encoding`: The text encoding of the file. Default: 65001 (UTF-8).
- `CsvStyle`: Specifies how quotes are handled.
 - [CsvStyle QuoteAfterDelimiter](#) (default): Quotes in a field are only significant immediately following the delimiter.
 - [CsvStyle QuoteAlways](#): Quotes in a field are always significant, regardless of where they appear.
- `QuoteStyle`: Specifies how quoted line breaks are handled.
 - [QuoteStyle.Csv](#) (default): Quoted line breaks are treated as part of the data, not as the end of the current row.

- [QuoteStyle.None](#): All line breaks are treated as the end of the current row, even when they occur inside a quoted value.
- `IncludeByteOrderMark`: A logical value indicating whether to include a Byte Order Mark (BOM) at the beginning of the CSV output. When set to `true`, the BOM is written (for example, UTF-8 BOM: `0xEF 0xBB 0xBF`); when set to `false`, no BOM is included. This option is applicable only in output scenarios. Default is `false`.
- `ExtraValues`: Refer to [ExtraValues.Type](#) for the supported values of `ExtraValues`.

Example 1

Process CSV text with column headers.

Usage

Power Query M

```
let
    csv = Text.Combine({"OrderID,Item", "1,Fishing rod", "2,1 lb. worms"}, "#(cr)#(lf)")
in
    Table.PromoteHeaders(Csv.Document(csv))
```

Output

Power Query M

```
Table.FromRecords({
    [OrderID = "1", Item = "Fishing rod"],
    [OrderID = "2", Item = "1 lb. worms"]
})
```

Example 2

Process CSV text with multiple delimiter characters. In this example, the third parameter specifies the delimiter pattern `#|#` to use instead of the default.

Usage

Power Query M

```
let
    csv = Text.Combine({"OrderID#|#Color", "1#|#Red", "2#|#Blue"}, "#(cr)#(lf)")
```

```
in
```

```
Table.PromoteHeaders(Csv.Document(csv, null, "#|#"))
```

Output

Power Query M

```
Table.FromRecords({  
    [OrderID = "1", Color = "Red"],  
    [OrderID = "2", Color = "Blue"]  
})
```

Cube.AddAndExpandDimensionColumn

06/19/2025

Syntax

```
Cube.AddAndExpandDimensionColumn(cube as table, dimensionSelector as any,  
attributeNames as list, optional newColumnNames as any) as table
```

About

Merges the specified dimension table, `dimensionSelector`, into the cube's, `cube`, filter context and changes the dimensional granularity by expanding the specified set, `attributeNames`, of dimension attributes. The dimension attributes are added to the tabular view with columns named `newColumnNames`, or `attributeNames` if not specified.

Cube.AddMeasureColumn

06/19/2025

Syntax

```
Cube.AddMeasureColumn(cube as table, column as text, measureSelector as any) as table
```

About

Adds a column with the name `column` to the `cube` that contains the results of the measure `measureSelector` applied in the row context of each row. Measure application is affected by changes to dimension granularity and slicing. Measure values will be adjusted after certain cube operations are performed.

Cube.ApplyParameter

06/19/2025

Syntax

```
Cube.ApplyParameter(cube as table, parameter as any, optional arguments as nullable list) as table
```

About

Returns a cube after applying `parameter` with `arguments` to `cube`.

Cube.AttributeMemberId

06/19/2025

Syntax

```
Cube.AttributeMemberId(attribute as any) as any
```

About

Returns the unique member identifier from a member property value. `attribute`. Returns null for any other values.

Cube.AttributeMemberProperty

06/19/2025

Syntax

```
Cube.AttributeMemberProperty(attribute as any, propertyName as text) as any
```

About

Returns the property `propertyName` of dimension attribute `attribute`.

Cube.CollapseAndRemoveColumns

06/19/2025

Syntax

```
Cube.CollapseAndRemoveColumns(cube as table, columnNames as list) as table
```

About

Changes the dimensional granularity of the filter context for the `cube` by collapsing the attributes mapped to the specified columns `columnNames`. The columns are also removed from the tabular view of the cube.

Cube.Dimensions

06/19/2025

Syntax

```
Cube.Dimensions(cube as table) as table
```

About

Returns a table containing the set of available dimensions within the `cube`. Each dimension is a table containing a set of dimension attributes and each dimension attribute is represented as a column in the dimension table. Dimensions can be expanded in the cube using [Cube.AddAndExpandDimensionColumn](#).

Cube.DisplayFolders

06/19/2025

Syntax

```
Cube.DisplayFolders(cube as table) as table
```

About

Returns a nested tree of tables representing the display folder hierarchy of the objects (for example, dimensions and measures) available for use in the `cube`.

Cube.MeasureProperties

06/19/2025

Syntax

```
Cube.MeasureProperties(cube as table) as table
```

About

Returns a table containing the set of available properties for measures that are expanded in the cube.

Cube.MeasureProperty

06/19/2025

Syntax

```
Cube.MeasureProperty(measure as any, propertyName as text) as any
```

About

Returns the property `propertyName` of measure `measure`.

Cube.Measures

06/19/2025

Syntax

```
Cube.Measures(cube as any) as table
```

About

Returns a table containing the set of available measures within the `cube`. Each measure is represented as a function. Measures can be applied to the cube using `Cube.AddMeasureColumn`.

Cube.Parameters

06/19/2025

Syntax

```
Cube.Parameters(cube as table) as table
```

About

Returns a table containing the set of parameters that can be applied to `cube`. Each parameter is a function that can be invoked to get `cube` with the parameter and its arguments applied.

Cube.Properties

06/19/2025

Syntax

```
Cube.Properties(cube as table) as table
```

About

Returns a table containing the set of available properties for dimensions that are expanded in the cube.

Cube.PropertyKey

06/19/2025

Syntax

```
Cube.PropertyKey(property as any) as any
```

About

Returns the key of property `property`.

Cube.ReplaceDimensions

06/19/2025

Syntax

```
Cube.ReplaceDimensions(cube as table, dimensions as table) as table
```

About

Replaces the set of dimensions returned by `Cube.Dimensions`. For example, this function can be used to add an ID column to a dimension attribute, so that the data source can group on the ID rather than the displayed value.

Cube.Transform

06/19/2025

Syntax

```
Cube.Transform(cube as table, transforms as list) as table
```

About

Applies the list cube functions, `transforms`, on the `cube`.

DB2.Database

06/19/2025

Syntax

```
DB2.Database(server as text, database as text, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views available in a Db2 database on server `server` in the database instance named `database`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `Query`: A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).
- `Implementation`: Specifies the internal database provider implementation to use. Valid values are: "IBM" and "Microsoft".
- `BinaryCodePage`: A number for the CCSID (Coded Character Set Identifier) to decode Db2 FOR BIT binary data into character strings. Applies to Implementation = "Microsoft". Set 0 to disable conversion (default). Set 1 to convert based on database encoding. Set other CCSID number to convert to application encoding.
- `PackageCollection`: Specifies a string value for package collection (default is "NULLID") to enable use of shared packages required to process SQL statements. Applies to Implementation = "Microsoft".

- `UseDb2ConnectGateway`: Specifies whether the connection is being made through a Db2 Connect gateway. Applies to Implementation = "Microsoft".

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

DeltaLake.Metadata

06/19/2025

Syntax

```
DeltaLake.Metadata(table as table) as table
```

About

Given a Delta Lake table, returns the log entries for that table.

DeltaLake.Table

06/19/2025

Syntax

```
DeltaLake.Table(directory as table, optional options as nullable record) as any
```

About

Returns the contents of the Delta Lake table.

Essbase.Cubes

06/19/2025

Syntax

```
Essbase.Cubes(url as text, optional options as nullable record) as table
```

About

Returns a table of cubes grouped by Essbase server from an Essbase instance at APS server `url`. An optional record parameter, `options`, may be specified to control the following options:

- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.

Excel.CurrentWorkbook

06/19/2025

Syntax

```
Excel.CurrentWorkbook() as table
```

About

Returns the contents of the current Excel workbook. It returns tables, named ranges, and dynamic arrays. Unlike [Excel.Workbook](#), it does not return sheets.

Excel.Workbook

06/19/2025

Syntax

```
Excel.Workbook(workbook as binary, optional useHeaders as any, optional delayTypes as nullable logical) as table
```

About

Returns the contents of the Excel workbook.

- `useHeaders` can be null, a logical (true/false) value indicating whether the first row of each returned table should be treated as a header, or an options record. Default: false.
- `delayTypes` can be null or a logical (true/false) value indicating whether the columns of each returned table should be left untyped. Default: false.

If a record is specified for `useHeaders` (and `delayTypes` is null), the following record fields may be provided:

- `UseHeaders`: Can be null or a logical (true/false) value indicating whether the first row of each returned table should be treated as a header. Default: false.
- `DelayTypes`: Can be null or a logical (true/false) value indicating whether the columns of each returned table should be left untyped. Default: false.
- `InferSheetDimensions`: Can be null or a logical (true/false) value indicating whether the area of a worksheet that contains data should be inferred by reading the worksheet itself, rather than by reading the dimensions metadata from the file. This can be useful in cases where the dimensions metadata is incorrect. Note that this option is only supported for Open XML Excel files, not for legacy Excel files. Default: false.

ⓘ Note

The `useHeaders` parameter or the `UseHeaders` record field converts numbers and dates to text using the current culture, and thus behaves differently when run in environments with different operating system cultures set. We recommend using [Table.PromoteHeaders](#) instead. For example, instead of using `Excel.Workbook(File.Contents("C:\myfile.xlsx", true, true))` OR `Excel.Workbook(File.Contents("C:\myfile.xlsx", [UseHeaders = true],`

```
null)), use Table.PromoteHeaders(Excel.Workbook(File.Contents("C:\myfile.xlsx", null, true), [PromoteAllScalars = true])) instead.
```

Example 1

Return the contents of Sheet1 from an Excel workbook.

Usage

Power Query M

```
Excel.Workbook(File.Contents("C:\Book1.xlsx"), null, true){[Item="Sheet1"]}[Data]
```

Output

Power Query M

```
Table.FromRecords({  
    [Column1 = "ID", Column2 = "Name", Column3 = "Phone"],  
    [Column1 = 1, Column2 = "Bob", Column3 = "123-4567"],  
    [Column1 = 3, Column2 = "Pam", Column3 = "543-7890"],  
    [Column1 = 2, Column2 = "Jim", Column3 = "987-6543"]  
})
```

Exchange.Contents

06/19/2025

Syntax

```
Exchange.Contents (optional mailboxAddress as nullable text) as table
```

About

Returns a table of contents from the Microsoft Exchange account `mailboxAddress`. If `mailboxAddress` is not specified, the default account for the credential will be used.

File.Contents

06/19/2025

Syntax

```
File.Contents(path as text, optional options as nullable record) as binary
```

About

Returns the contents of the file, `path`, as binary. The `options` parameter is currently intended for internal use only.

Folder.Contents

06/19/2025

Syntax

```
Folder.Contents(path as text, optional options as nullable record) as table
```

About

Returns a table containing a row for each folder and file found in the folder `path`. Each row contains properties of the folder or file and a link to its content. The `options` parameter is currently intended for internal use only.

Folder.Files

06/19/2025

Syntax

```
Folder.Files(path as text, optional options as nullable record) as table
```

About

Returns a table containing a row for each file found in the folder `path` and all its subfolders. Each row contains properties of the file and a link to its content. The `options` parameter is currently intended for internal use only.

GoogleAnalytics.Accounts

06/19/2025

Syntax

```
GoogleAnalytics.Accounts(optional options as nullable record) as table
```

About

Returns Google Analytics accounts that are accessible from the current credential.

Hdfs.Contents

06/19/2025

Syntax

```
Hdfs.Contents(url as text) as table
```

About

Returns a table containing a row for each folder and file found at the folder URL, `url`, from a Hadoop file system. Each row contains properties of the folder or file and a link to its content.

Hdfs.Files

06/19/2025

Syntax

```
Hdfs.Files(url as text) as table
```

About

Returns a table containing a row for each file found at the folder URL, `url`, and subfolders from a Hadoop file system. Each row contains properties of the file and a link to its content.

HdInsight.Containers

06/19/2025

Syntax

```
HdInsight.Containers(account as text) as table
```

About

Returns a navigational table containing a row for each container found at the account URL, `account`, from an Azure storage vault. Each row contains a link to the container blobs.

HdInsight.Contents

06/19/2025

Syntax

```
HdInsight.Contents(account as text) as table
```

About

Returns a navigational table containing a row for each container found at the account URL, `account`, from an Azure storage vault. Each row contains a link to the container blobs.

HdInsight.Files

06/19/2025

Syntax

```
HdInsight.Files(account as text, containerName as text) as table
```

About

Returns a table containing a row for each blob file found at the container URL, `account`, from an Azure storage vault. Each row contains properties of the file and a link to its content.

Html.Table

06/19/2025

Syntax

```
Html.Table(html as any, columnNameSelectorPairs as list, optional options as nullable record) as table
```

About

Returns a table containing the results of running the specified CSS selectors against the provided `html`. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `RowSelector`

Example 1

Returns a table from a sample html text value.

Usage

Power Query M

```
Html.Table("<div class=\"\"name\"\">Jo</div><span>Manager</span>", {"Name", ".name"}, {"Title", "span"}}, [RowSelector=".name"])\
```

Output

```
#table({"Name", "Title"}, {"Jo", "Manager"})
```

Example 2

Extracts all the hrefs from a sample html text value.

Usage

Power Query M


```
Html.Table("<a href='\"'/test.html\"'>Test</a>", {"Link", "a", each [Attributes]  
[href]})
```

Output

```
#table({"Link"}, {""/test.html"})
```

Identity.From

06/19/2025

Syntax

```
Identity.From(identityProvider as function, value as any) as record
```

About

Creates an identity.

Identity.IsMemberOf

06/19/2025

Syntax

```
Identity.IsMemberOf(identity as record, collection as record) as logical
```

About

Determines whether an identity is a member of an identity collection.

IdentityProvider.Default

06/19/2025

Syntax

```
IdentityProvider.Default() as any
```

About

The default identity provider for the current host.

Informix.Database

06/19/2025

Syntax

```
Informix.Database(server as text, database as text, optional options as nullable  
record) as table
```

About

Returns a table of SQL tables and views available in an Informix database on server `server` in the database instance named `database`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `Query`: A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

Json.Document

06/19/2025

Syntax

```
Json.Document(jsonText as any, optional encoding as nullable number) as any
```

About

Returns the content of the JSON document.

- `jsonText`: The content of the JSON document. The value of this parameter can be text or a binary value returned by a function like [File.Content](#).
- `encoding`: A [TextEncoding.Type](#) that specifies the encoding used in the JSON document. If `encoding` is omitted, UTF8 is used.

Example 1

Returns the content of the specified JSON text as a record.

Usage

Power Query M

```
let
    Source = "{
        \"project\": \"Contosoware\",
        \"description\": \"A comprehensive initiative aimed at enhancing digital presence.\",
        \"components\": [
            \"Website Development\",
            \"CRM Implementation\",
            \"Mobile Application\"
        ]
    }",
    jsonDocument = Json.Document(Source)
in
    jsonDocument
```

Output

Power Query M

```
[
  project = "Contosoware",
  description = "A comprehensive initiative aimed at enhancing digital
presence."
  components =
  {
    "Website Development",
    "CRM Implementation",
    "Mobile Application"
  }
]
```

Example 2

Returns the content of a local JSON file.

Usage

Power Query M

```
let
    Source = (Json.Document(
        File.Contents("C:\test-examples\JSON\Contosoware.json")
    )
in
    Source
```

Output

A record, list, or primitive value representing the JSON data contained in the file

Example 3

Returns the content of an online UTF16 encoded JSON file.

Usage

Power Query M

```
let
    Source = Json.Document(
        Web.Contents("https://contoso.com/products/Contosoware.json"),
        TextEncoding.Utf16
    )
```

Output

A record, list, or primitive value representing the JSON UTF16 data contained in the file

Json.FromValue

06/19/2025

Syntax

```
Json.FromValue(value as any, optional encoding as nullable number) as binary
```

About

Produces a JSON representation of a given value `value` with a text encoding specified by `encoding`. If `encoding` is omitted, UTF8 is used. Values are represented as follows:

- Null, text and logical values are represented as the corresponding JSON types
- Numbers are represented as numbers in JSON, except that `#infinity`, `-#infinity` and `#nan` are converted to null
- Lists are represented as JSON arrays
- Records are represented as JSON objects
- Tables are represented as an array of objects
- Dates, times, datetimes, datetimezones and durations are represented as ISO-8601 text
- Binary values are represented as base-64 encoded text
- Types and functions produce an error

Example 1

Convert a complex value to JSON.

Usage

Power Query M

```
Text.FromBinary(Json.FromValue([A = {1, true, "3"}, B = #date(2012, 3, 25)]))
```

Output

```
"{"A": [1, true, "3"], "B": "2012-03-25"}"
```

MySQL.Database

06/19/2025

Syntax

```
MySQL.Database(server as text, database as text, optional options as nullable  
record) as table
```

About

Returns a table of SQL tables, views, and stored scalar functions available in a MySQL database on server `server` in the database instance named `database`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `Encoding`: A `TextEncoding` value that specifies the character set used to encode all queries sent to the server (default is null).
- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `Query`: A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `TreatTinyAsBoolean`: A logical (true/false) that determines whether to force `tinyint` columns on the server as logical values. The default value is true.
- `OldGuids`: A logical (true/false) that sets whether `char(36)` columns (if false) or `binary(16)` columns (if true) will be treated as GUIDs. The default value is false.
- `ReturnSingleDatabase`: A logical (true/false) that sets whether to return all tables of all databases (if false) or to return tables and views of the specified database (if true). The default value is false.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

OData.Feed

06/19/2025

Syntax

```
OData.Feed(serviceUri as text, optional headers as nullable record, optional options as any) as any
```

About

Returns a table of OData feeds offered by an OData service from a uri `serviceUri`, headers `headers`. A boolean value specifying whether to use concurrent connections or an optional record parameter, `options`, may be specified to control the following options:

- `Query`: Programmatically add query parameters to the URL without having to worry about escaping.
- `Headers`: Specifying this value as a record will supply additional headers to an HTTP request.
- `ExcludedFromCacheKey`: Specifying this value as a list will exclude these HTTP header keys from being part of the calculation for caching data.
- `ApiKeyName`: If the target site has a notion of an API key, this parameter can be used to specify the name (not the value) of the key parameter that must be used in the URL. The actual key value is provided in the credential.
- `Timeout`: Specifying this value as a duration will change the timeout for an HTTP request. The default value is 600 seconds.
- `EnableBatch`: A logical (true/false) that sets whether to allow generation of an OData \$batch request if the MaxUriLength is exceeded (default is false).
- `MaxUriLength`: A number that indicates the max length of an allowed uri sent to an OData service. If exceeded and EnableBatch is true then the request will be made to an OData \$batch endpoint, otherwise it will fail (default is 2048).
- `Concurrent`: A logical (true/false) when set to true, requests to the service will be made concurrently. When set to false, requests will be made sequentially. When not specified, the value will be determined by the service's AsynchronousRequestsSupported annotation. If the service does not specify whether AsynchronousRequestsSupported is supported, requests will be made sequentially.
- `ODataVersion`: A number (3 or 4) that specifies the OData protocol version to use for this OData service. When not specified, all supported versions will be requested. The service

version will be determined by the OData-Version header returned by the service.

- **FunctionOverloads**: A logical (true/false) when set to true, function import overloads will be listed in the navigator as separate entries, when set to false, function import overloads will be listed as one union function in the navigator. Default value for V3: false. Default value for V4: true.
- **MoreColumns**: A logical (true/false) when set to true, adds a "More Columns" column to each entity feed containing open types and polymorphic types. This will contain the fields not declared in the base type. When false, this field is not present. Defaults to false.
- **IncludeAnnotations**: A comma separated list of namespace qualified term names or patterns to include with "*" as a wildcard. By default, none of the annotations are included.
- **IncludeMetadataAnnotations**: A comma separated list of namespace qualified term names or patterns to include on metadata document requests, with "*" as a wildcard. By default, includes the same annotations as IncludeAnnotations.
- **OmitValues**: Allows the OData service to avoid writing out certain values in responses. If acknowledged by the service, we will infer those values from the omitted fields. Options include:
 - **ODataOmitValues.Nulls**: Allows the OData service to omit null values.
- **Implementation**: Specifies the implementation of the OData connector to use. Valid values are "2.0" or null.

Example 1

Connect to the TripPin OData service.

Usage

Power Query M

```
OData.Feed("https://services.odata.org/V4/TripPinService")
```

Output

table

Odbc.DataSource

06/19/2025

Syntax

```
Odbc.DataSource(connectionString as any, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views from the ODBC data source specified by the connection string `connectionString`. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is 15 seconds.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth`: A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.

Example 1

Return the SQL tables and views from the provided connection string.

Usage

Power Query M

```
Odbc.DataSource("dsn=your_dsn")
```

Output

Power Query M
table

Odbc.InferOptions

06/19/2025

Syntax

```
Odbc.InferOptions(connectionString as any) as record
```

About

Returns the result of trying to infer SQL capabilities with the connection string `connectionString` using ODBC. `connectionString` can be text or a record of property value pairs. Property values can either be text or number.

Example 1

Return the inferred SQL capabilities for a connection string.

Usage

Power Query M

```
Odbc.InferOptions("dsn=your_dsn")
```

Output

Power Query M

```
record
```


Odbc.Query

06/19/2025

Syntax

```
Odbc.Query(connectionString as any, query as text, optional options as nullable record) as table
```

About

Returns the result of running `query` with the connection string `connectionString` using ODBC. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is 15 seconds.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth`: A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.

Example 1

Return the result of running a simple query against the provided connection string.

Usage

```
Power Query M
```

```
Odbc.Query("dsn=your_dsn", "select * from Customers")
```

Output

```
Power Query M
```

```
table
```

OleDb.DataSource

06/19/2025

Syntax

```
OleDb.DataSource(connectionString as any, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views from the OLE DB data source specified by the connection string `connectionString`. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `Query`: A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is true).
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth`: A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

OleDb.Query

06/19/2025

Syntax

```
OleDb.Query(connectionString as any, query as text, optional options as nullable record) as table
```

About

Returns the result of running `query` with the connection string `connectionString` using OLE DB. `connectionString` can be text or a record of property value pairs. Property values can either be text or number. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `SqlCompatibleWindowsAuth`: A logical (true/false) that determines whether to produce SQL Server-compatible connection string options for Windows authentication. The default value is true.

Oracle.Database

06/19/2025

Syntax

```
Oracle.Database(server as text, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views from the Oracle database on server `server`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `Query`: A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

Pdf.Tables

06/19/2025

Syntax

```
Pdf.Tables(pdf as binary, optional options as nullable record) as table
```

About

Returns any tables found in `pdf`. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `Implementation`: The version of the algorithm to use when identifying tables. Old versions are available only for backwards compatibility, to prevent old queries from being broken by algorithm updates. The newest version should always give the best results. Valid values are "1.3", "1.2", "1.1", or null.
- `StartPage`: Specifies the first page in the range of pages to examine. Default: 1.
- `EndPage`: Specifies the last page in the range of pages to examine. Default: the last page of the document.
- `MultiPageTables`: Controls whether similar tables on consecutive pages will be automatically combined into a single table. Default: true.
- `EnforceBorderLines`: Controls whether border lines are always enforced as cell boundaries (when true), or simply used as one hint among many for determining cell boundaries (when false). Default: false.

Example 1

Returns the tables contained in `sample.pdf`.

Usage

Power Query M

```
Pdf.Tables(File.Contents("c:\sample.pdf"))
```

Output

Power Query M

```
#table({"Name", "Kind", "Data"}, ...)
```

PostgreSQL.Database

06/19/2025

Syntax

```
PostgreSQL.Database(server as text, database as text, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views available in a PostgreSQL database on server `server` in the database instance named `database`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `Query`: A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

RData.FromBinary

06/19/2025

Syntax

```
RData.FromBinary(stream as binary) as any
```

About

Returns a record of data frames from the RData file.

Salesforce.Data

06/19/2025

Syntax

```
Salesforce.Data(optional loginUrl as any, optional options as nullable record) as table
```

About

Returns the objects on the Salesforce account provided in the credentials. The account will be connected through the provided environment `loginUrl`. If no environment is provided then the account will connect to production (<https://login.salesforce.com> [↗](#)). An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is false).
- `ApiVersion`: The Salesforce API version to use for this query. When not specified, API version 29.0 is used.
- `Timeout`: A duration that controls how long to wait before abandoning the request to the server. The default value is source-specific.

Salesforce.Reports

06/19/2025

Syntax

```
Salesforce.Reports(optional loginUrl as nullable text, optional options as nullable record) as table
```

About

Returns the reports on the Salesforce account provided in the credentials. The account will be connected through the provided environment `loginUrl`. If no environment is provided then the account will connect to production (<https://login.salesforce.com> [↗](#)). An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `ApiVersion`: The Salesforce API version to use for this query. When not specified, API version 29.0 is used.
- `Timeout`: A duration that controls how long to wait before abandoning the request to the server. The default value is source-specific.

SapBusinessWarehouse.Cubes

06/19/2025

Syntax

```
SapBusinessWarehouse.Cubes(server as text, systemNumberOrSystemId as text,  
clientId as text, optional optionsOrLogonGroup as any, optional options as  
nullable record) as table
```

About

Returns a table of InfoCubes and queries grouped by InfoArea from an SAP Business Warehouse instance at server `server` with system number `systemNumberOrSystemId` and Client ID `clientId`. An optional record parameter, `optionsOrLogonGroup`, may be specified to control options.

SapHana.Database

06/19/2025

Syntax

```
SapHana.Database(server as text, optional options as nullable record) as table
```

About

Returns a table of multidimensional packages from the SAP HANA database `server`. An optional record parameter, `options`, may be specified to control the following options:

- `Query`: A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `Distribution`: A [SapHanaDistribution](#) that sets the value of the "Distribution" property in the connection string. Statement routing is the method of evaluating the correct server node of a distributed system before statement execution. The default value is [SapHanaDistribution.All](#).
- `Implementation`: Specifies the implementation of the SAP HANA connector to use.
- `EnableColumnBinding`: Binds variables to the columns of a SAP HANA result set when fetching data. May potentially improve performance at the cost of slightly higher memory utilization. The default value is false.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is 15 seconds.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.

SharePoint.Contents

06/19/2025

Syntax

```
SharePoint.Contents(url as text, optional options as nullable record) as table
```

About

Returns a table containing a row for each folder and document found at the specified SharePoint site, **url**. Each row contains properties of the folder or file and a link to its content.

options may be specified to control the following options:

- **ApiVersion**: A number (14 or 15) or the text "Auto" that specifies the SharePoint API version to use for this site. When not specified, API version 14 is used. When Auto is specified, the server version will be automatically discovered if possible, otherwise version defaults to 14. Non-English SharePoint sites require at least version 15.
- **Implementation**: Optional. Specifies which version of the SharePoint connector to use. Accepted values are **2.0** or **null**. If the value is **2.0**, the 2.0 implementation of the SharePoint connector is used. If the value is **null**, the original implementation of the SharePoint connector is used.

SharePoint.Files

06/19/2025

Syntax

```
SharePoint.Files(url as text, optional options as nullable record) as table
```

About

Returns a table containing a row for each document found at the specified SharePoint site, **url**, and subfolders. Each row contains properties of the folder or file and a link to its content.

options may be specified to control the following options:

- **ApiVersion**: A number (14 or 15) or the text "Auto" that specifies the SharePoint API version to use for this site. When not specified, API version 14 is used. When Auto is specified, the server version will be automatically discovered if possible, otherwise version defaults to 14. Non-English SharePoint sites require at least version 15.

SharePoint.Tables

06/19/2025

Syntax

```
SharePoint.Tables(url as text, optional options as nullable record) as table
```

About

Returns a table containing a row for each List item found at the specified SharePoint list, **url**. Each row contains properties of the List. **options** may be specified to control the following options:

- **ApiVersion**: A number (14 or 15) or the text "Auto" that specifies the SharePoint API version to use for this site. When not specified, API version 14 is used. When Auto is specified, the server version will be automatically discovered if possible, otherwise version defaults to 14. Non-English SharePoint sites require at least version 15.
- **Implementation**: Optional. Specifies which version of the SharePoint connector to use. Accepted values are "2.0" or null. If the value is "2.0", the 2.0 implementation of the SharePoint connector is used. If the value is null, the original implementation of the SharePoint connector is used.
- **ViewMode**: Optional. This option is only valid for implementation 2.0. Accepted values are "All" and "Default". If no value is specified, the value is set to "All". When "All" is specified, the view includes all user-created and system-defined columns. When "Default" is specified, the view will match what the user sees when looking at the list online in whichever view that user set as Default in their settings. If the user edits their default view to add or remove either user-created or system-defined columns, or by creating a new view and setting it as default, these changes will propagate through the connector.
- **DisableAppendNoteColumns**: Prevents the connector from using a separate endpoint for note columns.

Soda.Feed

06/19/2025

Syntax

```
Soda.Feed(url as text) as table
```

About

Returns a table from the contents at the specified URL **url** formatted according to the SODA 2.0 API. The URL must point to a valid SODA-compliant source that ends in a .csv extension.

Sql.Database

06/19/2025

Syntax

```
Sql.Database(server as text, database as text, optional options as nullable  
record) as table
```

About

Returns a table of SQL tables, views, and stored functions from the SQL Server database `database` on server `server`. The port may be optionally specified with the server, separated by a colon or a comma. An optional record parameter, `options`, may be specified to control the following options:

- `Query`: A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `MaxDegreeOfParallelism`: A number that sets the value of the "maxdop" query clause in the generated SQL query.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).
- `MultiSubnetFailover`: A logical (true/false) that sets the value of the "MultiSubnetFailover" property in the connection string (default is false).
- `UnsafeTypeConversions`: A logical (true/false) that, if true, attempts to fold type conversions which could fail and cause the entire query to fail. Not recommended for general use.
- `ContextInfo`: A binary value that is used to set the CONTEXT_INFO before running each command.

- `OmitSRID`: A logical (true/false) that, if true, omits the SRID when producing Well-Known Text from geometry and geography types.
- `EnableCrossDatabaseFolding`: A logical (true/false) value that, if true, allows query folding across databases on the same server. The default value is false.

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

Sql.Databases

06/19/2025

Syntax

```
Sql.Databases(server as text, optional options as nullable record) as table
```

About

Returns a table of databases on the specified SQL server, `server`. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `MaxDegreeOfParallelism`: A number that sets the value of the "maxdop" query clause in the generated SQL query.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).
- `MultiSubnetFailover`: A logical (true/false) that sets the value of the "MultiSubnetFailover" property in the connection string (default is false).
- `UnsafeTypeConversions`: A logical (true/false) that, if true, attempts to fold type conversions which could fail and cause the entire query to fail. Not recommended for general use.
- `ContextInfo`: A binary value that is used to set the CONTEXT_INFO before running each command.
- `OmitSRID`: A logical (true/false) that, if true, omits the SRID when producing Well-Known Text from geometry and geography types.
- `EnableCrossDatabaseFolding`: A logical (true/false) value that, if true, allows query folding across databases on the same server. The default value is false.

The record parameter is specified as [option1 = value1, option2 = value2...] for example.

Does not support setting a SQL query to run on the server. [Sql.Database](#) should be used instead to run a SQL query.

Sybase.Database

06/19/2025

Syntax

```
Sybase.Database(server as text, database as text, optional options as nullable  
record) as table
```

About

Returns a table of SQL tables and views available in a Sybase database on server `server` in the database instance named `database`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `Query`: A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

Teradata.Database

06/19/2025

Syntax

```
Teradata.Database(server as text, optional options as nullable record) as table
```

About

Returns a table of SQL tables and views from the Teradata database on server `server`. The port may be optionally specified with the server, separated by a colon. An optional record parameter, `options`, may be specified to control the following options:

- `CreateNavigationProperties`: A logical (true/false) that sets whether to generate navigation properties on the returned values (default is true).
- `NavigationPropertyNameGenerator`: A function that is used for the creation of names for navigation properties.
- `Query`: A native SQL query used to retrieve data. If the query produces multiple result sets, only the first will be returned.
- `CommandTimeout`: A duration that controls how long the server-side query is allowed to run before it is canceled. The default value is ten minutes.
- `ConnectionTimeout`: A duration that controls how long to wait before abandoning an attempt to make a connection to the server. The default value is driver-dependent.
- `HierarchicalNavigation`: A logical (true/false) that sets whether to view the tables grouped by their schema names (default is false).

The record parameter is specified as [option1 = value1, option2 = value2...] or [Query = "select ..."] for example.

WebAction.Request

06/19/2025

Syntax

```
WebAction.Request(method as text, url as text, optional options as nullable  
record) as action
```

About

Creates an action that, when executed, will return the results of performing a `method` request against `url` using HTTP as a binary value. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `Query`: Programmatically add query parameters to the URL without having to worry about escaping.
- `ApiKeyName`: If the target site has a notion of an API key, this parameter can be used to specify the name (not the value) of the key parameter that must be used in the URL. The actual key value is provided in the credential.
- `Headers`: Specifying this value as a record will supply additional headers to an HTTP request.
- `Timeout`: Specifying this value as a duration will change the timeout for an HTTP request. The default value is 100 seconds.
- `ExcludedFromCacheKey`: Specifying this value as a list will exclude these HTTP header keys from being part of the calculation for caching data.
- `IsRetry`: Specifying this logical value as true will ignore any existing response in the cache when fetching data.
- `ManualStatusHandling`: Specifying this value as a list will prevent any builtin handling for HTTP requests whose response has one of these status codes.
- `RelativePath`: Specifying this value as text appends it to the base URL before making the request.
- `Content`: Specifying this value will cause its contents to become the body of the HTTP request.

Note that this function is disabled in most contexts. Consider using [Web.Contents](#) or [Web.Headers](#) instead.

Example 1

Perform a GET request against Bing.

Usage

Power Query M

```
WebAction.Request(WebMethod.Get, "https://bing.com")
```

Output

Action

Web.BrowserContents

06/19/2025

Syntax

```
Web.BrowserContents(url as text, optional options as nullable record) as text
```

About

Returns the HTML for the specified `url`, as viewed by a web browser. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `ApiKeyName`: If the target site has a notion of an API key, this parameter can be used to specify the name (not the value) of the key parameter that must be used in the URL. The actual key value is provided in the credential.
- `WaitFor`: Specifies a condition to wait for before downloading the HTML, in addition to waiting for the page to load (which is always done). Can be a record containing Timeout and/or Selector fields. If only a Timeout is specified, the function will wait the amount of time specified before downloading the HTML. If both a Selector and Timeout are specified, and the Timeout elapses before the Selector exists on the page, an error will be thrown. If a Selector is specified with no Timeout, a default Timeout of 30 seconds is applied.

Example 1

Returns the HTML for <https://microsoft.com>.

Usage

Power Query M

```
Web.BrowserContents("https://microsoft.com")
```

Output

```
"<!DOCTYPE html><html xmlns=..."
```

Example 2

Returns the HTML for <https://microsoft.com> after waiting for a CSS selector to exist.

Usage

Power Query M

```
Web.BrowserContents("https://microsoft.com", [WaitFor = [Selector = "div.ready"]])
```

Output

```
"<!DOCTYPE html><html xmlns=..."
```

Example 3

Returns the HTML for <https://microsoft.com> after waiting ten seconds.

Usage

Power Query M

```
Web.BrowserContents("https://microsoft.com", [WaitFor = [Timeout =  
#duration(0,0,0,10)]])
```

Output

```
"<!DOCTYPE html><html xmlns=..."
```

Example 4

Returns the HTML for <https://microsoft.com> after waiting up to ten seconds for a CSS selector to exist.

Usage

Power Query M

```
Web.BrowserContents("https://microsoft.com", [WaitFor = [Selector = "div.ready",  
Timeout = #duration(0,0,0,10)]])
```

Output

```
"<!DOCTYPE html><html xmlns=..."
```

Web.Contents

06/19/2025

Syntax

```
Web.Contents(url as text, optional options as nullable record) as binary
```

About

Returns the contents downloaded from `url` as binary. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `Query`: Programmatically add query parameters to the URL without having to worry about escaping.
- `ApiKeyName`: If the target site has a notion of an API key, this parameter can be used to specify the name (not the value) of the key parameter that must be used in the URL. The actual key value is provided in the credential.
- `Headers`: Specifying this value as a record will supply additional headers to an HTTP request.
- `Timeout`: Specifying this value as a duration will change the timeout for an HTTP request. The default value is 100 seconds.
- `ExcludedFromCacheKey`: Specifying this value as a list will exclude these HTTP header keys from being part of the calculation for caching data.
- `IsRetry`: Specifying this logical value as true will ignore any existing response in the cache when fetching data.
- `ManualStatusHandling`: Specifying this value as a list will prevent any builtin handling for HTTP requests whose response has one of these status codes.
- `RelativePath`: Specifying this value as text appends it to the base URL before making the request.
- `Content`: Specifying this value changes the web request from a GET to a POST, using the value of the option as the content of the POST.

The HTTP request is made as either a GET (when no `Content` is specified) or a POST (when there is `Content`). POST requests may only be made anonymously.

The headers of the HTTP response are available as metadata on the binary result. Outside of a custom data connector context, only a subset of response headers is available (for security

reasons).

Example 1

Retrieve the contents of `"https://bing.com/search?q=Power+Query"` using the `RelativePath` and `Query` options. These options can be used to dynamically query a static base URL.

Usage

Power Query M

```
let
    searchText = "Power Query"
in
    Web.Contents(
        "https://www.bing.com",
        [
            RelativePath = "search",
            Query = [q = searchText]
        ]
    )
```

Output

binary

Example 2

Perform a POST against a URL, passing a binary JSON payload and parsing the response as JSON.

Usage

Power Query M

```
let
    url = ...,
    headers = [{"Content-Type" = "application/json"}],
    postData = Json.FromValue([x = 235.7, y = 41.53]),
    response = Web.Contents(
        url,
        [
            Headers = headers,
            Content = postData
        ]
    ),
    jsonResponse = Json.Document(response)
```

```
in  
    jsonResponse
```

Output

table

Example 3

Connect to a secure URL that accepts an authentication key as part of its query string. Instead of hard-coding the secret key in M (which would pose a security risk), the key can be provided securely by specifying its name (not its value) in M, choosing Web API authentication, and entering the key value as part of the Web API credential. When used in this way, the following example will generate a request to `"https://contoso.com/api/customers/get?api_key=*****"`.

Usage

Power Query M

```
Web.Contents("https://contoso.com/api/customers/get", [ApiKeyName="api_key"])
```

Output

binary

More information

[Status code handling with Web.Contents in custom connectors](#)

Web.Headers

06/19/2025

Syntax

```
Web.Headers(url as text, optional options as nullable record) as record
```

About

Returns the headers downloaded from `url` as a record. An optional record parameter, `options`, may be provided to specify additional properties. The record can contain the following fields:

- `Query`: Programmatically add query parameters to the URL without having to worry about escaping.
- `ApiKeyName`: If the target site has a notion of an API key, this parameter can be used to specify the name (not the value) of the key parameter that must be used in the URL. The actual key value is provided in the credential.
- `Headers`: Specifying this value as a record will supply additional headers to an HTTP request.
- `Timeout`: Specifying this value as a duration will change the timeout for an HTTP request. The default value is 100 seconds.
- `ExcludedFromCacheKey`: Specifying this value as a list will exclude these HTTP header keys from being part of the calculation for caching data.
- `IsRetry`: Specifying this logical value as true will ignore any existing response in the cache when fetching data.
- `ManualStatusHandling`: Specifying this value as a list will prevent any builtin handling for HTTP requests whose response has one of these status codes.
- `RelativePath`: Specifying this value as text appends it to the base URL before making the request.

The HTTP request is made with the HEAD method. Outside of a custom data connector context, only a subset of response headers is available (for security reasons).

Example 1

Retrieve the HTTP headers for `"https://bing.com/search?q=Power+Query"` using the `RelativePath` and `Query` options.

Usage

Power Query M

```
let
    searchText = "Power Query"
in
    Web.Headers(
        "https://www.bing.com",
        [
            RelativePath = "search",
            Query = [q = searchText]
        ]
    )
```

Output

Power Query M

```
([
    #"Cache-Control" = "private, max-age=0",
    #"Content-Encoding" = "gzip",
    #"Content-Length" = "0",
    #"Content-Type" = "text/html; charset=utf-8",
    Date = "Tue, 14 Dec 2021 16:57:25 GMT",
    Expires = "Tue, 14 Dec 2021 16:56:25 GMT",
    Vary = "Accept-Encoding"
]
meta [
    Response.Status = 200
])
```


Web.Page

06/19/2025

Syntax

```
Web.Page(html as any) as table
```

About

Returns the contents of the HTML document broken into its constituent structures, as well as a representation of the full document and its text after removing tags.

Xml.Document

06/19/2025

Syntax

```
Xml.Document(contents as any, optional encoding as nullable number) as table
```

About

Returns the contents of the XML document as a hierarchical table.

Xml.Tables

06/19/2025

Syntax

```
Xml.Tables(contents as any, optional options as nullable record, optional encoding as nullable number) as table
```

About

Returns the contents of the XML document as a nested collection of flattened tables.

Example 1

Retrieve the contents of a local XML file.

Usage

Power Query M

```
Xml.Tables(File.Contents("C:\invoices.xml"))
```

Output

table

Binary functions

Article • 02/21/2023

These functions create and manipulate binary data.

Binary Formats

Reading numbers

Name	Description
BinaryFormat.7BitEncodedSignedInteger	A binary format that reads a 64-bit signed integer that was encoded using a 7-bit variable-length encoding.
BinaryFormat.7BitEncodedUnsignedInteger	A binary format that reads a 64-bit unsigned integer that was encoded using a 7-bit variable-length encoding.
BinaryFormat.Binary	Returns a binary format that reads a binary value.
BinaryFormat.Byte	A binary format that reads an 8-bit unsigned integer.
BinaryFormat.Choice	Returns a binary format that chooses the next binary format based on a value that has already been read.
BinaryFormat.Decimal	A binary format that reads a .NET 16-byte decimal value.
BinaryFormat.Double	A binary format that reads an 8-byte IEEE double-precision floating point value.
BinaryFormat.Group	Returns a binary format that reads a group of items. Each item value is preceded by a unique key value. The result is a list of item values.
BinaryFormat.Length	Returns a binary format that limits the amount of data that can be read. Both BinaryFormat.List and BinaryFormat.Binary can be used to read until end of the data. BinaryFormat.Length can be used to limit the number of bytes that are read.
BinaryFormat.List	Returns a binary format that reads a sequence of items and returns a list.

Name	Description
BinaryFormat.Null	A binary format that reads zero bytes and returns null.
BinaryFormat.Record	Returns a binary format that reads a record. Each field in the record can have a different binary format.
BinaryFormat.SignedInteger16	A binary format that reads a 16-bit signed integer.
BinaryFormat.SignedInteger32	A binary format that reads a 32-bit signed integer.
BinaryFormat.SignedInteger64	A binary format that reads a 64-bit signed integer.
BinaryFormat.Single	A binary format that reads a 4-byte IEEE single-precision floating point value.
BinaryFormat.Text	Returns a binary format that reads a text value. The optional encoding value specifies the encoding of the text.
BinaryFormat.Transform	Returns a binary format that will transform the values read by another binary format.
BinaryFormat.UnsignedInteger16	A binary format that reads a 16-bit unsigned integer.
BinaryFormat.UnsignedInteger32	A binary format that reads a 32-bit unsigned integer.
BinaryFormat.UnsignedInteger64	A binary format that reads a 64-bit unsigned integer.

Controlling byte order

Name	Description
BinaryFormat.ByteOrder	Returns a binary format with the byte order specified by a function.
Table.PartitionValues	Returns information about how a table is partitioned.

Binary data

Name	Description
Binary.ApproximateLength	Returns the approximate length of the binary.

Name	Description
Binary.Buffer	Buffers the binary value in memory. The result of this call is a stable binary value, which means it will have a deterministic length and order of bytes.
Binary.Combine	Combines a list of binaries into a single binary.
Binary.Compress	Compresses a binary value using the given compression type.
Binary.Decompress	Decompresses a binary value using the given compression type.
Binary.From	Returns a binary value from the given value.
Binary.FromList	Converts a list of numbers into a binary value
Binary.FromText	Decodes data from a text form into binary.
Binary.InferContentType	Returns a record with field Content.Type that contains the inferred MIME-type.
Binary.Length	Returns the length of binary values.
Binary.Range	Returns a subset of the binary value beginning at an offset.
Binary.Split	Splits the specified binary into a list of binaries using the specified page size.
Binary.ToList	Converts a binary value into a list of numbers
Binary.ToText	Encodes binary data into a text form.
Binary.View	Creates or extends a binary with user-defined handlers for query and action operations.
Binary.ViewError	Creates a modified error record which won't trigger a fallback when thrown by a handler defined on a view (via Binary.View).
Binary.ViewFunction	Creates a function that can be intercepted by a handler defined on a view (via <code>Binary.View</code>).
#binary	Creates a binary value from numbers or text.

Feedback

Was this page helpful?



Get help at [Microsoft Q&A](#)

Binary.ApproximateLength

06/19/2025

Syntax

```
Binary.ApproximateLength(binary as nullable binary) as nullable number
```

About

Returns the approximate length of `binary`, or an error if the data source doesn't support an approximate length.

Example 1

Get the approximate length of the binary value.

Usage

Power Query M

```
Binary.ApproximateLength(Binary.FromText("i45WMlSKjQUA", BinaryEncoding.Base64))
```

Output

9

Binary.Buffer

06/19/2025

Syntax

```
Binary.Buffer(binary as nullable binary) as nullable binary
```

About

Buffers the binary value in memory. The result of this call is a stable binary value, which means it will have a deterministic length and order of bytes.

Example 1

Create a stable version of the binary value.

Usage

```
Power Query M
```

```
Binary.Buffer(Binary.FromList({0..10}))
```

Output

```
#binary({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10})
```


Binary.Combine

06/19/2025

Syntax

```
Binary.Combine(binaries as list) as binary
```

About

Combines a list of binaries into a single binary.

Binary.Compress

06/19/2025

Syntax

```
Binary.Compress(binary as nullable binary, compressionType as number) as nullable binary
```

About

Compresses a binary value using the given compression type. The result of this call is a compressed copy of the input. Compression types include:

- [Compression.GZip](#)
- [Compression.Deflate](#)

Example 1

Compress the binary value.

Usage

Power Query M

```
Binary.Compress(Binary.FromList(List.Repeat({10}, 1000)), Compression.Deflate)
```

Output

```
#binary({227, 226, 26, 5, 163, 96, 20, 12, 119, 0, 0})
```

Binary.Decompress

06/19/2025

Syntax

```
Binary.Decompress(binary as nullable binary, compressionType as number) as  
nullable binary
```

About

Decompresses a binary value using the given compression type. The result of this call is a decompressed copy of the input. Compression types include:

- [Compression.GZip](#)
- [Compression.Deflate](#)

Example 1

Decompress the binary value.

Usage

Power Query M

```
Binary.Decompress(#binary({115, 103, 200, 7, 194, 20, 134, 36, 134, 74, 134, 84,  
6, 0}), Compression.Deflate)
```

Output

```
#binary({71, 0, 111, 0, 111, 0, 100, 0, 98, 0, 121, 0, 101, 0})
```

Binary.From

06/19/2025

Syntax

```
Binary.From(value as any, optional encoding as nullable number) as nullable binary
```

About

Returns a `binary` value from the given `value`. If the given `value` is `null`, **Binary.From** returns `null`. If the given `value` is `binary`, `value` is returned. Values of the following types can be converted to a `binary` value:

- `text`: A `binary` value from the text representation. Refer to [Binary.FromText](#) for details.

If `value` is of any other type, an error is returned.

Example 1

Get the `binary` value of `"1011"`.

Usage

Power Query M

```
Binary.From("1011")
```

Output

```
Binary.FromText("1011", BinaryEncoding.Base64)
```

Binary.FromList

06/19/2025

Syntax

```
Binary.FromList(list as list) as binary
```

About

Converts a list of numbers into a binary value.

Binary.FromText

06/19/2025

Syntax

```
Binary.FromText(text as nullable text, optional encoding as nullable number) as nullable binary
```

About

Returns the result of converting text value `text` to a binary (list of `number`). `encoding` may be specified to indicate the encoding used in the text value. The following `BinaryEncoding` values may be used for `encoding`.

- [BinaryEncoding.Base64](#): Base 64 encoding
- [BinaryEncoding.Hex](#): Hex encoding

Example 1

Decode "1011" into binary.

Usage

Power Query M

```
Binary.FromText("1011")
```

Output

Power Query M

```
Binary.FromText("1011", BinaryEncoding.Base64)
```

Example 2

Decode "1011" into binary with Hex encoding.

Usage

Power Query M

```
Binary.FromText("1011", BinaryEncoding.Hex)
```

Output

Power Query M

```
Binary.FromText("EBE=", BinaryEncoding.Base64)
```

Binary.InferContentType

06/19/2025

Syntax

```
Binary.InferContentType(source as binary) as record
```

About

Returns a record with field `Content.Type` that contains the inferred MIME-type. If the inferred content type is `text/*`, and an encoding code page is detected, then additionally returns field `Content.Encoding` that contains the encoding of the stream. If the inferred content type is `text/csv`, and the format is delimited, additionally returns field `Csv.PotentialDelimiter` containing a table for analysis of potential delimiters. If the inferred content type is `text/csv`, and the format is fixed-width, additionally returns field `Csv.PotentialPositions` containing a list for analysis of potential fixed width column positions.

Binary.Length

06/19/2025

Syntax

```
Binary.Length(binary as nullable binary) as nullable number
```

About

Returns the number of characters.

Binary.Range

06/19/2025

Syntax

```
Binary.Range(binary as binary, offset as number, optional count as nullable number) as binary
```

About

Returns a subset of the binary value beginning at the offset `binary`. An optional parameter, `offset`, sets the maximum length of the subset.

Example 1

Returns a subset of the binary value starting at offset 6.

Usage

Power Query M

```
Binary.Range(#binary({0..10}), 6)
```

Output

```
#binary({6, 7, 8, 9, 10})
```

Example 2

Returns a subset of length 2 from offset 6 of the binary value.

Usage

Power Query M

```
Binary.Range(#binary({0..10}), 6, 2)
```

Output

```
#binary({6, 7})
```

Binary.ToList

06/19/2025

Syntax

```
Binary.ToList(binary as binary) as list
```

About

Converts a binary value into a list of numbers.

Binary.ToText

06/19/2025

Syntax

```
Binary.ToText(binary as nullable binary, optional encoding as nullable number) as  
nullable text
```

About

Returns the result of converting a binary list of numbers `binary` into a text value. Optionally, `encoding` may be specified to indicate the encoding to be used in the text value produced. The following `BinaryEncoding` values may be used for `encoding`.

- `BinaryEncoding.Base64`: Base 64 encoding
- `BinaryEncoding.Hex`: Hex encoding

Binary.View

06/19/2025

Syntax

```
Binary.View(binary as nullable binary, handlers as record) as binary
```

About

Returns a view of `binary` where the functions specified in `handlers` are used in lieu of the default behavior of an operation when the operation is applied to the view.

If `binary` is provided, all handler functions are optional. If `binary` isn't provided, the `GetStream` handler function is required. If a handler function isn't specified for an operation, the default behavior of the operation is applied to `binary` instead (except in the case of `GetExpression`).

Handler functions must return a value that is semantically equivalent to the result of applying the operation against `binary` (or the resulting view in the case of `GetExpression`).

If a handler function raises an error, the default behavior of the operation is applied to the view.

Binary.View can be used to implement folding to a data source—the translation of M queries into source-specific operations (for example, to download a section of a file).

Refer to the published Power Query custom connector documentation for a more complete description of **Binary.View**.

Example 1

Create a basic view that doesn't require accessing the data in order to determine the length.

Usage

Power Query M

```
Binary.View(  
    null,  
    [  
        GetLength = () => 12,  
        GetStream = () => Text.ToBinary("hello world!")  
    ]  
)
```

```
]
)
```

Output

Power Query M

```
Text.ToBinary("hello world!")
```

Binary.ViewError

06/19/2025

Syntax

```
Binary.ViewError(errorRecord as record) as record
```

About

Creates a modified error record from `errorRecord` which won't trigger a fallback when thrown by a handler defined on a view (via [Binary.View](#)).

Binary.ViewFunction

06/19/2025

Syntax

```
Binary.ViewFunction(function as function) as function
```

About

Creates a view function based on `function` that can be handled in a view created by [Binary.View](#).

The `OnInvoke` handler of **Binary.View** can be used to define a handler for the view function.

As with the handlers for built-in operations, if no `OnInvoke` handler is specified, or if it does not handle the view function, or if an error is raised by the handler, `function` is applied on top of the view.

Refer to the published Power Query custom connector documentation for a more complete description of **Binary.View** and custom view functions.

BinaryFormat.7BitEncodedSignedInteger

06/19/2025

Syntax

```
BinaryFormat.7BitEncodedSignedInteger(binary as binary) as any
```

About

A binary format that reads a 64-bit signed integer that was encoded using a 7-bit variable-length encoding.

BinaryFormat.7BitEncodedUnsignedInteger

06/19/2025

Syntax

```
BinaryFormat.7BitEncodedUnsignedInteger(binary as binary) as any
```

About

A binary format that reads a 64-bit unsigned integer that was encoded using a 7-bit variable-length encoding.

BinaryFormat.Binary

06/19/2025

Syntax

```
BinaryFormat.Binary(optional length as any) as function
```

About

Returns a binary format that reads a binary value. If **length** is specified, the binary value will contain that many bytes. If **length** is not specified, the binary value will contain the remaining bytes. The **length** can be specified either as a number, or as a binary format of the length that precedes the binary data.

BinaryFormat.Byte

06/19/2025

Syntax

```
BinaryFormat.Byte(binary as binary) as any
```

About

A binary format that reads an 8-bit unsigned integer.

BinaryFormat.ByteOrder

06/19/2025

Syntax

```
BinaryFormat.ByteOrder(binaryFormat as function, byteOrder as number) as function
```

About

Returns a binary format with the byte order specified by `binaryFormat`. The default byte order is `ByteOrder.BigEndian`.

BinaryFormat.Choice

06/19/2025

Syntax

```
BinaryFormat.Choice(binaryFormat as function, chooseFunction as function, optional type as nullable type, optional combineFunction as nullable function) as function
```

About

Returns a binary format that chooses the next binary format based on a value that has already been read. The binary format value produced by this function works in stages:

- The binary format specified by the `binaryFormat` parameter is used to read a value.
- The value is passed to the choice function specified by the `chooseFunction` parameter.
- The choice function inspects the value and returns a second binary format.
- The second binary format is used to read a second value.
- If the combine function is specified, then the first and second values are passed to the combine function, and the resulting value is returned.
- If the combine function is not specified, the second value is returned.
- The second value is returned.

The optional `type` parameter indicates the type of binary format that will be returned by the choice function. Either `type any`, `type list`, or `type binary` may be specified. If the `type` parameter is not specified, then `type any` is used. If `type list` or `type binary` is used, then the system may be able to return a streaming `binary` or `list` value instead of a buffered one, which may reduce the amount of memory necessary to read the format.

Example 1

Read a list of bytes where the number of elements is determined by the first byte.

Usage

Power Query M

```
let
    binaryData = #binary({2, 3, 4, 5}),
    listFormat = BinaryFormat.Choice(
        BinaryFormat.Byte,
```

```

        (length) => BinaryFormat.List(BinaryFormat.Byte, length)
    )
in
    listFormat(binaryData)

```

Output

```
{3,4}
```

Example 2

Read a list of bytes where the number of elements is determined by the first byte, and preserve the first byte read.

Usage

Power Query M

```

let
    binaryData = #binary({2, 3, 4, 5}),
    listFormat = BinaryFormat.Choice(
        BinaryFormat.Byte,
        (length) => BinaryFormat.Record([
            length = length,
            list = BinaryFormat.List(BinaryFormat.Byte, length)
        ])
    )
in
    listFormat(binaryData)

```

Output

```
[length = 2, list = {3, 4}]
```

Example 3

Read a list of bytes where the number of elements is determined by the first byte using a streaming list.

Usage

Power Query M

```

let
    binaryData = #binary({2, 3, 4, 5}),
    listFormat = BinaryFormat.Choice(

```



```
        BinaryFormat.Byte,  
        (length) => BinaryFormat.List(BinaryFormat.Byte, length),  
        type list  
    )  
in  
    listFormat(binaryData)
```

Output

```
{3, 4}
```

BinaryFormat.Decimal

06/19/2025

Syntax

```
BinaryFormat.Decimal(binary as binary) as any
```

About

A binary format that reads a .NET 16-byte decimal value.

BinaryFormat.Double

06/19/2025

Syntax

```
BinaryFormat.Double(binary as binary) as any
```

About

A binary format that reads an 8-byte IEEE double-precision floating point value.

BinaryFormat.Group

06/19/2025

Syntax

```
BinaryFormat.Group(binaryFormat as function, group as list, optional extra as nullable function, optional lastKey as any) as function
```

About

The parameters are as follows:

- The `binaryFormat` parameter specifies the binary format of the key value.
- The `group` parameter provides information about the group of known items.
- The optional `extra` parameter can be used to specify a function that will return a binary format value for the value following any key that was unexpected. If the `extra` parameter is not specified, then an error will be raised if there are unexpected key values.

The `group` parameter specifies a list of item definitions. Each item definition is a list, containing 3-5 values, as follows:

- Key value. The value of the key that corresponds to the item. This must be unique within the set of items.
- Item format. The binary format corresponding to the value of the item. This allows each item to have a different format.
- Item occurrence. The [BinaryOccurrence.Type](#) value for how many times the item is expected to appear in the group. Required items that are not present cause an error. Required or optional duplicate items are handled like unexpected key values.
- Default item value (optional). If the default item value appears in the item definition list and is not null, then it will be used instead of the default. The default for repeating or optional items is null, and the default for repeating values is an empty list { }.
- Item value transform (optional). If the item value transform function is present in the item definition list and is not null, then it will be called to transform the item value before it is returned. The transform function is only called if the item appears in the input (it will never be called with the default value).

Example 1

The following assumes a key value that is a single byte, with 4 expected items in the group, all of which have a byte of data following the key. The items appear in the input as follows:

- Key 1 is required, and does appear with value 11.
- Key 2 repeats, and appears twice with value 22, and results in a value of { 22, 22 }.
- Key 3 is optional, and does not appear, and results in a value of null.
- Key 4 repeats, but does not appear, and results in a value of { }.
- Key 5 is not part of the group, but appears once with value 55. The extra function is called with the key value 5, and returns the format corresponding to that value ([BinaryFormat.Byte](#)). The value 55 is read and discarded.

Usage

Power Query M

```
let
    b = #binary({
        1, 11,
        2, 22,
        2, 22,
        5, 55,
        1, 11
    }),
    f = BinaryFormat.Group(
        BinaryFormat.Byte,
        {
            {1, BinaryFormat.Byte, BinaryOccurrence.Required},
            {2, BinaryFormat.Byte, BinaryOccurrence.Repeating},
            {3, BinaryFormat.Byte, BinaryOccurrence.Optional},
            {4, BinaryFormat.Byte, BinaryOccurrence.Repeating}
        },
        (extra) => BinaryFormat.Byte
    )
in
    f(b)
```

Output

```
{11, {22, 22}, null, {}}
```

Example 2

The following example illustrates the item value transform and default item value. The repeating item with key 1 sums the list of values read using [List.Sum](#). The optional item with key 2 has a default value of 123 instead of null.

Usage

Power Query M

```
let
    b = #binary({
        1, 101,
        1, 102
    }),
    f = BinaryFormat.Group(
        BinaryFormat.Byte,
        {
            {1, BinaryFormat.Byte, BinaryOccurrence.Repeating,
              0, (list) => List.Sum(list)},
            {2, BinaryFormat.Byte, BinaryOccurrence.Optional, 123}
        }
    )
in
    f(b)
```

Output

```
{203, 123}
```

BinaryFormat.Length

06/19/2025

Syntax

```
BinaryFormat.Length(binaryFormat as function, length as any) as function
```

About

Returns a binary format that limits the amount of data that can be read. Both [BinaryFormat.List](#) and [BinaryFormat.Binary](#) can be used to read until end of the data. **BinaryFormat.Length** can be used to limit the number of bytes that are read. The `binaryFormat` parameter specifies the binary format to limit. The `length` parameter specifies the number of bytes to read. The `length` parameter may either be a number value, or a binary format value that specifies the format of the length value that appears that precedes the value being read.

Example 1

Limit the number of bytes read to 2 when reading a list of bytes.

Usage

Power Query M

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.Length(
        BinaryFormat.List(BinaryFormat.Byte),
        2
    )
in
    listFormat(binaryData)
```

Output

```
{1, 2}
```

Example 2

Limit the number of byte read when reading a list of bytes to the byte value preceding the list.

Usage

Power Query M

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.Length(
        BinaryFormat.List(BinaryFormat.Byte),
        BinaryFormat.Byte
    )
in
    listFormat(binaryData)
```

Output

```
{2}
```


BinaryFormat.List

06/19/2025

Syntax

```
BinaryFormat.List(binaryFormat as function, optional countOrCondition as any) as function
```

About

Returns a binary format that reads a sequence of items and returns a `list`. The `binaryFormat` parameter specifies the binary format of each item. There are three ways to determine the number of items read:

- If the `countOrCondition` is not specified, then the binary format will read until there are no more items.
- If the `countOrCondition` is a number, then the binary format will read that many items.
- If the `countOrCondition` is a function, then that function will be invoked for each item read. The function returns true to continue, and false to stop reading items. The final item is included in the list.
- If the `countOrCondition` is a binary format, then the count of items is expected to precede the list, and the specified format is used to read the count.

Example 1

Read bytes until the end of the data.

Usage

Power Query M

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.List(BinaryFormat.Byte)
in
    listFormat(binaryData)
```

Output

```
{1, 2, 3}
```

Example 2

Read two bytes.

Usage

Power Query M

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.List(BinaryFormat.Byte, 2)
in
    listFormat(binaryData)
```

Output

```
{1, 2}
```

Example 3

Read bytes until the byte value is greater than or equal to two.

Usage

Power Query M

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.List(BinaryFormat.Byte, (x) => x < 2)
in
    listFormat(binaryData)
```

Output

```
{1, 2}
```

BinaryFormat.Null

06/19/2025

Syntax

```
BinaryFormat.Null(binary as binary) as any
```

About

A binary format that reads zero bytes and returns null.

BinaryFormat.Record

06/19/2025

Syntax

```
BinaryFormat.Record(record as record) as function
```

About

Returns a binary format that reads a record. The `record` parameter specifies the format of the record. Each field in the record can have a different binary format. If a field contains a value that is not a binary format value, then no data is read for that field, and the field value is echoed to the result.

Example 1

Read a record containing one 16-bit integer and one 32-bit integer.

Usage

Power Query M

```
let
    binaryData = #binary({
        0x00, 0x01,
        0x00, 0x00, 0x00, 0x02
    }),
    recordFormat = BinaryFormat.Record([
        A = BinaryFormat.UnsignedInteger16,
        B = BinaryFormat.UnsignedInteger32
    ])
in
    recordFormat(binaryData)
```

Output

```
[A = 1, B = 2]
```

BinaryFormat.SignedInteger16

06/19/2025

Syntax

```
BinaryFormat.SignedInteger16(binary as binary) as any
```

About

A binary format that reads a 16-bit signed integer.

BinaryFormat.SignedInteger32

06/19/2025

Syntax

```
BinaryFormat.SignedInteger32(binary as binary) as any
```

About

A binary format that reads a 32-bit signed integer.

BinaryFormat.SignedInteger64

06/19/2025

Syntax

```
BinaryFormat.SignedInteger64(binary as binary) as any
```

About

A binary format that reads a 64-bit signed integer.

BinaryFormat.Single

06/19/2025

Syntax

```
BinaryFormat.Single(binary as binary) as any
```

About

A binary format that reads a 4-byte IEEE single-precision floating point value.

BinaryFormat.Text

06/19/2025

Syntax

```
BinaryFormat.Text(length as any, optional encoding as nullable number) as function
```

About

Returns a binary format that reads a text value. The `length` specifies the number of bytes to decode, or the binary format of the length that precedes the text. The optional `encoding` value specifies the encoding of the text. If the `encoding` is not specified, then the encoding is determined from the Unicode byte order marks. If no byte order marks are present, then `TextEncoding.Utf8` is used.

Example 1

Decode two bytes as ASCII text.

Usage

Power Query M

```
let
    binaryData = #binary({65, 66, 67}),
    textFormat = BinaryFormat.Text(2, TextEncoding.Ascii)
in
    textFormat(binaryData)
```

Output

```
"AB"
```

Example 2

Decode ASCII text where the length of the text in bytes appears before the text as a byte.

Usage

Power Query M

```
let
    binaryData = #binary({2, 65, 66}),
    textFormat = BinaryFormat.Text(
        BinaryFormat.Byte,
        TextEncoding.Ascii
    )
in
    textFormat(binaryData)
```

Output

"AB"

BinaryFormat.Transform

06/19/2025

Syntax

```
BinaryFormat.Transform(binaryFormat as function, function as function) as function
```

About

Returns a binary format that will transform the values read by another binary format. The `binaryFormat` parameter specifies the binary format that will be used to read the value. The `function` is invoked with the value read, and returns the transformed value.

Example 1

Read a byte and add one to it.

Usage

Power Query M

```
let
    binaryData = #binary({1}),
    transformFormat = BinaryFormat.Transform(
        BinaryFormat.Byte,
        (x) => x + 1
    )
in
    transformFormat(binaryData)
```

Output

2

BinaryFormat.UnsignedInteger16

06/19/2025

Syntax

```
BinaryFormat.UnsignedInteger16(binary as binary) as any
```

About

A binary format that reads a 16-bit unsigned integer.

BinaryFormat.UnsignedInteger32

06/19/2025

Syntax

```
BinaryFormat.UnsignedInteger32(binary as binary) as any
```

About

A binary format that reads a 32-bit unsigned integer.

BinaryFormat.UnsignedInteger64

06/19/2025

Syntax

```
BinaryFormat.UnsignedInteger64(binary as binary) as any
```

About

A binary format that reads a 64-bit unsigned integer.

#binary

06/19/2025

Syntax

```
#binary(value as any) as any
```

About

Creates a binary value from a list of numbers or a base 64 encoded text value.

Example 1

Create a binary value from a list of numbers.

Usage

Power Query M

```
#binary({0x30, 0x31, 0x32})
```

Output

```
Text.ToBinary("012")
```

Example 2

Create a binary value from a base 64 encoded text value.

Usage

Power Query M

```
#binary("1011")
```

Output

```
Binary.FromText("1011", BinaryEncoding.Base64)
```

Combiner functions

Article • 11/14/2024

These functions are used by other library functions that merge values. For example, [Table.ToList](#) and [Table.CombineColumns](#) apply a combiner function to each row in a table to produce a single value for each row.

 Expand table

Name	Description
Combiner.CombineTextByDelimiter	Returns a function that combines a list of text using the specified delimiter.
Combiner.CombineTextByEachDelimiter	Returns a function that combines a list of text using a sequence of delimiters.
Combiner.CombineTextByLengths	Returns a function that combines a list of text using the specified lengths.
Combiner.CombineTextByPositions	Returns a function that combines a list of text using the specified output positions.
Combiner.CombineTextByRanges	Returns a function that combines a list of text using the specified positions and lengths.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Combiner.CombineTextByDelimiter

06/19/2025

Syntax

```
Combiner.CombineTextByDelimiter(delimiter as text, optional quoteStyle as nullable number) as function
```

About

Returns a function that combines a list of text values into a single text value using the specified delimiter.

Example 1

Combine a list of text values using a semicolon delimiter.

Usage

Power Query M

```
Combiner.CombineTextByDelimiter(";")({"a", "b", "c"})
```

Output

```
"a;b;c"
```

Example 2

Combine the text of two columns using a comma delimiter and CSV-style quoting.

Usage

Power Query M

```
let
    Source = #table(
        type table [Column1 = text, Column2 = text],
        {{"a", "b"}, {"c", "d,e,f"}}
    ),
    Merged = Table.CombineColumns(
```

```
        Source,
        {"Column1", "Column2"},
        Combiner.CombineTextByDelimiter(",", QuoteStyle.Csv),
        "Merged"
    )
in
    Merged
```

Output

Power Query M

```
#table(
    type table [Merged = text],
    {"a,b"}, {"c,""d,e,f"""}
)
```

Combiner.CombineTextByEachDelimiter

06/19/2025

Syntax

```
Combiner.CombineTextByEachDelimiter(delimiters as list, optional quoteStyle as nullable number) as function
```

About

Returns a function that combines a list of text values into a single text value using a sequence of delimiters.

Example 1

Combine a list of text values using a sequence of delimiters.

Usage

Power Query M

```
Combiner.CombineTextByEachDelimiter({"=", "+"})({"a", "b", "c"})
```

Output

```
"a=b+c"
```

Combiner.CombineTextByLengths

06/19/2025

Syntax

```
Combiner.CombineTextByLengths(lengths as list, optional template as nullable text)  
as function
```

About

Returns a function that combines a list of text values into a single text value using the specified lengths.

Example 1

Combine a list of text values by extracting the specified numbers of characters from each input value.

Usage

Power Query M

```
Combiner.CombineTextByLengths({1, 2, 3})({"aaa", "bbb", "ccc"})
```

Output

```
"abbccc"
```

Example 2

Combine a list of text values by extracting the specified numbers of characters, after first pre-filling the result with the template text.

Usage

Power Query M

```
Combiner.CombineTextByLengths({1, 2, 3}, "*****")({"aaa", "bbb", "ccc"})
```

Output

```
"abbccc***"
```

Combiner.CombineTextByPositions

06/19/2025

Syntax

```
Combiner.CombineTextByPositions(positions as list, optional template as nullable text) as function
```

About

Returns a function that combines a list of text values into a single text value using the specified output positions.

Example 1

Combine a list of text values by placing them in the output at the specified positions.

Usage

Power Query M

```
Combiner.CombineTextByPositions({0, 5, 10})({"abc", "def", "ghi"})
```

Output

Power Query M

```
"abc  def  ghi"
```

Combiner.CombineTextByRanges

06/19/2025

Syntax

```
Combiner.CombineTextByRanges(ranges as list, optional template as nullable text)  
as function
```

About

Returns a function that combines a list of text values into a single text value using the specified output positions and lengths. A null length indicates that the entire text value should be included.

Example 1

Combine a list of text values using the specified output positions and lengths.

Usage

Power Query M

```
Combiner.CombineTextByRanges({{0, 1}, {3, 2}, {6, null}})({"abc", "def",  
"ghijkl"})
```

Output

Power Query M

```
"a  de  ghijkl"
```

Comparer functions

Article • 11/22/2024

These functions test equality and determine ordering.

 Expand table

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 and case-sensitivity.
Comparer.Ordinal	Returns a comparer function which uses Ordinal rules to compare values.
Comparer.OrdinalIgnoreCase	Returns a case-insensitive comparer function which uses Ordinal rules to compare the provided values.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Comparer.Equals

06/19/2025

Syntax

```
Comparer.Equals(comparer as function, x as any, y as any) as logical
```

About

Returns a `logical` value based on the equality check over the two given values, `x` and `y`, using the provided `comparer`.

`comparer` is a `Comparer` which is used to control the comparison. A comparer is a function that accepts two arguments and returns -1, 0, or 1 based on whether the first value is less than, equal to, or greater than the second. Comparers can be used to provide case-insensitive or culture and locale-aware comparisons.

The following built-in comparers are available in the formula language:

- [Comparer.Ordinal](#): Used to perform an exact ordinal comparison
- [Comparer.OrdinalIgnoreCase](#): Used to perform an exact ordinal case-insensitive comparison
- [Comparer.FromCulture](#): Used to perform a culture-aware comparison

Example 1

Compare "1" and "A" using "en-US" locale to determine if the values are equal.

Usage

Power Query M

```
Comparer.Equals(Comparer.FromCulture("en-US"), "1", "A")
```

Output

```
false
```

Related content

- How culture affects text formatting

Comparer.FromCulture

06/19/2025

Syntax

```
Comparer.FromCulture(culture as text, optional ignoreCase as nullable logical) as function
```

About

Returns a comparer function that uses the **culture** and the case-sensitivity specified by **ignoreCase** to perform comparisons.

A comparer function accepts two arguments and returns -1, 0, or 1 based on whether the first value is less than, equal to, or greater than the second.

The default value for **ignoreCase** is false. The **culture** should be one of the locales supported by the operating system (for example, "en-US").

Example 1

Compare "a" and "A" using "en-US" locale to determine if the values are equal.

Usage

Power Query M

```
Comparer.FromCulture("en-US")("a", "A")
```

Output

-1

Example 2

Compare "a" and "A" using "en-US" locale ignoring the case to determine if the values are equal.

Usage

Power Query M

```
Comparer.FromCulture("en-US", true)("a", "A")
```

Output

0

Comparer.Ordinal

06/19/2025

Syntax

```
Comparer.Ordinal(x as any, y as any) as number
```

About

Returns a comparer function which uses Ordinal rules to compare the provided values `x` and `y`.

Example 1

Using Ordinal rules, compare if "encyclopædia" and "encyclopaedia" are equivalent. Note these are equivalent using `Comparer.FromCulture("en-US")`.

A comparer function accepts two arguments and returns -1, 0, or 1 based on whether the first value is less than, equal to, or greater than the second.

Usage

```
powerquery-m
```

```
Comparer.Equals(Comparer.Ordinal, "encyclopædia", "encyclopaedia")
```

Output

```
false
```

Related content

- [How culture affects text formatting](#)

Comparer.OrdinalIgnoreCase

06/19/2025

Syntax

```
Comparer.OrdinalIgnoreCase(x as any, y as any) as number
```

About

Returns a case-insensitive comparer function which uses Ordinal rules to compare the provided values `x` and `y`.

A comparer function accepts two arguments and returns -1, 0, or 1 based on whether the first value is less than, equal to, or greater than the second.

Example 1

Using case-insensitive Ordinal rules, compare "Abc" with "abc". Note "Abc" is less than "abc" using `Comparer.Ordinal`.

Usage

Power Query M

```
Comparer.OrdinalIgnoreCase("Abc", "abc")
```

Output

0

Date functions

Article • 05/28/2025

These functions create and manipulate the date component of date, datetime, and datetimezone values.

 Expand table

Name	Description
Date.AddDays	Returns a Date/DateTime/DateTimeZone value with the day portion incremented by the number of days provided. It also handles incrementing the month and year portions of the value as appropriate.
Date.AddMonths	Returns a DateTime value with the month portion incremented by n months.
Date.AddQuarters	Returns a Date/DateTime/DateTimeZone value incremented by the number of quarters provided. Each quarter is defined as a duration of three months. It also handles incrementing the year portion of the value as appropriate.
Date.AddWeeks	Returns a Date/DateTime/DateTimeZone value incremented by the number of weeks provided. Each week is defined as a duration of seven days. It also handles incrementing the month and year portions of the value as appropriate.
Date.AddYears	Returns a DateTime value with the year portion incremented by n years.
Date.Day	Returns the day for a DateTime value.
Date.DayOfWeek	Returns a number (from 0 to 6) indicating the day of the week of the provided value.
Date.DayOfWeekName	Returns the day of the week name.
Date.DayOfYear	Returns a number that represents the day of the year from a DateTime value.
Date.DaysInMonth	Returns the number of days in the month from a DateTime value.
Date.EndOfDay	Returns the end of the day.
Date.EndOfMonth	Returns the end of the month.
Date.EndOfQuarter	Returns the end of the quarter.
Date.EndOfWeek	Returns the end of the week.
Date.EndOfYear	Returns the end of the year.
Date.From	Returns a date value from a value.

Name	Description
Date.FromText	Creates a Date from local, universal, and custom Date formats.
Date.IsInCurrentDay	Indicates whether the given datetime value <code>dateTime</code> occurs during the current day, as determined by the current date and time on the system.
Date.IsInCurrentMonth	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the current month, as determined by the current date and time on the system.
Date.IsInCurrentQuarter	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the current quarter, as determined by the current date and time on the system.
Date.IsInCurrentWeek	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the current week, as determined by the current date and time on the system.
Date.IsInCurrentYear	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the current year, as determined by the current date and time on the system.
Date.IsInNextDay	Indicates whether the given datetime value <code>dateTime</code> occurs during the next day, as determined by the current date and time on the system.
Date.IsInNextMonth	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the next month, as determined by the current date and time on the system.
Date.IsInNextNDays	Indicates whether the given datetime value <code>dateTime</code> occurs during the next number of days, as determined by the current date and time on the system.
Date.IsInNextNMonths	Indicates whether the given datetime value <code>dateTime</code> occurs during the next number of months, as determined by the current date and time on the system.
Date.IsInNextNQuarters	Indicates whether the given datetime value <code>dateTime</code> occurs during the next number of quarters, as determined by the current date and time on the system.
Date.IsInNextNWeeks	Indicates whether the given datetime value <code>dateTime</code> occurs during the next number of weeks, as determined by the current date and time on the system.
Date.IsInNextNYears	Indicates whether the given datetime value <code>dateTime</code> occurs during the next number of years, as determined by the current date and time on the system.
Date.IsInNextQuarter	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the next quarter, as

Name	Description
	determined by the current date and time on the system.
Date.IsInNextWeek	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the next week, as determined by the current date and time on the system.
Date.IsInNextYear	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the next year, as determined by the current date and time on the system.
Date.IsInPreviousDay	Indicates whether the given datetime value dateTime occurs during the previous day, as determined by the current date and time on the system.
Date.IsInPreviousMonth	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the previous month, as determined by the current date and time on the system.
Date.IsInPreviousNDays	Indicates whether the given datetime value dateTime occurs during the previous number of days, as determined by the current date and time on the system.
Date.IsInPreviousNMonths	Indicates whether the given datetime value dateTime occurs during the previous number of months, as determined by the current date and time on the system.
Date.IsInPreviousNQuarters	Indicates whether the given datetime value dateTime occurs during the previous number of quarters, as determined by the current date and time on the system.
Date.IsInPreviousNWeeks	Indicates whether the given datetime value dateTime occurs during the previous number of weeks, as determined by the current date and time on the system.
Date.IsInPreviousNYears	Indicates whether the given datetime value dateTime occurs during the previous number of years, as determined by the current date and time on the system.
Date.IsInPreviousQuarter	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the previous quarter, as determined by the current date and time on the system.
Date.IsInPreviousWeek	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the previous week, as determined by the current date and time on the system.
Date.IsInPreviousYear	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred during the previous year, as determined by the current date and time on the system.

Name	Description
Date.IsInYearToDate	Returns a logical value indicating whether the given Date/DateTime/DateTimeZone occurred in the period starting January 1st of the current year and ending on the current day, as determined by the current date and time on the system.
Date.IsLeapYear	Returns a logical value indicating whether the year portion of a DateTime value is a leap year.
Date.Month	Returns the month from a DateTime value.
Date.MonthName	Returns the name of the month component.
Date.QuarterOfYear	Returns a number between 1 and 4 for the quarter of the year from a DateTime value.
Date.StartOfDay	Returns the start of the day.
Date.StartOfMonth	Returns the start of the month.
Date.StartOfQuarter	Returns the start of the quarter.
Date.StartOfWeek	Returns the start of the week.
Date.StartOfYear	Returns the start of the year.
Date.ToRecord	Returns a record containing parts of a Date value.
Date.ToText	Returns a text value from a Date value.
Date.WeekOfMonth	Returns a number for the count of week in the current month.
Date.WeekOfYear	Returns a number for the count of week in the current year.
Date.Year	Returns the year from a DateTime value.
#date	Creates a date value from year, month, and day.

Date.AddDays

06/19/2025

Syntax

```
Date.AddDays(dateTime as any, numberOfDays as number) as any
```

About

Returns the `date`, `datetime`, or `datetimezone` result from adding `numberOfDays` days to the `dateTime` value `dateTime`.

- `dateTime`: The `date`, `datetime`, or `datetimezone` value to which days are being added.
- `numberOfDays`: The number of days to add.

Example 1

Add 5 days to the `date`, `datetime`, or `datetimezone` value representing the date 5/14/2011.

Usage

Power Query M

```
Date.AddDays(#date(2011, 5, 14), 5)
```

Output

```
#date(2011, 5, 19)
```

Related content

- [#date](#)
- [#datetime](#)
- [#datetimezone](#)

Date.AddMonths

06/19/2025

Syntax

```
Date.AddMonths(dateTime as any, numberOfMonths as number) as any
```

About

Returns the `date`, `datetime`, or `datetimezone` result from adding `numberOfMonths` months to the `datetime` value `dateTime`.

- `dateTime`: The `date`, `datetime`, or `datetimezone` value to which months are being added.
- `numberOfMonths`: The number of months to add.

Example 1

Add 5 months to the `date`, `datetime`, or `datetimezone` value representing the date 5/14/2011.

Usage

Power Query M

```
Date.AddMonths(#date(2011, 5, 14), 5)
```

Output

```
#date(2011, 10, 14)
```

Example 2

Add 18 months to the `date`, `datetime`, or `datetimezone` value representing the date and time of 5/14/2011 08:15:22 AM.

Usage

Power Query M

```
Date.AddMonths(#datetime(2011, 5, 14, 8, 15, 22), 18)
```

Output

```
#datetime(2012, 11, 14, 8, 15, 22)
```

Related content

- [#date](#)
- [#datetime](#)
- [#datetimezone](#)

Date.AddQuarters

06/19/2025

Syntax

```
Date.AddQuarters(dateTime as any, numberOfQuarters as number) as any
```

About

Returns the `date`, `datetime`, or `datetimezone` result from adding `numberOfQuarters` quarters to the `datetime` value `dateTime`.

- `dateTime`: The `date`, `datetime`, or `datetimezone` value to which quarters are being added.
- `numberOfQuarters`: The number of quarters to add.

Example 1

Add 1 quarter to the `date`, `datetime`, or `datetimezone` value representing the date 5/14/2011.

Usage

Power Query M

```
Date.AddQuarters(#date(2011, 5, 14), 1)
```

Output

```
#date(2011, 8, 14)
```

Related content

- [#date](#)
- [#datetime](#)
- [#datetimezone](#)

Date.AddWeeks

06/19/2025

Syntax

```
Date.AddWeeks(dateTime as any, numberOfWeeks as number) as any
```

About

Returns the `date`, `datetime`, or `datetimezone` result from adding `numberOfWeeks` weeks to the `datetime` value `dateTime`.

- `dateTime`: The `date`, `datetime`, or `datetimezone` value to which weeks are being added.
- `numberOfWeeks`: The number of weeks to add.

Example 1

Add 2 weeks to the `date`, `datetime`, or `datetimezone` value representing the date 5/14/2011.

Usage

Power Query M

```
Date.AddWeeks(#date(2011, 5, 14), 2)
```

Output

```
#date(2011, 5, 28)
```

Related content

- [#date](#)
- [#datetime](#)
- [#datetimezone](#)

Date.AddYears

06/19/2025

Syntax

```
Date.AddYears(dateTime as any, numberOfYears as number) as any
```

About

Returns the `date`, `datetime`, or `datetimezone` result of adding `numberOfYears` to a `datetime` value `dateTime`.

- `dateTime`: The `date`, `datetime`, or `datetimezone` value to which years are added.
- `numberOfYears`: The number of years to add.

Example 1

Add 4 years to the `date`, `datetime`, or `datetimezone` value representing the date 5/14/2011.

Usage

Power Query M

```
Date.AddYears(#date(2011, 5, 14), 4)
```

Output

```
#date(2015, 5, 14)
```

Example 2

Add 10 years to the `date`, `datetime`, or `datetimezone` value representing the date and time of 5/14/2011 08:15:22 AM.

Usage

Power Query M


```
Date.AddYears(#datetime(2011, 5, 14, 8, 15, 22), 10)
```

Output

```
#datetime(2021, 5, 14, 8, 15, 22)
```

Related content

- [#date](#)
- [#datetime](#)
- [#datetimezone](#)

Date.Day

06/19/2025

Syntax

```
Date.Day(dateTime as any) as nullable number
```

About

Returns the day component of a `date`, `datetime`, or `datetimezone` value.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from which the day component is extracted.

Example 1

Get the day component of a `date`, `datetime`, or `datetimezone` value representing the date and time of 5/14/2011 05:00:00 PM.

Usage

Power Query M

```
Date.Day(#datetime(2011, 5, 14, 17, 0, 0))
```

Output

14

Date.DayOfWeek

06/19/2025

Syntax

```
Date.DayOfWeek(dateTime as any, optional firstDayOfWeek as nullable number) as nullable number
```

About

Returns a number (from 0 to 6) indicating the day of the week of the provided `dateTime`.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value.
- `firstDayOfWeek`: A `Day` value indicating which day should be considered the first day of the week. Allowed values are [Day.Sunday](#), [Day.Monday](#), [Day.Tuesday](#), [Day.Wednesday](#), [Day.Thursday](#), [Day.Friday](#), or [Day.Saturday](#). If unspecified, a culture-dependent default is used.

Example 1

Get the day of the week represented by Monday, February 21st, 2011, treating Sunday as the first day of the week.

Usage

```
Power Query M
```

```
Date.DayOfWeek(#date(2011, 02, 21), Day.Sunday)
```

Output

```
1
```

Example 2

Get the day of the week represented by Monday, February 21st, 2011, treating Monday as the first day of the week.

Usage

Power Query M

```
Date.DayOfWeek(#date(2011, 02, 21), Day.Monday)
```

Output

0

Related content

- [How culture affects text formatting](#)

Date.DayOfWeekName

06/19/2025

Syntax

```
Date.DayOfWeekName(date as any, optional culture as nullable text) as nullable text
```

About

Returns the day of the week name for the provided `date`. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the day of the week name.

Usage

Power Query M

```
Date.DayOfWeekName(#date(2011, 12, 31), "en-US")
```

Output

```
"Saturday"
```

Related content

- [How culture affects text formatting](#)

Date.DayOfYear

06/19/2025

Syntax

```
Date.DayOfYear(dateTime as any) as nullable number
```

About

Returns a number representing the day of the year in the provided `date`, `datetime`, or `datetimezone` value, `dateTime`.

Example 1

The day of the year for March 1st, 2011.

Usage

Power Query M

```
Date.DayOfYear(#date(2011, 03, 01))
```

Output

60

Date.DaysInMonth

06/19/2025

Syntax

```
Date.DaysInMonth(dateTime as any) as nullable number
```

About

Returns the number of days in the month in the `date`, `datetime`, or `datetimezone` value `dateTime`.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value for which the number of days in the month is returned.

Example 1

Number of days in the month December as represented by `#date(2011, 12, 01)`.

Usage

Power Query M

```
Date.DaysInMonth(#date(2011, 12, 01))
```

Output

31

Date.EndOfDay

06/19/2025

```
Date.EndOfDay(dateTime as any) as any
```

About

Returns the end of the day represented by `dateTime`. Time zone information is preserved.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from from which the end of the day is calculated.

Example 1

Get the end of the day for 5/14/2011 05:00:00 PM.

Usage

```
Power Query M
```

```
Date.EndOfDay(#datetime(2011, 5, 14, 17, 0, 0))
```

Output

```
#datetime(2011, 5, 14, 23, 59, 59.9999999)
```

Example 2

Get the end of the day for 5/17/2011 05:00:00 PM -7:00.

Usage

```
Power Query M
```

```
Date.EndOfDay(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

Output

```
#datetimezone(2011, 5, 17, 23, 59, 59.9999999, -7, 0)
```


Date.EndOfMonth

06/19/2025

Syntax

```
Date.EndOfMonth(dateTime as any) as any
```

About

Returns the end of the month that contains `dateTime`.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from which the end of the month is calculated.

Example 1

Get the end of the month for 5/14/2011.

Usage

Power Query M

```
Date.EndOfMonth(#date(2011, 5, 14))
```

Output

```
#date(2011, 5, 31)
```

Example 2

Get the end of the month for 5/17/2011 05:00:00 PM -7:00.

Usage

Power Query M

```
Date.EndOfMonth(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

Output

```
#datetimezone(2011, 5, 31, 23, 59, 59.9999999, -7, 0)
```

Date.EndOfQuarter

06/19/2025

Syntax

```
Date.EndOfQuarter(dateTime as any) as any
```

About

Returns the end of the quarter that contains `dateTime`. Time zone information is preserved.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from which the end of the quarter is calculated.

Example 1

Find the end of the quarter for October 10th, 2011, 8:00AM.

Usage

```
Power Query M
```

```
Date.EndOfQuarter(#datetime(2011, 10, 10, 8, 0, 0))
```

Output

```
#datetime(2011, 12, 31, 23, 59, 59.9999999)
```

Date.EndOfWeek

06/19/2025

Syntax

```
Date.EndOfWeek(dateTime as any, optional firstDayOfWeek as nullable number) as any
```

About

Returns the end of the week that contains `dateTime`. This function takes an optional `Day`, `firstDayOfWeek`, to set as the first day of the week for this relative calculation. The default value is `Day.Sunday`.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from which the last day of the week is calculated
- `firstDayOfWeek`: *[Optional]* A `Day.Type` value representing the first day of the week. Possible values are `Day.Sunday`, `Day.Monday`, `Day.Tuesday`, `Day.Wednesday`, `Day.Thursday`, `Day.Friday` and `Day.Saturday`. The default value is `Day.Sunday`.

Example 1

Get the end of the week for 5/14/2011.

Usage

```
Power Query M
```

```
Date.EndOfWeek(#date(2011, 5, 14))
```

Output

```
#date(2011, 5, 14)
```

Example 2

Get the end of the week for 5/17/2011 05:00:00 PM -7:00, with Sunday as the first day of the week.

Usage

Power Query M

```
Date.EndOfWeek(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0), Day.Sunday)
```

Output

```
#datetimezone(2011, 5, 21, 23, 59, 59.9999999, -7, 0)
```

Date.EndOfYear

06/19/2025

Syntax

```
Date.EndOfYear(dateTime as any) as any
```

About

Returns the end of the year that contains `dateTime`, including fractional seconds. Time zone information is preserved.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value from which the end of the year is calculated.

Example 1

Get the end of the year for 5/14/2011 05:00:00 PM.

Usage

Power Query M

```
Date.EndOfYear(#datetime(2011, 5, 14, 17, 0, 0))
```

Output

```
#datetime(2011, 12, 31, 23, 59, 59.9999999)
```

Example 2

Get the end of hour for 5/17/2011 05:00:00 PM -7:00.

Usage

Power Query M

```
Date.EndOfYear(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

Output

```
#datetimezone(2011, 12, 31, 23, 59, 59.9999999, -7, 0)
```

Date.From

06/19/2025

Syntax

```
Date.From(value as any, optional culture as nullable text) as nullable date
```

About

Returns a date value from the given value.

- **value**: The value to covert to a date. If the given value is **null**, this function returns **null**. If the given value is **date**, **value** is returned. Values of the following types can be converted to a **date** value:
 - **text**: A **date** value from textual representation. Refer to [Date.FromText](#) for details.
 - **datetime**: The date component of the **value**.
 - **datetimezone**: The date component of the local datetime equivalent of **value**.
 - **number**: The date component of the datetime equivalent of a floating-point number whose integral component is the number of days before or after midnight, 30 December 1899, and whose fractional component represents the time on that day divided by 24. For example, midnight, 31 December 1899 is represented by 1.0; 6 A.M., 1 January 1900 is represented by 2.25; midnight, 29 December 1899 is represented by -1.0; and 6 A.M., 29 December 1899 is represented by -1.25. The base value is midnight, 30 December 1899. The minimum value is midnight, 1 January 0100. The maximum value is the last moment of 31 December 9999.

If **value** is of any other type, an error is returned.

- **culture**: The culture of the given value (for example, "en-US").

Example 1

Convert the specified date and time to a date value.

Usage


```
Date.From(#datetime(1899, 12, 30, 06, 45, 12))
```

Output

```
#date(1899, 12, 30)
```

Example 2

Convert the specified number to a date value.

Usage

Power Query M

```
Date.From(43910)
```

Output

```
#date(2020, 3, 20)
```

Example 3

Convert the German text dates in the Posted Date column to date values.

Usage

Power Query M

```
let
    Source = #table(type table [Account Code = text, Posted Date = text, Sales = number],
        {
            {"US-2004", "20 Januar 2023", 580},
            {"CA-8843", "18 Juli, 2023", 280},
            {"PA-1274", "12 Januar, 2022", 90},
            {"PA-4323", "14 April 2023", 187},
            {"US-1200", "14 Dezember, 2022", 350},
            {"PTY-507", "4 Juni, 2023", 110}
        }
    ),
    #"Filtered rows" = Table.TransformColumns(
        Source,
        {"Posted Date", each Date.From(_, "de-DE"), type date}
    )
in
    #"Filtered rows"
```

Output

Power Query M

```
#table(type table [Account Code = text, Posted Date = date, Sales = number],
{
    {"US-2004", #date(2023, 1, 20), 580},
    {"CA-8843", #date(2023, 7, 18), 280},
    {"PA-1274", #date(2022, 1, 12), 90},
    {"PA-4323", #date(2023, 4, 14), 187},
    {"US-1200", #date(2022, 12, 14), 350},
    {"PTY-507", #date(2023, 6, 4), 110}
})
```

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

Date.FromText

06/19/2025

Syntax

```
Date.FromText(text as nullable text, optional options as any) as nullable date
```

About

Creates a date value from a textual representation.

- **text**: A text value to covert to a date.
- **options**: An optional **record** that can be provided to specify additional properties. The **record** can contain the following fields:
 - **Format**: A **text** value indicating the format to use. For more details, go to [Standard date and time format strings](#) and [Custom date and time format strings](#). Omitting this field or providing **null** results in parsing the date using a best effort.
 - **Culture**: When **Format** isn't null, **Culture** controls some format specifiers. For example, in "en-US" "MMM" is "Jan", "Feb", "Mar", ..., while in "ru-RU" "MMM" is "ЯНВ", "ФЕВ", "МАР", ... When **Format** is null, **Culture** controls the default format to use. When **Culture** is null or omitted, [Culture.Current](#) is used.

To support legacy workflows, **options** can also be a text value. This has the same behavior as if **options** = [Format = null, Culture = options].

Example 1

Convert "2010-12-31" into a **date** value.

Usage

Power Query M

```
Date.FromText("2010-12-31")
```

Output

```
#date(2010, 12, 31)
```

Example 2

Convert using a custom format and the German culture.

Usage

Power Query M

```
Date.FromText("30 Dez 2010", [Format="dd MMM yyyy", Culture="de-DE"])
```

Output

```
#date(2010, 12, 30)
```

Example 3

Find the date in the Gregorian calendar that corresponds to the beginning of 1400 in the Hijri calendar.

Usage

Power Query M

```
Date.FromText("1400", [Format="yyyy", Culture="ar-SA"])
```

Output

```
#date(1979, 11, 20)
```

Example 4

Convert the Italian text dates with abbreviated months in the Posted Date column to date values.

Usage

Power Query M

```
let
    Source = #table(type table [Account Code = text, Posted Date = text, Sales = number],
        {
            {"US-2004", "20 gen. 2023", 580},
            {"CA-8843", "18 lug. 2024", 280},
            {"PA-1274", "12 gen. 2023", 90},
        }
    )
```

```

        {"PA-4323", "14 apr. 2023", 187},
        {"US-1200", "14 dic. 2023", 350},
        {"PTY-507", "4 giu. 2024", 110}
    }},
    #"Converted Date" = Table.TransformColumns(
        Source,
        {"Posted Date", each Date.FromText(_, [Culture = "it-IT"]), type date}
    )
in
    #"Converted Date"

```

Output

Power Query M

```

#table(type table [Account Code = text, Posted Date = date, Sales = number],
{
    {"US-2004", #date(2023, 1, 20), 580},
    {"CA-8843", #date(2024, 7, 18), 280},
    {"PA-1274", #date(2023, 1, 12), 90},
    {"PA-4323", #date(2023, 4, 14), 187},
    {"US-1200", #date(2023, 12, 14), 350},
    {"PTY-507", #date(2024, 6, 4), 110}
})

```

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

Date.IsInCurrentDay

06/19/2025

Syntax

```
Date.IsInCurrentDay(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current day, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example

Determine if the current system time is in the current day.

Usage

```
Power Query M
```

```
Date.IsInCurrentDay(DateTime.FixedLocalNow())
```

Output

```
true
```

Date.IsInCurrentMonth

06/19/2025

Syntax

```
Date.IsInCurrentMonth(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current month, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current month.

Usage

```
Power Query M
```

```
Date.IsInCurrentMonth(DateTime.FixedLocalNow())
```

Output

```
true
```

Date.IsInCurrentQuarter

06/19/2025

Syntax

```
Date.IsInCurrentQuarter(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current quarter, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current quarter.

Usage

```
Power Query M
```

```
Date.IsInCurrentQuarter(DateTime.FixedLocalNow())
```

Output

```
true
```


Date.IsInCurrentWeek

06/19/2025

Syntax

```
Date.IsInCurrentWeek(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current week, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current week.

Usage

```
Power Query M
```

```
Date.IsInCurrentWeek(DateTime.FixedLocalNow())
```

Output

```
true
```

Date.IsInCurrentYear

06/19/2025

Syntax

```
Date.IsInCurrentYear(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current year, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current year.

Usage

```
Power Query M
```

```
Date.IsInCurrentYear(DateTime.FixedLocalNow())
```

Output

```
true
```

Date.IsInNextDay

06/19/2025

Syntax

```
Date.IsInNextDay(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next day, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current day.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the day after the current system time is in the next day.

Usage

```
Power Query M
```

```
Date.IsInNextDay(Date.AddDays(DateTime.FixedLocalNow(), 1))
```

Output

```
true
```

Date.IsInNextMonth

06/19/2025

Syntax

```
Date.IsInNextMonth(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next month, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current month.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the month after the current system time is in the next month.

Usage

```
Power Query M
```

```
Date.IsInNextMonth(Date.AddMonths(DateTime.FixedLocalNow(), 1))
```

Output

```
true
```

Date.IsInNextNDays

06/19/2025

Syntax

```
Date.IsInNextNDays(dateTime as any, days as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of days, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current day.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `days`: The number of days.

Example 1

Determine if the day after the current system time is in the next two days.

Usage

```
Power Query M
```

```
Date.IsInNextNDays(Date.AddDays(DateTime.FixedLocalNow(), 1), 2)
```

Output

```
true
```

Date.IsInNextNMonths

06/19/2025

```
Date.IsInNextNMonths(dateTime as any, months as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of months, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current month.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `months`: The number of months.

Example 1

Determine if the month after the current system time is in the next two months.

Usage

```
Power Query M
```

```
Date.IsInNextNMonths(Date.AddMonths(DateTime.FixedLocalNow(), 1), 2)
```

Output

```
true
```

Date.IsInNextNQuarters

06/19/2025

Syntax

```
Date.IsInNextNQuarters(dateTime as any, quarters as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of quarters, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current quarter.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `quarters`: The number of quarters.

Example 1

Determine if the quarter after the current system time is in the next two quarters.

Usage

Power Query M

```
Date.IsInNextNQuarters(Date.AddQuarters(DateTime.FixedLocalNow(), 1), 2)
```

Output

```
true
```

Date.IsInNextNWeeks

06/19/2025

Syntax

```
Date.IsInNextNWeeks(dateTime as any, weeks as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of weeks, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current week.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `weeks`: The number of weeks.

Example 1

Determine if the week after the current system time is in the next two weeks.

Usage

Power Query M

```
Date.IsInNextNWeeks(Date.AddDays(DateTime.FixedLocalNow(), 7), 2)
```

Output

```
true
```


Date.IsInNextNYears

06/19/2025

Syntax

```
Date.IsInNextNYears(dateTime as any, years as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of years, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current year.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `years`: The number of years.

Example 1

Determine if the year after the current system time is in the next two years.

Usage

```
Power Query M
```

```
Date.IsInNextNYears(Date.AddYears(DateTime.FixedLocalNow(), 1), 2)
```

Output

```
true
```

Date.IsInNextQuarter

06/19/2025

Syntax

```
Date.IsInNextQuarter(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next quarter, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current quarter.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the quarter after the current system time is in the next quarter.

Usage

Power Query M

```
Date.IsInNextQuarter(Date.AddQuarters(DateTime.FixedLocalNow(), 1))
```

Output

```
true
```

Date.IsInNextWeek

06/19/2025

Syntax

```
Date.IsInNextWeek(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next week, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current week.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the week after the current system time is in the next week.

Usage

Power Query M

```
Date.IsInNextWeek(Date.AddDays(DateTime.FixedLocalNow(), 7))
```

Output

```
true
```

Date.IsInNextYear

06/19/2025

Syntax

```
Date.IsInNextYear(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next year, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current year.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the year after the current system time is in the next year.

Usage

Power Query M

```
Date.IsInNextYear(Date.AddYears(DateTime.FixedLocalNow(), 1))
```

Output

true

Date.IsInPreviousDay

06/19/2025

Syntax

```
Date.IsInPreviousDay(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous day, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current day.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the day before the current system time is in the previous day.

Usage

Power Query M

```
Date.IsInPreviousDay(Date.AddDays(DateTime.FixedLocalNow(), -1))
```

Output

```
true
```

Date.IsInPreviousMonth

06/19/2025

Syntax

```
Date.IsInPreviousMonth(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous month, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current month.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the month before the current system time is in the previous month.

Usage

Power Query M

```
Date.IsInPreviousMonth(Date.AddMonths(DateTime.FixedLocalNow(), -1))
```

Output

```
true
```

Date.IsInPreviousNDays

06/19/2025

Syntax

```
Date.IsInPreviousNDays(dateTime as any, days as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of days, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current day.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `days`: The number of days.

Example 1

Determine if the day before the current system time is in the previous two days.

Usage

Power Query M

```
Date.IsInPreviousNDays(Date.AddDays(DateTime.FixedLocalNow(), -1), 2)
```

Output

```
true
```

Date.IsInPreviousNMonths

06/19/2025

Syntax

```
Date.IsInPreviousNMonths(dateTime as any, months as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of months, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current month.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `months`: The number of months.

Example 1

Determine if the month before the current system time is in the previous two months.

Usage

Power Query M

```
Date.IsInPreviousNMonths(Date.AddMonths(DateTime.FixedLocalNow(), -1), 2)
```

Output

```
true
```


Date.IsInPreviousNQuarters

06/19/2025

Syntax

```
Date.IsInPreviousNQuarters(dateTime as any, quarters as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of quarters, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current quarter.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `quarters`: The number of quarters.

Example 1

Determine if the quarter before the current system time is in the previous two quarters.

Usage

Power Query M

```
Date.IsInPreviousNQuarters(Date.AddQuarters(DateTime.FixedLocalNow(), -1), 2)
```

Output

true

Date.IsInPreviousNWeeks

06/19/2025

Syntax

```
Date.IsInPreviousNWeeks(dateTime as any, weeks as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of weeks, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current week.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `weeks`: The number of weeks.

Example 1

Determine if the week before the current system time is in the previous two weeks.

Usage

```
Power Query M
```

```
Date.IsInPreviousNWeeks(Date.AddDays(DateTime.FixedLocalNow(), -7), 2)
```

Output

```
true
```

Date.IsInPreviousNYears

06/19/2025

Syntax

```
Date.IsInPreviousNYears(dateTime as any, years as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of years, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current year.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.
- `years`: The number of years.

Example 1

Determine if the year before the current system time is in the previous two years.

Usage

Power Query M

```
Date.IsInPreviousNYears(Date.AddYears(DateTime.FixedLocalNow(), -1), 2)
```

Output

```
true
```

Date.IsInPreviousQuarter

06/19/2025

Syntax

```
Date.IsInPreviousQuarter(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous quarter, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current quarter.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the quarter before the current system time is in the previous quarter.

Usage

Power Query M

```
Date.IsInPreviousQuarter(Date.AddQuarters(DateTime.FixedLocalNow(), -1))
```

Output

```
true
```

Date.IsInPreviousWeek

06/19/2025

Syntax

```
Date.IsInPreviousWeek(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous week, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current week.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the week before the current system time is in the previous week.

Usage

Power Query M

```
Date.IsInPreviousWeek(Date.AddDays(DateTime.FixedLocalNow(), -7))
```

Output

```
true
```

Date.IsInPreviousYear

06/19/2025

Syntax

```
Date.IsInPreviousYear(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous year, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current year.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the year before the current system time is in the previous year.

Usage

Power Query M

```
Date.IsInPreviousYear(Date.AddYears(DateTime.FixedLocalNow(), -1))
```

Output

```
true
```

Date.IsInYearToDate

06/19/2025

Syntax

```
Date.IsInYearToDate(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current year and is on or before the current day, as determined by the current date and time on the system.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the year to date.

Usage

```
Power Query M
```

```
Date.IsInYearToDate(DateTime.FixedLocalNow())
```

Output

```
true
```

Date.IsLeapYear

06/19/2025

Syntax

```
Date.IsLeapYear(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` falls in is a leap year.

- `dateTime`: A `date`, `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the year 2012, as represented by `#date(2012, 01, 01)` is a leap year.

Usage

Power Query M

```
Date.IsLeapYear(#date(2012, 01, 01))
```

Output

```
true
```


Date.Month

06/19/2025

Syntax

```
Date.Month(dateTime as any) as nullable number
```

About

Returns the month component of the provided `datetime` value, `dateTime`.

Example 1

Find the month in `#datetime(2011, 12, 31, 9, 15, 36)`.

Usage

Power Query M

```
Date.Month(#datetime(2011, 12, 31, 9, 15, 36))
```

Output

12

Date.MonthName

06/19/2025

Syntax

```
Date.MonthName(date as any, optional culture as nullable text) as nullable text
```

About

Returns the name of the month component for the provided `date`. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the month name.

Usage

Power Query M

```
Date.MonthName(#datetime(2011, 12, 31, 5, 0, 0), "en-US")
```

Output

```
"December"
```

Related content

- [How culture affects text formatting](#)

Date.QuarterOfYear

06/19/2025

Syntax

```
Date.QuarterOfYear(dateTime as any) as nullable number
```

About

Returns a number from 1 to 4 indicating which quarter of the year the date `dateTime` falls in. `dateTime` can be a `date`, `datetime`, or `datetimezone` value.

Example 1

Find which quarter of the year the date `#date(2011, 12, 31)` falls in.

Usage

Power Query M

```
Date.QuarterOfYear(#date(2011, 12, 31))
```

Output

4

Date.StartOfDay

06/19/2025

Syntax

```
Date.StartOfDay(dateTime as any) as any
```

About

Returns the start of the day represented by `dateTime`. `dateTime` must be a `date`, `datetime`, or `datetimezone` value.

Example 1

Find the start of the day for October 10th, 2011, 8:00AM.

Usage

Power Query M

```
Date.StartOfDay(#datetime(2011, 10, 10, 8, 0, 0))
```

Output

```
#datetime(2011, 10, 10, 0, 0, 0)
```

Date.StartOfMonth

06/19/2025

Syntax

```
Date.StartOfMonth(dateTime as any) as any
```

About

Returns the start of the month that contains `dateTime`. `dateTime` must be a `date` or `datetime` value.

Example 1

Find the start of the month for October 10th, 2011, 8:10:32AM.

Usage

Power Query M

```
Date.StartOfMonth(#datetime(2011, 10, 10, 8, 10, 32))
```

Output

```
#datetime(2011, 10, 1, 0, 0, 0)
```

Date.StartOfQuarter

06/19/2025

Syntax

```
Date.StartOfQuarter(dateTime as any) as any
```

About

Returns the start of the quarter that contains `dateTime`. `dateTime` must be a `date`, `datetime`, or `datetimezone` value.

Example 1

Find the start of the quarter for October 10th, 2011, 8:00AM.

Usage

Power Query M

```
Date.StartOfQuarter(#datetime(2011, 10, 10, 8, 0, 0))
```

Output

```
#datetime(2011, 10, 1, 0, 0, 0)
```

Date.StartOfWeek

06/19/2025

Syntax

```
Date.StartOfWeek(dateTime as any, optional firstDayOfWeek as nullable number) as any
```

About

Returns the start of the week that contains `dateTime`. `dateTime` must be a `date`, `datetime`, or `datetimezone` value.

Example 1

Find the start of the week for Tuesday, October 11th, 2011.

Usage

Power Query M

```
Date.StartOfWeek(#datetime(2011, 10, 11, 8, 10, 32))
```

Output

Power Query M

```
// Sunday, October 9th, 2011  
#datetime(2011, 10, 9, 0, 0, 0)
```

Example 2

Find the start of the week for Tuesday, October 11th, 2011, using Monday as the start of the week.

Usage

Power Query M

```
Date.StartOfWeek(#datetime(2011, 10, 11, 8, 10, 32), Day.Monday)
```

Output

Power Query M

```
// Monday, October 10th, 2011  
#datetime(2011, 10, 10, 0, 0, 0)
```


Date.StartOfYear

06/19/2025

Syntax

```
Date.StartOfYear(dateTime as any) as any
```

About

Returns the start of the year that contains `dateTime`. `dateTime` must be a `date`, `datetime`, or `datetimezone` value.

Example 1

Find the start of the year for October 10th, 2011, 8:10:32AM.

Usage

Power Query M

```
Date.StartOfYear(#datetime(2011, 10, 10, 8, 10, 32))
```

Output

```
#datetime(2011, 1, 1, 0, 0, 0)
```

Date.ToRecord

06/19/2025

Syntax

```
Date.ToRecord(date as date) as record
```

About

Returns a record containing the parts of the given date value, `date`.

- `date`: A `date` value for from which the record of its parts is to be calculated.

Example 1

Convert the `#date(2011, 12, 31)` value into a record containing parts from the date value.

Usage

Power Query M

```
Date.ToRecord(#date(2011, 12, 31))
```

Output

Power Query M

```
[
    Year = 2011,
    Month = 12,
    Day = 31
]
```

Date.ToText

06/19/2025

Syntax

```
Date.ToText(date as nullable date, optional options as any, optional culture as nullable text) as nullable text
```

About

Returns a textual representation of `date`. An optional `record` parameter, `options`, may be provided to specify additional properties. `culture` is only used for legacy workflows. The `record` can contain the following fields:

- Format**: A `text` value indicating the format to use. For more details, go to [Standard date and time format strings](#) and [Custom date and time format strings](#). Omitting this field or providing `null` will result in formatting the date using the default defined by `Culture`.
- Culture**: When `Format` is not null, `Culture` controls some format specifiers. For example, in "en-US" "MMM" is "Jan", "Feb", "Mar", ..., while in "ru-RU" "MMM" is "янв", "фев", "мар", ... When `Format` is null, `Culture` controls the default format to use. When `Culture` is null or omitted, [Culture.Current](#) is used.

To support legacy workflows, `options` and `culture` may also be text values. This has the same behavior as if `options = [Format = options, Culture = culture]`.

Example 1

Convert `#date(2010, 12, 31)` into a `text` value. *Result output may vary depending on current culture.*

Usage

Power Query M

```
Date.ToText(#date(2010, 12, 31))
```

Output

"12/31/2010"

Example 2

Convert using a custom format and the German culture.

Usage

Power Query M

```
Date.ToText(#date(2010, 12, 31), [Format="dd MMM yyyy", Culture="de-DE"])
```

Output

"31 Dez 2010"

Example 3

Find the year in the Hijri calendar that corresponds to January 1st, 2000 in the Gregorian calendar.

Usage

Power Query M

```
Date.ToText(#date(2000, 1, 1), [Format="yyyy", Culture="ar-SA"])
```

Output

"1420"

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

Date.WeekOfMonth

06/19/2025

Syntax

```
Date.WeekOfMonth(dateTime as any, optional firstDayOfWeek as nullable number) as nullable number
```

About

Returns a number from 1 to 6 indicating which week of the month the date `dateTime` falls in.

- `dateTime`: A `datetime` value for which the week-of-the-month is determined.

Example 1

Determine which week of March the 15th falls on in 2011.

Usage

```
Power Query M
```

```
Date.WeekOfMonth(#date(2011, 03, 15))
```

Output

```
3
```

Date.WeekOfYear

06/19/2025

Syntax

```
Date.WeekOfYear(dateTime as any, optional firstDayOfWeek as nullable number) as nullable number
```

About

Returns a number from 1 to 54 indicating which week of the year the date, `dateTime`, falls in.

- `dateTime`: A `datetime` value for which the week-of-the-year is determined.
- `firstDayOfWeek`: An optional `Day.Type` value that indicates which day is considered the start of a new week (for example, `Day.Sunday`). If unspecified, a culture-dependent default is used.

Example 1

Determine which week of the year contains March 27th, 2011.

Usage

Power Query M

```
Date.WeekOfYear(#date(2011, 03, 27))
```

Output

14

Example 2

Determine which week of the year contains March 27th, 2011, using Monday as the start of the week.

Usage

Power Query M

```
Date.WeekOfYear(#date(2011, 03, 27), Day.Monday)
```

Output

13

Related content

- [How culture affects text formatting](#)

Date.Year

06/19/2025

```
Date.Year(dateTime as any) as nullable number
```

About

Returns the year component of the provided `datetime` value, `dateTime`.

Example 1

Find the year in `#datetime(2011, 12, 31, 9, 15, 36)`.

Usage

Power Query M

```
Date.Year(#datetime(2011, 12, 31, 9, 15, 36))
```

Output

```
2011
```


#date

06/19/2025

Syntax

```
#date(year as number, month as number, day as number) as date
```

About

Creates a date value from whole numbers representing the year, month, and day. Raises an error if these conditions are not true:

- $1 \leq \text{year} \leq 9999$
- $1 \leq \text{month} \leq 12$
- $1 \leq \text{day} \leq 31$

Example 1

Create a date representing December 26, 2023.

Usage

Power Query M

```
#date(2023, 12, 26)
```

Output

```
#date(2023, 12, 26)
```

Example 2

Convert a date to text using a custom format and the German culture.

Usage

Power Query M

```
Date.ToText(#date(2023, 12, 26), [Format="dd MMM yyyy", Culture="de-DE"])
```

Output

"26 Dez 2023"

Example 3

Get the rows from a table that contain a date in 2023.

Usage

Power Query M

```
let
Source = #table(type table [Account Code = text, Posted Date = date, Sales = number],
    {
        {"US-2004", #date(2023,1,20), 580},
        {"CA-8843", #date(2023,7,18), 280},
        {"PA-1274", #date(2022,1,12), 90},
        {"PA-4323", #date(2023,4,14), 187},
        {"US-1200", #date(2022,12,14), 350},
        {"PTY-507", #date(2023,6,4), 110}
    }),
#"Filtered rows" = Table.SelectRows(
    Source,
    each Date.Year([Posted Date]) = 2023
)
in
    #"Filtered rows"
```

Output

Power Query M

```
#table (type table [Account Code = text, Posted Date = date, Sales = number],
{
    {"US-2004", #date(2023, 1, 20), 580},
    {"CA-8843", #date(2023, 7, 18), 280},
    {"PA-4323", #date(2023, 4, 14), 187},
    {"PTY-507", #date(2023, 6, 4), 110}
})
```

DateTime functions

Article • 11/22/2024

These functions create and manipulate datetime and datetimezone values.

 Expand table

Name	Description
DateTime.AddZone	Adds timezone information to the datetime value.
DateTime.Date	Returns the date component of the given date, datetime, or datetimezone value.
DateTime.FixedLocalNow	Returns the current date and time in the local timezone. This value is fixed and doesn't change with successive calls.
DateTime.From	Creates a datetime from the given value.
DateTime.FromFileTime	Creates a datetime from a 64-bit long number.
DateTime.FromText	Creates a datetime from local and universal datetime formats.
DateTime.IsInCurrentHour	Indicates whether this datetime occurs during the current hour, as determined by the current date and time on the system.
DateTime.IsInCurrentMinute	Indicates whether this datetime occurs during the current minute, as determined by the current date and time on the system.
DateTime.IsInCurrentSecond	Indicates whether this datetime occurs during the current second, as determined by the current date and time on the system.
DateTime.IsInNextHour	Indicates whether this datetime occurs during the next hour, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current hour.
DateTime.IsInNextMinute	Indicates whether this datetime occurs during the next minute, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current minute.
DateTime.IsInNextNHours	Indicates whether this datetime occurs during the next number of hours, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current hour.

Name	Description
DateTime.IsInNextNMinutes	Indicates whether this datetime occurs during the next number of minutes, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current minute.
DateTime.IsInNextNSeconds	Indicates whether this datetime occurs during the next number of seconds, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current second.
DateTime.IsInNextSecond	Indicates whether this datetime occurs during the next second, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current second.
DateTime.IsInPreviousHour	Indicates whether this datetime occurs during the previous hour, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current hour.
DateTime.IsInPreviousMinute	Indicates whether this datetime occurs during the previous minute, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current minute.
DateTime.IsInPreviousNHours	Indicates whether this datetime occurs during the previous number of hours, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current hour.
DateTime.IsInPreviousNMinutes	Indicates whether this datetime occurs during the previous number of minutes, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current minute.
DateTime.IsInPreviousNSeconds	Indicates whether this datetime occurs during the previous number of seconds, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current second.
DateTime.IsInPreviousSecond	Indicates whether this datetime occurs during the previous second, as determined by the current date and time on the system. This function returns <code>false</code> when passed a value that occurs within the current second.
DateTime.LocalNow	Returns the current date and time in the local timezone.
DateTime.Time	Returns the time part of the given datetime value.

Name	Description
DateTime.ToRecord	Returns a record containing the datetime value's parts.
DateTime.ToText	Returns a textual representation of the datetime value.
#datetime	Creates a datetime value from year, month, day, hour, minute, and second.

Feedback

Was this page helpful?



Yes



No

[Provide product feedback](#) | [Ask the community](#) 

DateTime.AddZone

06/19/2025

Syntax

```
DateTime.AddZone(dateTime as nullable datetime, timezoneHours as number, optional  
timezoneMinutes as nullable number) as nullable datetimetype
```

About

Adds timezone information to the `dateTime` value. The timezone information includes `timezoneHours` and optionally `timezoneMinutes`, which specify the desired offset from UTC time.

Example 1

Set the timezone to UTC+7:30 (7 hours and 30 minutes past UTC).

Usage

Power Query M

```
DateTime.AddZone(#datetime(2010, 12, 31, 11, 56, 02), 7, 30)
```

Output

```
#datetimezone(2010, 12, 31, 11, 56, 2, 7, 30)
```

DateTime.Date

06/19/2025

Syntax

```
DateTime.Date(dateTime as any) as nullable date
```

About

Returns the date component of the `dateTime` parameter if the parameter is a `date`, `datetime`, or `datetimezone` value, or `null` if the parameter is `null`.

Example 1

Find date value of `#datetime(2010, 12, 31, 11, 56, 02)`.

Usage

Power Query M

```
DateTime.Date(#datetime(2010, 12, 31, 11, 56, 02))
```

Output

```
#date(2010, 12, 31)
```

DateTime.FixedLocalNow

07/08/2025

Syntax

```
DateTime.FixedLocalNow() as datetime
```

About

Returns a `datetime` value set to the current date and time on the system. This value is fixed and will not change with successive calls, unlike [DateTime.LocalNow](#), which may return different values over the course of execution of an expression.

Related content

[Local, fixed, and UTC variants of current time functions](#)

DateTime.From

06/19/2025

Syntax

```
DateTime.From(value as any, optional culture as nullable text) as nullable datetime
```

About

Returns a `datetime` value from the given `value`. An optional `culture` may also be provided (for example, "en-US"). If the given `value` is `null`, **DateTime.From** returns `null`. If the given `value` is `datetime`, `value` is returned. Values of the following types can be converted to a `datetime` value:

- `text`: A `datetime` value from textual representation. Refer to [DateTime.FromText](#) for details.
- `date`: A `datetime` with `value` as the date component and `12:00:00 AM` as the time component.
- `datetimezone`: The local `datetime` equivalent of `value`.
- `time`: A `datetime` with the date equivalent of the OLE Automation Date of `0` as the date component and `value` as the time component.
- `number`: A `datetime` equivalent of the OLE Automation Date expressed by `value`.

If `value` is of any other type, an error is returned.

Example 1

Convert `#time(06, 45, 12)` to a `datetime` value.

Usage

Power Query M

```
DateTime.From(#time(06, 45, 12))
```

Output

```
#datetime(1899, 12, 30, 06, 45, 12)
```

Example 2

Convert `#date(1975, 4, 4)` to a `datetime` value.

Usage

Power Query M

```
DateTime.From(#date(1975, 4, 4))
```

Output

```
#datetime(1975, 4, 4, 0, 0, 0)
```

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

DateTime.FromFileTime

06/19/2025

Syntax

```
DateTime.FromFileTime(fileTime as nullable number) as nullable datetime
```

About

Creates a `datetime` value from the `fileTime` value and converts it to the local time zone. The filetime is a Windows file time value that represents the number of 100-nanosecond intervals that have elapsed since 12:00 midnight, January 1, 1601 A.D. (C.E.) Coordinated Universal Time (UTC).

Example 1

Convert `129876402529842245` into a datetime value.

Usage

Power Query M

```
DateTime.FromFileTime(129876402529842245)
```

Output

```
#datetime(2012, 7, 24, 14, 50, 52.9842245)
```

DateTime.FromText

06/19/2025

Syntax

```
DateTime.FromText(text as nullable text, optional options as any) as nullable datetime
```

About

Creates a `datetime` value from a textual representation, `text`. An optional `record` parameter, `options`, may be provided to specify additional properties. The `record` can contain the following fields:

- **Format**: A `text` value indicating the format to use. For more details, go to [Standard date and time format strings](#) and [Custom date and time format strings](#). Omitting this field or providing `null` will result in parsing the date using a best effort.
- **Culture**: When `Format` is not null, `Culture` controls some format specifiers. For example, in "en-US" "MMM" is "Jan", "Feb", "Mar", ..., while in "ru-RU" "MMM" is "янв", "фев", "мар", ... When `Format` is `null`, `Culture` controls the default format to use. When `Culture` is `null` or omitted, [Culture.Current](#) is used.

To support legacy workflows, `options` may also be a text value. This has the same behavior as if `options = [Format = null, Culture = options]`.

Example 1

Convert "2010-12-31T01:30:00" into a datetime value.

Usage

Power Query M

```
DateTime.FromText("2010-12-31T01:30:25")
```

Output

```
#datetime(2010, 12, 31, 1, 30, 25)
```

Example 2

Convert "2010-12-31T01:30:00.121212" into a datetime value.

Usage

Power Query M

```
DateTime.FromText("30 Dez 2010 02:04:50.369730", [Format="dd MMM yyyy  
HH:mm:ss.ffffff", Culture="de-DE"])
```

Output

```
#datetime(2010, 12, 30, 2, 4, 50.36973)
```

Example 3

Convert "2010-12-31T01:30:00" into a datetime value.

Usage

Power Query M

```
DateTime.FromText("2000-02-08T03:45:12Z", [Format="yyyy-MM-dd'T'HH:mm:ss'Z'",  
Culture="en-US"])
```

Output

```
#datetime(2000, 2, 8, 3, 45, 12)
```

Example 4

Convert "20101231T013000" into a datetime value.

Usage

Power Query M

```
DateTime.FromText("20101231T013000", [Format="yyyyMMdd'T'HHmmss", Culture="en-  
US"])
```

Output

```
#datetime(2010, 12, 31, 1, 30, 0)
```

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

DateTime.IsInCurrentHour

06/19/2025

Syntax

```
DateTime.IsInCurrentHour(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current hour, as determined by the current date and time on the system.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current hour.

Usage

```
Power Query M
```

```
DateTime.IsInCurrentHour(DateTime.FixedLocalNow())
```

Output

```
true
```

DateTime.IsInCurrentMinute

06/19/2025

Syntax

```
DateTime.IsInCurrentMinute(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current minute, as determined by the current date and time on the system.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current minute.

Usage

```
Power Query M
```

```
DateTime.IsInCurrentMinute(DateTime.FixedLocalNow())
```

Output

```
true
```


DateTime.IsInCurrentSecond

06/19/2025

Syntax

```
DateTime.IsInCurrentSecond(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the current second, as determined by the current date and time on the system.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the current system time is in the current second.

Usage

```
Power Query M
```

```
DateTime.IsInCurrentSecond(DateTime.FixedLocalNow())
```

Output

```
true
```

DateTime.IsInNextHour

06/19/2025

Syntax

```
DateTime.IsInNextHour(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next hour, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current hour.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the hour after the current system time is in the next hour.

Usage

Power Query M

```
DateTime.IsInNextHour(DateTime.FixedLocalNow() + #duration(0, 1, 0, 0))
```

Output

true

DateTime.IsInNextMinute

06/19/2025

Syntax

```
DateTime.IsInNextMinute(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next minute, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current minute.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the minute after the current system time is in the next minute.

Usage

Power Query M

```
DateTime.IsInNextMinute(DateTime.FixedLocalNow() + #duration(0, 0, 1, 0))
```

Output

true

DateTime.IsInNextNHours

06/19/2025

Syntax

```
DateTime.IsInNextNHours(dateTime as any, hours as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of hours, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current hour.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.
- `hours`: The number of hours.

Example 1

Determine if the hour after the current system time is in the next two hours.

Usage

Power Query M

```
DateTime.IsInNextNHours(DateTime.FixedLocalNow() + #duration(0, 2, 0, 0), 2)
```

Output

```
true
```

DateTime.IsInNextNMinutes

06/19/2025

Syntax

```
DateTime.IsInNextNMinutes(dateTime as any, minutes as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of minutes, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current minute.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.
- `minutes`: The number of minutes.

Example 1

Determine if the minute after the current system time is in the next two minutes.

Usage

Power Query M

```
DateTime.IsInNextNMinutes(DateTime.FixedLocalNow() + #duration(0, 0, 2, 0), 2)
```

Output

```
true
```

DateTime.IsInNextNSeconds

06/19/2025

Syntax

```
DateTime.IsInNextNSeconds(dateTime as any, seconds as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next number of seconds, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current second.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.
- `seconds`: The number of seconds.

Example 1

Determine if the second after the current system time is in the next two seconds.

Usage

Power Query M

```
DateTime.IsInNextNSeconds(DateTime.FixedLocalNow() + #duration(0, 0, 0, 2), 2)
```

Output

```
true
```

DateTime.IsInNextSecond

06/19/2025

Syntax

```
DateTime.IsInNextSecond(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the next second, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current second.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the second after the current system time is in the next second.

Usage

Power Query M

```
DateTime.IsInNextSecond(DateTime.FixedLocalNow() + #duration(0, 0, 0, 1))
```

Output

```
true
```

DateTime.IsInPreviousHour

06/19/2025

Syntax

```
DateTime.IsInPreviousHour(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous hour, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current hour.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the hour before the current system time is in the previous hour.

Usage

Power Query M

```
DateTime.IsInPreviousHour(DateTime.FixedLocalNow() - #duration(0, 1, 0, 0))
```

Output

```
true
```


DateTime.IsInPreviousMinute

06/19/2025

Syntax

```
DateTime.IsInPreviousMinute(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous minute, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current minute.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the minute before the current system time is in the previous minute.

Usage

Power Query M

```
DateTime.IsInPreviousMinute(DateTime.FixedLocalNow() - #duration(0, 0, 1, 0))
```

Output

```
true
```

DateTime.IsInPreviousNHours

06/19/2025

Syntax

```
DateTime.IsInPreviousNHours(dateTime as any, hours as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of hours, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current hour.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.
- `hours`: The number of hours.

Example 1

Determine if the hour before the current system time is in the previous two hours.

Usage

Power Query M

```
DateTime.IsInPreviousNHours(DateTime.FixedLocalNow() - #duration(0, 2, 0, 0), 2)
```

Output

```
true
```

DateTime.IsInPreviousNMinutes

06/19/2025

Syntax

```
DateTime.IsInPreviousNMinutes(dateTime as any, minutes as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of minutes, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current minute.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.
- `minutes`: The number of minutes.

Example 1

Determine if the minute before the current system time is in the previous two minutes.

Usage

Power Query M

```
DateTime.IsInPreviousNMinutes(DateTime.FixedLocalNow() - #duration(0, 0, 2, 0), 2)
```

Output

```
true
```

DateTime.IsInPreviousNSeconds

06/19/2025

Syntax

```
DateTime.IsInPreviousNSeconds(dateTime as any, seconds as number) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous number of seconds, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current second.

- `dateTime`: A `datetime` >, or `datetimezone` value to be evaluated.
- `seconds`: The number of seconds.

Example 1

Determine if the second before the current system time is in the previous two seconds.

Usage

Power Query M

```
DateTime.IsInPreviousNSeconds(DateTime.FixedLocalNow() - #duration(0, 0, 0, 2), 2)
```

Output

```
true
```

DateTime.IsInPreviousSecond

06/19/2025

Syntax

```
DateTime.IsInPreviousSecond(dateTime as any) as nullable logical
```

About

Indicates whether the given datetime value `dateTime` occurs during the previous second, as determined by the current date and time on the system. Note that this function will return false when passed a value that occurs within the current second.

- `dateTime`: A `datetime`, or `datetimezone` value to be evaluated.

Example 1

Determine if the second before the current system time is in the previous second.

Usage

Power Query M

```
DateTime.IsInPreviousSecond(DateTime.FixedLocalNow() - #duration(0, 0, 0, 1))
```

Output

```
true
```

DateTime.LocalNow

07/08/2025

Syntax

```
DateTime.LocalNow() as datetime
```

About

Returns a `datetime` value set to the current date and time on the system.

The value returned by this function depends on whether you're running your query on a local machine or online. For example, if you run your query on a system located in the U.S. Pacific Time zone, Power Query Desktop returns the date and time set on your local machine. However, if you run your query on the cloud, Power Query Online returns UTC time because it's reading the time set on the cloud virtual machines, which are all set to UTC.

Example 1

Invoke this function on a local machine running Power Query Desktop.

Usage

```
Power Query M
```

```
DateTime.LocalNow()
```

Output

The current local date and time.

Example 2

Invoke this function on the cloud running Power Query Online.

Usage

```
Power Query M
```

```
DateTime.LocalNow()
```

Output

The current online (UTC) date and time.

Related content

[Local, fixed, and UTC variants of current time functions](#)

DateTime.Time

06/19/2025

Syntax

```
DateTime.Time(dateTime as any) as nullable time
```

About

Returns the time part of the given datetime value, `dateTime`.

Example 1

Find the time value of `#datetime(2010, 12, 31, 11, 56, 02)`.

Usage

Power Query M

```
DateTime.Time(#datetime(2010, 12, 31, 11, 56, 02))
```

Output

```
#time(11, 56, 2)
```


DateTime.ToRecord

06/19/2025

Syntax

```
DateTime.ToRecord(dateTime as datetime) as record
```

About

Returns a record containing the parts of the given datetime value, `dateTime`.

- `dateTime`: A `datetime` value for from which the record of its parts is to be calculated.

Example 1

Convert the `#datetime(2011, 12, 31, 11, 56, 2)` value into a record containing Date and Time values.

Usage

Power Query M

```
DateTime.ToRecord(#datetime(2011, 12, 31, 11, 56, 2))
```

Output

Power Query M

```
[
    Year = 2011,
    Month = 12,
    Day = 31,
    Hour = 11,
    Minute = 56,
    Second = 2
]
```

DateTime.ToText

06/19/2025

Syntax

```
DateTime.ToText(dateTime as nullable datetime, optional options as any, optional culture as nullable text) as nullable text
```

About

Returns a textual representation of `dateTime`. An optional `record` parameter, `options`, may be provided to specify additional properties. `culture` is only used for legacy workflows. The `record` can contain the following fields:

- Format**: A `text` value indicating the format to use. For more details, go to [Standard date and time format strings](#) and [Custom date and time format strings](#). Omitting this field or providing `null` will result in formatting the date using the default defined by `Culture`.
- Culture**: When `Format` is not null, `Culture` controls some format specifiers. For example, in "en-US" "MMM" is "Jan", "Feb", "Mar", ..., while in "ru-RU" "MMM" is "янв", "фев", "мар", ... When `Format` is null, `Culture` controls the default format to use. When `Culture` is null or omitted, [Culture.Current](#) is used.

To support legacy workflows, `options` and `culture` may also be text values. This has the same behavior as if `options = [Format = options, Culture = culture]`.

Example 1

Convert `#datetime(2010, 12, 31, 01, 30, 25)` into a `text` value. *Result output may vary depending on current culture.*

Usage

Power Query M

```
DateTime.ToText(#datetime(2010, 12, 31, 01, 30, 25))
```

Output

"12/31/2010 1:30:25 AM"

Example 2

Convert using a custom format and the German culture.

Usage

Power Query M

```
DateTime.ToText(#datetime(2010, 12, 30, 2, 4, 50.36973), [Format="dd MMM yyyy  
HH:mm:ss.ffffff", Culture="de-DE"])
```

Output

"30 Dez 2010 02:04:50.369730"

Example 3

Convert using the ISO 8601 pattern.

Usage

Power Query M

```
DateTime.ToText(#datetime(2000, 2, 8, 3, 45, 12), [Format="yyyy-MM-  
dd'T'HH:mm:ss'Z'", Culture="en-US"])
```

Output

"2000-02-08T03:45:12Z"

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

#datetime

06/19/2025

Syntax

```
#datetime(year as number, month as number, day as number, hour as number, minute  
as number, second as number) as datetime
```

About

Creates a datetime value from numbers representing the year, month, day, hour, minute, and (fractional) second. Raises an error if these conditions are not true:

- $1 \leq \text{year} \leq 9999$
- $1 \leq \text{month} \leq 12$
- $1 \leq \text{day} \leq 31$
- $0 \leq \text{hour} \leq 23$
- $0 \leq \text{minute} \leq 59$
- $0 \leq \text{second} < 60$

DateTimeZone functions

Article • 11/13/2024

These functions create and manipulate datetimezone values.

 Expand table

Name	Description
DateTimeZone.FixedLocalNow	Returns a DateTimeZone value set to the current date, time, and timezone offset on the system.
DateTimeZone.FixedUtcNow	Returns the current date and time in UTC (the GMT timezone).
DateTimeZone.From	Returns a datetimezone value from a value.
DateTimeZone.FromFileTime	Returns a DateTimeZone from a number value.
DateTimeZone.FromText	Creates a datetimezone from local, universal, and custom datetimezone formats.
DateTimeZone.LocalNow	Returns a DateTime value set to the current system date and time.
DateTimeZone.RemoveZone	Returns a datetime value with the zone information removed from the input datetimezone value.
DateTimeZone.SwitchZone	Changes the timezone information for the input DateTimeZone.
DateTimeZone.ToLocal	Returns a DateTime value from the local time zone.
DateTimeZone.ToRecord	Returns a record containing parts of a DateTime value.
DateTimeZone.ToText	Returns a text value from a DateTime value.
DateTimeZone.ToUtc	Returns a DateTime value to the Utc time zone.
DateTimeZone.UtcNow	Returns a DateTime value set to the current system date and time in the Utc timezone.
DateTimeZone.ZoneHours	Gets the timezone hour of the value.
DateTimeZone.ZoneMinutes	Gets the timezone minute of the value.
#datetimezone	Creates a datetimezone value from year, month, day, hour, minute, second, offset-hours, and offset-minutes.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

DateTimeZone.FixedLocalNow

07/08/2025

Syntax

```
DateTimeZone.FixedLocalNow() as datetimezone
```

About

Returns a `datetime` value set to the current date and time on the system. The returned value contains timezone information representing the local timezone. This value is fixed and will not change with successive calls, unlike [DateTimeZone.LocalNow](#), which may return different values over the course of execution of an expression.

Related content

[Local, fixed, and UTC variants of current time functions](#)

DateTimeZone.FixedUtcNow

07/08/2025

Syntax

```
DateTimeZone.FixedUtcNow() as datetimezone
```

About

Returns the current date and time in UTC (the GMT timezone). This value is fixed and will not change with successive calls.

Related content

[Local, fixed, and UTC variants of current time functions](#)

DateTimeZone.From

06/19/2025

Syntax

```
DateTimeZone.From(value as any, optional culture as nullable text) as nullable datetimezone
```

About

Returns a `datetimezone` value from the given `value`. An optional `culture` may also be provided (for example, "en-US"). If the given `value` is `null`, **DateTimeZone.From** returns `null`. If the given `value` is `datetimezone`, `value` is returned. Values of the following types can be converted to a `datetimezone` value:

- `text`: A `datetimezone` value from textual representation. Refer to [DateTimeZone.FromText](#) for details.
- `date`: A `datetimezone` with `value` as the date component, `12:00:00 AM` as the time component, and the offset corresponding the local time zone.
- `datetime`: A `datetimezone` with `value` as the datetime and the offset corresponding the local time zone.
- `time`: A `datetimezone` with the date equivalent of the OLE Automation Date of `0` as the date component, `value` as the time component, and the offset corresponding the local time zone.
- `number`: A `datetimezone` with the datetime equivalent to the OLE Automation Date expressed by `value` and the offset corresponding the local time zone.

If `value` is of any other type, an error is returned.

Example 1

Convert `"2020-10-30T01:30:00-08:00"` to a `datetimezone` value.

Usage

Power Query M

```
DateTimeZone.From("2020-10-30T01:30:00-08:00")
```

Output

```
#datetimezone(2020, 10, 30, 01, 30, 00, -8, 00)
```

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

DateTimeZone.FromFileTime

06/19/2025

Syntax

```
DateTimeZone.FromFileTime(fileTime as nullable number) as nullable datetimezone
```

About

Creates a `datetimezone` value from the `fileTime` value and converts it to the local time zone. The filetime is a Windows file time value that represents the number of 100-nanosecond intervals that have elapsed since 12:00 midnight, January 1, 1601 A.D. (C.E.) Coordinated Universal Time (UTC).

Example 1

Convert `129876402529842245` into a `datetimezone` value.

Usage

Power Query M

```
DateTimeZone.FromFileTime(129876402529842245)
```

Output

```
#datetimezone(2012, 7, 24, 14, 50, 52.9842245, -7, 0)
```

DateTimeZone.FromText

06/19/2025

Syntax

```
DateTimeZone.FromText(text as nullable text, optional options as any) as nullable datetimezone
```

About

Creates a `datetimezone` value from a textual representation, `text`. An optional `record` parameter, `options`, may be provided to specify additional properties. The `record` can contain the following fields:

- **Format**: A `text` value indicating the format to use. For more details, go to [Standard date and time format strings](#) and [Custom date and time format strings](#). Omitting this field or providing `null` will result in parsing the date using a best effort.
- **Culture**: When `Format` is not null, `Culture` controls some format specifiers. For example, in "en-US" "MMM" is "Jan", "Feb", "Mar", ..., while in "ru-RU" "MMM" is "ЯНВ", "ФЕВ", "МАР", ... When `Format` is `null`, `Culture` controls the default format to use. When `Culture` is `null` or omitted, [Culture.Current](#) is used.

To support legacy workflows, `options` may also be a text value. This has the same behavior as if `options = [Format = null, Culture = options]`.

Example 1

Convert "2010-12-31T01:30:00-08:00" into a `datetimezone` value.

Usage

Power Query M

```
DateTimeZone.FromText("2010-12-31T01:30:00-08:00")
```

Output

```
#datetimezone(2010, 12, 31, 1, 30, 0, -8, 0)
```

Example 2

Convert using a custom format and the German culture.

Usage

Power Query M

```
DateTimeZone.FromText("30 Dez 2010 02:04:50.369730 +02:00", [Format="dd MMM yyyy  
HH:mm:ss.ffffff zzz", Culture="de-DE"])
```

Output

```
#datetimezone(2010, 12, 30, 2, 4, 50.36973, 2, 0)
```

Example 3

Convert using ISO 8601.

Usage

Power Query M

```
DateTimeZone.FromText("2009-06-15T13:45:30.0000000-07:00", [Format="O",  
Culture="en-US"])
```

Output

```
#datetimezone(2009, 6, 15, 13, 45, 30, -7, 0)
```

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

DateTimeZone.LocalNow

07/08/2025

Syntax

```
DateTimeZone.LocalNow() as datetimezone
```

About

Returns a `datetimezone` value set to the current date and time on the system. The returned value contains timezone information representing the local timezone.

Related content

[Local, fixed, and UTC variants of current time functions](#)

DateTimeZone.RemoveZone

06/19/2025

Syntax

```
DateTimeZone.RemoveZone(dateTimeZone as nullable datetimetype) as nullable  
datetime
```

About

Returns a #datetime value from `dateTimeZone` with timezone information removed.

Example 1

Remove timezone information from the value `#datetimetype(2011, 12, 31, 9, 15, 36, -7, 0)`.

Usage

Power Query M

```
DateTimeZone.RemoveZone(#datetimetype(2011, 12, 31, 9, 15, 36, -7, 0))
```

Output

```
#datetime(2011, 12, 31, 9, 15, 36)
```

DateTimeZone.SwitchZone

06/19/2025

Syntax

```
DateTimeZone.SwitchZone(dateTimeZone as nullable datetimezone, timeZoneHours as number, optional timeZoneMinutes as nullable number) as nullable datetimezone
```

About

Changes timezone information to on the datetimezone value `dateTimeZone` to the new timezone information provided by `timeZoneHours` and optionally `timeZoneMinutes`. If `dateTimeZone` does not have a timezone component, an exception is thrown.

Example 1

Change timezone information for `#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30)` to 8 hours.

Usage

Power Query M

```
DateTimeZone.SwitchZone(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30), 8)
```

Output

```
#datetimezone(2010, 12, 31, 12, 26, 2, 8, 0)
```

Example 2

Change timezone information for `#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30)` to -30 minutes.

Usage

Power Query M

```
DateTimeZone.SwitchZone(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30), 0, -30)
```

Output


```
#datetimezone(2010, 12, 31, 3, 56, 2, 0, -30)
```

DateTimeZone.ToLocal

06/19/2025

Syntax

```
DateTimeZone.ToLocal(dateTimeZone as nullable datetimezone) as nullable datetimezone
```

About

Changes timezone information of the datetimezone value `dateTimeZone` to the local timezone information. If `dateTimeZone` does not have a timezone component, the local timezone information is added.

Example 1

Change timezone information for `#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30)` to local timezone (assuming PST).

Usage

Power Query M

```
DateTimeZone.ToLocal(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30))
```

Output

```
#datetimezone(2010, 12, 31, 12, 26, 2, -8, 0)
```

DateTimeZone.ToRecord

06/19/2025

Syntax

```
DateTimeZone.ToRecord(dateTimeZone as datetimezone) as record
```

About

Returns a record containing the parts of the given datetimezone value, **dateTimeZone**.

- **dateTimeZone**: A **datetimezone** value for from which the record of its parts is to be calculated.

Example 1

Convert the **#datetimezone(2011, 12, 31, 11, 56, 2, 8, 0)** value into a record containing Date, Time, and Zone values.

Usage

Power Query M

```
DateTimeZone.ToRecord(#datetimezone(2011, 12, 31, 11, 56, 2, 8, 0))
```

Output

Power Query M

```
[
    Year = 2011,
    Month = 12,
    Day = 31,
    Hour = 11,
    Minute = 56,
    Second = 2,
    ZoneHours = 8,
    ZoneMinutes = 0
]
```

DateTimeZone.ToText

06/19/2025

Syntax

```
DateTimeZone.ToText(dateTimeZone as nullable datetimezone, optional options as any, optional culture as nullable text) as nullable text
```

About

Returns a textual representation of `dateTimeZone`. An optional `record` parameter, `options`, may be provided to specify additional properties. `culture` is only used for legacy workflows. The `record` can contain the following fields:

- **Format**: A `text` value indicating the format to use. For more details, go to * [Standard date and time format strings](#) and [Custom date and time format strings](#). Omitting this field or providing `null` will result in formatting the date using the default defined by `Culture`.
- **Culture**: When `Format` is not null, `Culture` controls some format specifiers. For example, in "en-US" "MMM" is "Jan", "Feb", "Mar", ..., while in "ru-RU" "MMM" is "янв", "фев", "мар", ... When `Format` is null, `Culture` controls the default format to use. When `Culture` is null or omitted, [Culture.Current](#) is used.

To support legacy workflows, `options` and `culture` may also be text values. This has the same behavior as if `options = [Format = options, Culture = culture]`.

Example 1

Convert `#datetimezone(2010, 12, 31, 01, 30, 25, 2, 0)` into a `text` value. *Result output may vary depending on current culture.*

Usage

Power Query M

```
DateTimeZone.ToText(#datetimezone(2010, 12, 31, 01, 30, 25, 2, 0))
```

Output

"12/31/2010 1:30:25 AM +02:00"

Example 2

Convert using a custom format and the German culture.

Usage

Power Query M

```
DateTimeZone.ToText(#datetimezone(2010, 12, 30, 2, 4, 50.36973, -8,0), [Format="dd  
MMM yyyy HH:mm:ss.ffffff zzz", Culture="de-DE"])
```

Output

"30 Dez 2010 02:04:50.369730 -08:00"

Example 3

Convert using the ISO 8601 pattern.

Usage

Power Query M

```
DateTimeZone.ToText(#datetimezone(2000, 2, 8, 3, 45, 12, 2, 0),[Format="O",  
Culture="en-US"])
```

Output

"2000-02-08T03:45:12.0000000+02:00"

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

DateTimeZone.ToUtc

06/19/2025

Syntax

```
DateTimeZone.ToUtc(dateTimeZone as nullable datetimezone) as nullable datetimezone
```

About

Changes timezone information of the datetime value `dateTimeZone` to the UTC or Universal Time timezone information. If `dateTimeZone` does not have a timezone component, the UTC timezone information is added.

Example 1

Change timezone information for `#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30)` to UTC timezone.

Usage

Power Query M

```
DateTimeZone.ToUtc(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30))
```

Output

```
#datetimezone(2010, 12, 31, 4, 26, 2, 0, 0)
```

DateTimeZone.UtcNow

07/08/2025

Syntax

```
DateTimeZone.UtcNow() as datetimezone
```

About

Returns the current date and time in UTC (the GMT timezone).

Example 1

Get the current date & time in UTC.

Usage

Power Query M

```
DateTimeZone.UtcNow()
```

Output

```
#datetimezone(2011, 8, 16, 23, 34, 37.745, 0, 0)
```

Related content

[Local, fixed, and UTC variants of current time functions](#)

DateTimeZone.ZoneHours

06/19/2025

Syntax

```
DateTimeZone.ZoneHours(dateTimeZone as nullable datetimezone) as nullable number
```

About

Returns the time zone hour component of a `datetimezone` value.

- `dateTimeZone`: A `datetimezone` value from which the time zone hour component is extracted. If `dateTimeZone` is `null`, the function returns `null`.

Example 1

Get the time zone hours component of the specified `datetimezone` value.

Usage

Power Query M

```
DateTimeZone.ZoneHours(#datetimezone(2024, 4, 28, 13, 24, 22, 7, 30))
```

Output

7

DateTimeZone.ZoneMinutes

06/19/2025

Syntax

```
DateTimeZone.ZoneMinutes(dateTimeZone as nullable datetimezone) as nullable number
```

About

Returns the time zone minutes component of a `datetimezone` value.

- `dateTimeZone`: a `datetimezone` value from which the time zone minutes component is extracted. If `dateTimeZone` is `null`, the function returns `null`.

Example 1

Get the time zone minutes component of the specified `datetimezone` value.

Usage

Power Query M

```
DateTimeZone.ZoneMinutes(#datetimezone(2024, 4, 28, 13, 24, 22, 7, 30))
```

Output

30

#datetimezone

06/19/2025

Syntax

```
#datetimezone(year as number, month as number, day as number, hour as number,  
minute as number, second as number, offsetHours as number, offsetMinutes as  
number) as datetimezone
```

About

Creates a datetimezone value from numbers representing the year, month, day, hour, minute, (fractional) second, (fractional) offset-hours, and offset-minutes. Raises an error if these conditions are not true:

- $1 \leq \text{year} \leq 9999$
- $1 \leq \text{month} \leq 12$
- $1 \leq \text{day} \leq 31$
- $0 \leq \text{hour} \leq 23$
- $0 \leq \text{minute} \leq 59$
- $0 \leq \text{second} < 60$
- $-14 \leq \text{offset-hours} + \text{offset-minutes} / 60 \leq 14$

Duration functions

Article • 01/29/2025

These functions create and manipulate duration values.

 Expand table

Name	Description
Duration.Days	Returns the days portion of a duration.
Duration.From	Creates a duration from the given value.
Duration.FromText	Returns a duration value from a text value.
Duration.Hours	Returns the hours portion of a duration.
Duration.Minutes	Returns the minutes portion of a duration.
Duration.Seconds	Returns the seconds portion of a duration.
Duration.ToRecord	Returns a record containing the parts of the duration.
Duration.TotalDays	Returns the total days this duration spans.
Duration.TotalHours	Returns the total hours this duration spans.
Duration.TotalMinutes	Returns the total minutes this duration spans.
Duration.TotalSeconds	Returns the total seconds this duration spans.
Duration.ToText	Returns the text of the form "d.h:m:s".
#duration	Creates a duration value from days, hours, minutes, and seconds.

Feedback

Was this page helpful?  

[Provide product feedback](#) | [Ask the community](#) 

Duration.Days

06/19/2025

Syntax

```
Duration.Days(duration as nullable duration) as nullable number
```

About

Returns the days portion of `duration`.

Example 1

Extract the number of days between two dates.

Usage

Power Query M

```
Duration.Days(#date(2022, 3, 4) - #date(2022, 2, 25))
```

Output

7

Duration.From

06/19/2025

Syntax

```
Duration.From(value as any) as nullable duration
```

About

Returns the duration value from the given value.

- **value**: The value from which the duration is derived. If the given **value** is **null**, this function returns **null**. If the given **value** is a **duration**, **value** is returned. Values of the following types can be converted to a **duration** value:
 - **text**: A **duration** value from textual elapsed time forms (d.h:m:s). Refer to [Duration.FromText](#) for details.
 - **number**: A **duration** equivalent to the number of whole and fractional days expressed by **value**.

If **value** is of any other type, an error is returned.

Example 1

Convert 2.525 into a **duration** value.

Usage

```
Power Query M
```

```
Duration.From(2.525)
```

Output

```
#duration(2, 12, 36, 0)
```

Example 2

Convert the text value "2.05:55:20.34567" into a **duration** value.

Usage

Power Query M

```
Duration.From("2.05:55:20.34567")
```

Output

```
#duration(2, 5, 55, 20.3456700)
```

Duration.FromText

06/19/2025

Syntax

```
Duration.FromText(text as nullable text) as nullable duration
```

About

Returns a duration value from the specified text, `text`. The following formats can be parsed by this function:

- (-)hh:mm(:ss(.ff))
- (-)ddd(.hh:mm(:ss(.ff)))

(All ranges are inclusive)

- ddd: Number of days.
- hh: Number of hours, between 0 and 23.
- mm: Number of minutes, between 0 and 59.
- ss: Number of seconds, between 0 and 59.
- ff: Fraction of seconds, between 0 and 9999999.

Example 1

Convert `"2.05:55:20"` into a `duration` value.

Usage

```
Power Query M
```

```
Duration.FromText("2.05:55:20")
```

Output

```
#duration(2, 5, 55, 20)
```

Duration.Hours

06/19/2025

Syntax

```
Duration.Hours(duration as nullable duration) as nullable number
```

About

Returns the hours portion of `duration`.

Example 1

Extract the hours from a duration value.

Usage

Power Query M

```
Duration.Hours(#duration(5, 4, 3, 2))
```

Output

4

Duration.Minutes

06/19/2025

Syntax

```
Duration.Minutes(duration as nullable duration) as nullable number
```

About

Returns the minutes portion of `duration`.

Example 1

Extract the minutes from a duration value.

Usage

Power Query M

```
Duration.Minutes(#duration(5, 4, 3, 2))
```

Output

3

Duration.Seconds

06/19/2025

Syntax

```
Duration.Seconds(duration as nullable duration) as nullable number
```

About

Returns the seconds portion of `duration`.

Example 1

Extract the seconds from a duration value.

Usage

Power Query M

```
Duration.Seconds(#duration(5, 4, 3, 2))
```

Output

2

Duration.ToRecord

06/19/2025

Syntax

```
Duration.ToRecord(duration as duration) as record
```

About

Returns a record containing the parts the duration value, `duration`.

- `duration`: A `duration` from which the record is created.

Example 1

Convert `#duration(2, 5, 55, 20)` into a record of its parts including days, hours, minutes, and seconds if applicable.

Usage

Power Query M

```
Duration.ToRecord(#duration(2, 5, 55, 20))
```

Output

Power Query M

```
[
    Days = 2,
    Hours = 5,
    Minutes = 55,
    Seconds = 20
]
```

Duration.TotalDays

06/19/2025

Syntax

```
Duration.TotalDays(duration as nullable duration) as nullable number
```

About

Returns the total days spanned by `duration`.

Example 1

Find the total days spanned by a duration value.

Usage

Power Query M

```
Duration.TotalDays(#duration(5, 4, 3, 2))
```

Output

```
5.1687731481481478
```

Duration.TotalHours

06/19/2025

Syntax

```
Duration.TotalHours(duration as nullable duration) as nullable number
```

About

Returns the total hours spanned by `duration`.

Example 1

Find the total hours spanned by a duration value.

Usage

Power Query M

```
Duration.TotalHours(#duration(5, 4, 3, 2))
```

Output

```
124.05055555555555
```

Duration.TotalMinutes

06/19/2025

Syntax

```
Duration.TotalMinutes(duration as nullable duration) as nullable number
```

About

Returns the total minutes spanned by `duration`.

Example 1

Find the total minutes spanned by a duration value.

Usage

Power Query M

```
Duration.TotalMinutes(#duration(5, 4, 3, 2))
```

Output

```
7443.0333333333338
```

Duration.TotalSeconds

06/19/2025

Syntax

```
Duration.TotalSeconds(duration as nullable duration) as nullable number
```

About

Returns the total seconds spanned by `duration`.

Example 1

Find the total seconds spanned by a duration value.

Usage

Power Query M

```
Duration.TotalSeconds(#duration(5, 4, 3, 2))
```

Output

446582

Duration.ToText

06/19/2025

Syntax

```
Duration.ToText(duration as nullable duration, optional format as nullable text)  
as nullable text
```

About

Returns a textual representation in the form "day.hour:mins:sec" of the given duration value, `duration`.

- `duration`: A `duration` from which the textual representation is calculated.
- `format`: *[Optional]* Deprecated, will throw an error if not null.

Example 1

Convert `#duration(2, 5, 55, 20)` into a text value.

Usage

Power Query M

```
Duration.ToText(#duration(2, 5, 55, 20))
```

Output

```
"2.05:55:20"
```


#duration

06/19/2025

Syntax

```
#duration(days as number, hours as number, minutes as number, seconds as number)  
as duration
```

About

Creates a duration value from numbers representing days, hours, minutes, and (fractional) seconds.

Error handling functions

Article • 07/15/2024

These functions can be used to trace or construct errors.

 Expand table

Name	Description
Diagnostics.ActivityId	Returns an opaque identifier for the currently-running evaluation.
Diagnostics.CorrelationId	Returns an opaque identifier to correlate incoming requests with outgoing ones.
Diagnostics.Trace	Writes a trace message, if tracing is enabled, and returns value.
Error.Record	Returns an error record from the provided text values for reason, message, detail, and error code.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Diagnostics.ActivityId

06/19/2025

Syntax

```
Diagnostics.ActivityId() as nullable text
```

About

Returns an opaque identifier for the currently-running evaluation.

Diagnostics.CorrelationId

06/19/2025

Syntax

```
Diagnostics.CorrelationId() as nullable text
```

About

Returns an opaque identifier to correlate incoming requests with outgoing ones.

Diagnostics.Trace

06/19/2025

Syntax

```
Diagnostics.Trace(traceLevel as number, message as anynonnull, value as any,  
optional delayed as nullable logical) as any
```

About

Writes a trace `message`, if tracing is enabled, and returns `value`. An optional parameter `delayed` specifies whether to delay the evaluation of `value` until the message is traced. `traceLevel` can take one of the following values:

- [TraceLevel.Critical](#)
- [TraceLevel.Error](#)
- [TraceLevel.Warning](#)
- [TraceLevel.Information](#)
- [TraceLevel.Verbose](#)

Example 1

Trace the message before invoking Text.From function and return the result.

Usage

Power Query M

```
Diagnostics.Trace(TraceLevel.Information, "TextValueFromNumber", () =>  
Text.From(123), true)
```

Output

```
"123"
```

Error.Record

06/19/2025

Syntax

```
Error.Record(reason as text, optional message as nullable text, optional detail as any, optional parameters as nullable list, optional errorCode as nullable text) as record
```

About

Returns an error record from the provided text values for reason, message, detail, and error code.

Expression functions

Article • 09/21/2022

These functions allow the construction and evaluation of M code.

Name	Description
Expression.Constant	Returns the M source code representation of a constant value.
Expression.Evaluate	Returns the result of evaluating an M expression.
Expression.Identifier	Returns the M source code representation of an identifier.

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Expression.Constant

06/19/2025

Syntax

```
Expression.Constant(value as any) as text
```

About

Returns the M source code representation of a constant value.

Example 1

Get the M source code representation of a number value.

Usage

Power Query M

```
Expression.Constant(123)
```

Output

```
"123"
```

Example 2

Get the M source code representation of a date value.

Usage

Power Query M

```
Expression.Constant(#date(2035, 01, 02))
```

Output

```
"#date(2035, 1, 2)"
```


Example 3

Get the M source code representation of a text value.

Usage

Power Query M

```
Expression.Constant("abc")
```

Output

```
""""abc""""
```

Expression.Evaluate

06/19/2025

Syntax

```
Expression.Evaluate(document as text, optional environment as nullable record) as any
```

About

Returns the result of evaluating an M expression `document`, with the available identifiers that can be referenced defined by `environment`.

Example 1

Evaluate a simple sum.

Usage

```
Power Query M
```

```
Expression.Evaluate("1 + 1")
```

Output

```
2
```

Example 2

Evaluate a more complex sum.

Usage

```
Power Query M
```

```
Expression.Evaluate("List.Sum({1, 2, 3})", [List.Sum = List.Sum])
```

Output

Example 3

Evaluate the concatenation of a text value with an identifier.

Usage

Power Query M

```
Expression.Evaluate(Expression.Constant("""abc") & " & " &  
Expression.Identifier("x"), [x = "def"""])
```

Output

```
"" abcdef ""
```

Expression.Identifier

06/19/2025

Syntax

```
Expression.Identifier(name as text) as text
```

About

Returns the M source code representation of an identifier `name`.

Example 1

Get the M source code representation of an identifier.

Usage

Power Query M

```
Expression.Identifier("MyIdentifier")
```

Output

```
"MyIdentifier"
```

Example 2

Get the M source code representation of an identifier that contains a space.

Usage

Power Query M

```
Expression.Identifier("My Identifier")
```

Output

```
"#"My Identifier"'"
```

Function values

Article • 10/25/2023

These functions create and invoke other M functions.

Name	Description
Function.From	Takes a unary function <code>function</code> and creates a new function with the type <code>functionType</code> that constructs a list out of its arguments and passes it to <code>function</code> .
Function.Invoke	Invokes the given function using the specified and returns the result.
Function.InvokeAfter	Returns the result of invoking function after duration delay has passed.
Function.InvokeWithErrorContext	This function is intended for internal use only.
Function.IsDataSource	Returns whether or not function is considered a data source.
Function.ScalarVector	Returns a scalar function of type <code>scalarFunctionType</code> that invokes <code>vectorFunction</code> with a single row of arguments and returns its single output.

Feedback

Was this page helpful?

 Yes

 No

Function.From

06/19/2025

Syntax

```
Function.From(functionType as type, function as function) as function
```

About

Takes a unary function `function` and creates a new function with the type `functionType` that constructs a list out of its arguments and passes it to `function`.

Example 1

Converts List.Sum into a two-argument function whose arguments are added together.

Usage

Power Query M

```
Function.From(type function (a as number, b as number) as number, List.Sum)(2, 1)
```

Output

3

Example 2

Converts a function taking a list into a two-argument function.

Usage

Power Query M

```
Function.From(type function (a as text, b as text) as text, (list) => list{0} & list{1})("2", "1")
```

Output

Function.Invoke

06/19/2025

Syntax

```
Function.Invoke(function as function, args as list) as any
```

About

Invokes the given function using the specified list of arguments and returns the result.

Example 1

Invokes [Record.FieldNames](#) with one argument [A=1,B=2]

Usage

Power Query M

```
Function.Invoke(Record.FieldNames, {[A = 1, B = 2]})
```

Output

```
{"A", "B"}
```


Function.InvokeAfter

06/19/2025

Syntax

```
Function.InvokeAfter(function as function, delay as duration) as any
```

About

Returns the result of invoking `function` after duration `delay` has passed.

Function.InvokeWithErrorContext

06/19/2025

Syntax

```
Function.InvokeWithErrorContext(function as function, context as text) as any
```

About

This function is intended for internal use only.

Function.IsDataSource

06/19/2025

Syntax

```
Function.IsDataSource(function as function) as logical
```

About

Returns whether or not `function` is considered a data source.

Function.ScalarVector

06/19/2025

Syntax

```
Function.ScalarVector(scalarFunctionType as type, vectorFunction as function) as function
```

About

Returns a scalar function of type `scalarFunctionType` that invokes `vectorFunction` with a single row of arguments and returns its single output. Additionally, when the scalar function is repeatedly applied for each row of a table of inputs, such as in `Table.AddColumn`, instead `vectorFunction` will be applied once for all inputs.

`vectorFunction` will be passed a table whose columns match in name and position the parameters of `scalarFunctionType`. Each row of this table contains the arguments for one call to the scalar function, with the columns corresponding to the parameters of `scalarFunctionType`.

`vectorFunction` must return a list of the same length as the input table, whose item at each position must be the same result as evaluating the scalar function on the input row of the same position.

The input table is expected to be streamed in, so `vectorFunction` is expected to stream its output as input comes in, only working with one chunk of input at a time. In particular, `vectorFunction` must not enumerate its input table more than once.

Lines functions

Article • 08/04/2022

These functions convert lists of text to and from binary and single text values.

Name	Description
Lines.FromBinary	Converts a binary value to a list of text values split at lines breaks.
Lines.FromText	Converts a text value to a list of text values split at lines breaks.
Lines.ToBinary	Converts a list of text into a binary value using the specified encoding and lineSeparator. The specified lineSeparator is appended to each line. If not specified then the carriage return and line feed characters are used.
Lines.ToText	Converts a list of text into a single text. The specified lineSeparator is appended to each line. If not specified then the carriage return and line feed characters are used.

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Lines.FromBinary

06/19/2025

Syntax

```
Lines.FromBinary(binary as binary, optional quoteStyle as any, optional  
includeLineSeparators as nullable logical, optional encoding as nullable number)  
as list
```

About

Converts a binary value to a list of text values split at line breaks.

- `binary`: The binary value to convert to the list.
- `quoteStyle`: Specifies how line breaks are handled. The value of `quoteStyle` can be `null`. The default value is [QuoteStyle.None](#).
- `includeLineSeparators`: Specifies whether to include the line break characters in the text. The value of `includeLineSeparators` can be `null`. The default value is `false`.
- `encoding`: Specifies the text encoding of the binary value. The value of `encoding` can be `null`. The default value is `65001` (UTF-8).

If a record is specified for `quoteStyle` (and `includeLineSeparators` and `encoding` are `null`), the following record fields can be provided:

- `QuoteStyle`: Specifies how quoted line breaks are handled.
 - [QuoteStyle.Csv](#): Quoted line breaks are treated as part of the data, not as the end of the current row.
 - [QuoteStyle.None](#): All line breaks are treated as the end of the current row, even when they occur inside a quoted value. This value is the default if the `CsvStyle` option isn't specified.
- `CsvStyle`: Specifies how quotes are handled. Should not be used with `QuoteStyle.None`.
 - [CsvStyle.QuoteAfterDelimiter](#): Quotes in a field are only significant immediately following the ``Delimiter`.
 - [CsvStyle.QuoteAlways](#): Quotes in a field are always significant, regardless of where they appear.
- `Delimiter`: A single character delimiter. Should be used only with `CsvStyle.QuoteAfterDelimiter`.

- `IncludeLineSeparators`: Specifies whether to include the line break characters in the text. The default value is `false`.
- `Encoding`: The text encoding of the binary value. The default value is `65001` (UTF-8).

Lines.FromText

06/19/2025

Syntax

```
Lines.FromText(text as text, optional quoteStyle as any, optional  
includeLineSeparators as nullable logical) as list
```

About

Converts a text value to a list of text values split at line breaks.

- `text`: The text value to convert to the list of text values.
- `quoteStyle`: Specifies how line breaks are handled. The value of `quoteStyle` can be `null`. The default value is [QuoteStyle.None](#).
- `includeLineSeparators`: Specifies whether to include the line break characters in the text. The value of `includeLineSeparators` can be `null`. The default value is `false`.

If a record is specified for `quoteStyle` (and `includeLineSeparators` is `null`), the following record fields can be provided:

- `QuoteStyle`: Specifies how quoted line breaks are handled.
 - [QuoteStyle.Csv](#): Quoted line breaks are treated as part of the data, not as the end of the current row.
 - [QuoteStyle.None](#): All line breaks are treated as the end of the current row, even when they occur inside a quoted value. This value is the default if the `CsvStyle` option isn't specified.
- `CsvStyle`: Specifies how quotes are handled. Should not be used with `QuoteStyle.None`.
 - [CsvStyle.QuoteAfterDelimiter](#): Quotes in a field are only significant immediately following the `Delimiter`.
 - [CsvStyle.QuoteAlways](#): Quotes in a field are always significant, regardless of where they appear.
- `Delimiter`: A single character delimiter. Should be used only with `CsvStyle.QuoteAfterDelimiter`.
- `IncludeLineSeparators`: Specifies whether to include the line break characters in the text. The default value is `false`.

Lines.ToBinary

06/19/2025

Syntax

```
Lines.ToBinary(lines as list, optional lineSeparator as nullable text, optional encoding as nullable number, optional includeByteOrderMark as nullable logical) as binary
```

About

Converts a list of text into a binary value using the specified encoding and lineSeparator. The specified lineSeparator is appended to each line. If not specified then the carriage return and line feed characters are used.

Lines.ToText

06/19/2025

Syntax

```
Lines.ToText(lines as list, optional lineSeparator as nullable text) as text
```

About

Converts a list of text into a single text. The specified lineSeparator is appended to each line. If not specified then the carriage return and line feed characters are used.

List functions

Article • 01/28/2025

These functions create and manipulate list values.

Information

 Expand table

Name	Description
List.Count	Returns the number of items in a list.
List.IsEmpty	Returns <code>true</code> if the list is empty.
List.NonNullCount	Returns the number of non-null items in the list.

Selection

 Expand table

Name	Description
List.Alternate	Returns a list comprised of all the odd numbered offset elements in a list.
List.Buffer	Buffers a list.
List.Distinct	Returns a list of values with duplicates removed.
List.FindText	Returns a list of values (including record fields) that contain the specified text.
List.First	Returns the first value of the list or the specified default if empty.
List.FirstN	Returns the first set of items in the list by specifying how many items to return or a qualifying condition.
List.InsertRange	Inserts values into a list at the given index.
List.IsDistinct	Indicates whether there are duplicates in the list.
List.Last	Returns the last value of the list or the specified default if empty.
List.LastN	Returns the last value in the list. Can optionally specify how many values to return or a qualifying condition.

Name	Description
List.MatchesAll	Returns <code>true</code> if the condition function is satisfied by all values in the list.
List.MatchesAny	Returns <code>true</code> if the condition function is satisfied by any value.
List.Positions	Returns a list of offsets for the input.
List.Range	Returns a subset of the list beginning at an offset.
List.Select	Returns a list of values that match the condition.
List.Single	Returns the one list item for a list of length one, otherwise throws an exception.
List.SingleOrDefault	Returns the one list item for a list of length one and the default value for an empty list.
List.Skip	Returns a list that skips the specified number of elements at the beginning of the list.

Transformation functions

[Expand table](#)

Name	Description
List.Accumulate	AAccumulates a summary value from the items in the list.
List.Combine	Returns a single list by combining multiple lists.
List.ConformToPageReader	This function is intended for internal use only.
List.RemoveFirstN	Returns a list that skips the specified number of elements at the beginning of the list.
List.RemoveItems	Removes items from the first list that are present in the second list.
List.RemoveLastN	Returns a list that removes the specified number of elements from the end of the list.
List.RemoveMatchingItems	Removes all occurrences of the input values.
List.RemoveNulls	Removes all <code>null</code> values from the specified list.
List.RemoveRange	Removes count number of values starting at the specified position.
List.Repeat	Returns a list that is <code>count</code> repetitions of the original list.

Name	Description
List.ReplaceMatchingItems	Replaces occurrences of existing values in the list with new values that match the condition.
List.ReplaceRange	Replaces <code>count</code> number of values starting at <code>position</code> with the replacement values.
List.ReplaceValue	Searches a list for the specified value and replaces it.
List.Reverse	Reverses the order of values in the list.
List.Split	Splits the specified list into a list of lists using the specified page size.
List.Transform	Returns a new list of values computed from this list.
List.TransformMany	Returns a list whose elements are transformed from the input list using specified functions.
List.Zip	Returns a list of lists by combining items at the same position in multiple lists.

Membership functions

Since all values can be tested for equality, these functions can operate over heterogeneous lists.

 Expand table

Name	Description
List.AllTrue	Returns <code>true</code> if all expressions are true.
List.AnyTrue	Returns true if any expression is true.
List.Contains	Indicates whether the list contains the value.
List.ContainsAll	Indicates where a list includes all the values in another list.
List.ContainsAny	Indicates where a list includes any of the values in another list.
List.PositionOf	Returns the offset(s) of a value in a list.
List.PositionOfAny	Returns the first offset of a value in a list.

Set operations

[Expand table](#)

Name	Description
List.Difference	Returns the difference of the two given lists.
List.Intersect	Returns the intersection of the list values found in the input.
List.Union	Returns the union of the list values found in the input.

Ordering

Ordering functions perform comparisons. All values that are compared must be comparable with each other. This means they must all come from the same datatype (or include null, which always compares smallest). Otherwise, an `Expression.Error` is thrown.

Comparable data types include:

- Number
- Duration
- DateTime
- Text
- Logical
- Null

[Expand table](#)

Name	Description
List.Max	Returns the maximum value or the default value for an empty list.
List.MaxN	Returns the maximum value(s) in the list. The number of values to return or a filtering condition must be specified.
List.Median	Returns the median value in the list.
List.Min	Returns the minimum value or the default value for an empty list.
List.MinN	Returns the minimum value(s) in the list. The number of values to return or a filtering condition may be specified.
List.Sort	Sorts a list of data according to the criteria specified.
List.Percentile	Returns one or more sample percentiles corresponding to the given probabilities.

Averages

These functions operate over homogeneous lists of Numbers, DateTimes, and Durations.

[Expand table](#)

Name	Description
List.Average	Returns the average of the values. Works with number, date, datetime, datetimezone and duration values.
List.Mode	Returns the most frequent value in the list.
List.Modes	Returns a list of the most frequent values in the list.
List.StandardDeviation	Returns a sample based estimate of the standard deviation. This function performs a sample based estimate. The result is a number for numbers, and a duration for DateTimes and Durations.

Addition

These functions work over homogeneous lists of Numbers or Durations.

[Expand table](#)

Name	Description
List.Sum	Returns the sum of the items in the list.

Numerics

These functions only work over numbers.

[Expand table](#)

Name	Description
List.Covariance	Returns the covariance between the two lists of numbers.
List.Product	Returns the product of the numbers in the list.

Generators

These functions generate list of values.

Name	Description
List.Dates	Generates a list of <code>date</code> values given an initial value, count, and incremental duration value.
List.DateTimes	Generates a list of <code>datetime</code> values given an initial value, count, and incremental duration value.
List.DateTimeZones	Generates a list of <code>datetimezone</code> values given an initial value, count, and incremental duration value.
List.Durations	Generates a list of <code>duration</code> values given an initial value, count, and incremental duration value.
List.Generate	Generates a list of values.
List.Numbers	Returns a list of numbers given an initial value, count, and optional increment value.
List.Random	Returns a list of random numbers.
List.Times	Generates a list of <code>time</code> values given an initial value, count, and incremental duration value.

Parameter values

Occurrence specification

- [Occurrence.First](#) = 0;
- [Occurrence.Last](#) = 1;
- [Occurrence.All](#) = 2;

Sort order

- [Order.Ascending](#) = 0;
- [Order.Descending](#) = 1;

Equation criteria

Equation criteria for list values can be specified as either:

- A function value that is either:
 - A key selector that determines the value in the list to apply the equality criteria.

- A comparer function that is used to specify the kind of comparison to apply.
Built in comparer functions can be specified—go to [Comparer functions](#).
- A list value that has:
 - Exactly two items.
 - The first element is the key selector as specified above.
 - The second element is a comparer as specified above.

For more information and examples, go to [List.Distinct](#).

Comparison criteria

Comparison criterion can be provided as either of the following values:

- A number value to specify a sort order. For more information, go to [Sort order](#).
- To compute a key to be used for sorting, a function of one argument can be used.
- To both select a key and control order, comparison criterion can be a list containing the key and order.
- To completely control the comparison, a function of two arguments can be used that returns -1, 0, or 1 given the relationship between the left and right inputs.
[Value.Compare](#) is a method that can be used to delegate this logic.

For more information and examples, go to [List.Sort](#).

Replacement operations

Replacement operations are specified by a list value. Each item of this list must be:

- A list value of exactly two items.
- First item is the old value in the list, to be replaced.
- Second item is the new value, which should replace all occurrences of the old value in the list.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

List.Accumulate

06/19/2025

Syntax

```
List.Accumulate(list as list, seed as any, accumulator as function) as any
```

About

Accumulates a summary value from the items in the list `list`, using `accumulator`. An optional seed parameter, `seed`, may be set.

Example 1

Accumulates the summary value from the items in the list {1, 2, 3, 4, 5} using ((state, current) => state + current).

Usage

Power Query M

```
List.Accumulate({1, 2, 3, 4, 5}, 0, (state, current) => state + current)
```

Output

15

List.AllTrue

06/19/2025

Syntax

```
List.AllTrue(list as list) as logical
```

About

Returns true if all expressions in the list `list` are true.

Example 1

Determine if all the expressions in the list {true, true, 2 > 0} are true.

Usage

Power Query M

```
List.AllTrue({true, true, 2 > 0})
```

Output

true

Example 2

Determine if all the expressions in the list {true, true, 2 < 0} are true.

Usage

Power Query M

```
List.AllTrue({true, false, 2 < 0})
```

Output

false

List.Alternate

06/19/2025

Syntax

```
List.Alternate(list as list, count as number, optional repeatInterval as nullable number, optional offset as nullable number) as list
```

About

Returns a list comprised of all the odd numbered offset elements in a list. Alternates between taking and skipping values from the list `list` depending on the parameters.

- `count`: Specifies number of values that are skipped each time.
- `repeatInterval`: An optional repeat interval to indicate how many values are added in between the skipped values.
- `offset`: An option offset parameter to begin skipping the values at the initial offset.

Example 1

Create a list from {1..10} that skips the first number.

Usage

Power Query M

```
List.Alternate({1..10}, 1)
```

Output

```
{2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Example 2

Create a list from {1..10} that skips every other number.

Usage

Power Query M

```
List.Alternate({1..10}, 1, 1)
```

Output

```
{2, 4, 6, 8, 10}
```

Example 3

Create a list from {1..10} that starts at 1 and skips every other number.

Usage

Power Query M

```
List.Alternate({1..10}, 1, 1, 1)
```

Output

```
{1, 3, 5, 7, 9}
```

Example 4

Create a list from {1..10} that starts at 1, skips one value, keeps two values, and so on.

Usage

Power Query M

```
List.Alternate({1..10}, 1, 2, 1)
```

Output

```
{1, 3, 4, 6, 7, 9, 10}
```

List.AnyTrue

06/19/2025

Syntax

```
List.AnyTrue(list as list) as logical
```

About

Returns true if any expression in the list `list` is true.

Example 1

Determine if any of the expressions in the list {true, false, 2 > 0} are true.

Usage

Power Query M

```
List.AnyTrue({true, false, 2>0})
```

Output

true

Example 2

Determine if any of the expressions in the list {2 = 0, false, 2 < 0} are true.

Usage

Power Query M

```
List.AnyTrue({2 = 0, false, 2 < 0})
```

Output

false

List.Average

06/19/2025

Syntax

```
List.Average(list as list, optional precision as nullable number) as any
```

About

Returns the average value for the items in the list, `list`. The result is given in the same datatype as the values in the list. Only works with number, date, time, datetime, datetimezone and duration values. If the list is empty null is returned.

Example 1

Find the average of the list of numbers, `{3, 4, 6}`.

Usage

Power Query M

```
List.Average({3, 4, 6})
```

Output

```
4.333333333333333
```

Example 2

Find the average of the date values January 1, 2011, January 2, 2011 and January 3, 2011.

Usage

Power Query M

```
List.Average({#date(2011, 1, 1), #date(2011, 1, 2), #date(2011, 1, 3)})
```

Output

```
#date(2011, 1, 2)
```


List.Buffer

06/19/2025

Syntax

```
List.Buffer(list as list) as list
```

About

Buffers the list `list` in memory. The result of this call is a stable list.

Example 1

Create a stable copy of the list {1..10}.

Usage

Power Query M

```
List.Buffer({1..10})
```

Output

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

List.Combine

06/19/2025

Syntax

```
List.Combine(lists as list) as list
```

About

Takes a list of lists, `lists`, and merges them into a single new list.

Example 1

Combine the two simple lists {1, 2} and {3, 4}.

Usage

Power Query M

```
List.Combine({{1, 2}, {3, 4}})
```

Output

Power Query M

```
{  
    1,  
    2,  
    3,  
    4  
}
```

Example 2

Combine the two lists, {1, 2} and {3, {4, 5}}, one of which contains a nested list.

Usage

Power Query M

```
List.Combine({{1, 2}, {3, {4, 5}}})
```

Output

Power Query M

```
{  
    1,  
    2,  
    3,  
    {4, 5}  
}
```

List.ConformToPageReader

06/19/2025

Syntax

```
List.ConformToPageReader(list as list, optional options as nullable record) as table
```

About

This function is intended for internal use only.

List.Contains

06/20/2025

Syntax

```
List.Contains(list as list, value as any, optional equationCriteria as any) as logical
```

About

Indicates whether the list contains the specified value. Returns `true` if the value is found in the list, `false` otherwise.

- `list`: The list to search.
- `value`: The value to search for in the list.
- `equationCriteria`: (Optional) The comparer used to determine if the two values are equal.

Example 1

Determine if the list {1, 2, 3, 4, 5} contains 3.

Usage

Power Query M

```
List.Contains({1, 2, 3, 4, 5}, 3)
```

Output

```
true
```

Example 2

Determine if the list {1, 2, 3, 4, 5} contains 6.

Usage

Power Query M

```
List.Contains({1, 2, 3, 4, 5}, 6)
```

Output

```
false
```

Example 3

Ignoring case, determine if the list contains "rhubarb".

Usage

Power Query M

```
List.Contains({"Pears", "Bananas", "Rhubarb", "Peaches"},  
    "rhubarb",  
    Comparer.OrdinalIgnoreCase  
)
```

Output

```
true
```

Example 4

Determine if the list contains the date April 8, 2022.

Usage

Power Query M

```
let  
    Source = {#date(2024, 2, 23), #date(2023, 12, 2), #date(2022, 4, 8),  
    #date(2021, 7, 6)},  
    ContainsDate = List.Contains(Source, Date.From("4/8/2022"))  
in  
    ContainsDate
```

Output

```
true
```

Related content

Equation criteria

List.ContainsAll

06/19/2025

Syntax

```
List.ContainsAll(list as list, values as list, optional equationCriteria as any)  
as logical
```

About

Indicates whether the list includes all the values from another list. Returns `true` if all the values are found in the list, `false` otherwise.

- `list`: The list to search.
- `values`: The list of values to search for in the first list.
- `equationCriteria`: (Optional) The comparer used to determine if the two values are equal.

Example 1

Determine if the list {1, 2, 3, 4, 5} contains 3 and 4.

Usage

Power Query M

```
List.ContainsAll({1, 2, 3, 4, 5}, {3, 4})
```

Output

```
true
```

Example 2

Determine if the list {1, 2, 3, 4, 5} contains 5 and 6.

Usage

Power Query M


```
List.ContainsAll({1, 2, 3, 4, 5}, {5, 6})
```

Output

```
false
```

Example 3

Determine if the list contains a dog and a horse, while ignoring case.

Usage

Power Query M

```
List.ContainsAll({"dog", "cat", "raccoon", "horse", "rabbit"}, {"DOG", "Horse"},  
Comparer.OrdinalIgnoreCase)
```

Output

```
true
```

Example 4

Determine if the list contains the dates April 8, 2022 and July 6, 2021.

Usage

Power Query M

```
let  
    Source = {#date(2024, 2, 23), #date(2023, 12, 2), #date(2022, 4, 8),  
#date(2021, 7, 6)},  
    ContainsDates = List.ContainsAll(Source, {#date(2022, 4, 8), #date(2021, 7,  
6)})  
in  
    ContainsDates
```

Output

```
true
```

Related content

Equation criteria

List.ContainsAny

06/19/2025

Syntax

```
List.ContainsAny(list as list, values as list, optional equationCriteria as any)  
as logical
```

About

Indicates whether the list contains any of the values from another list. Returns `true` if the values are found in the list, `false` otherwise.

- `list`: The list to search.
- `values`: The list of values to search for in the first list.
- `equationCriteria`: (Optional) The comparer used to determine if the two values are equal.

Example 1

Determine if the list {1, 2, 3, 4, 5} contains 3 or 9.

Usage

Power Query M

```
List.ContainsAny({1, 2, 3, 4, 5}, {3, 9})
```

Output

`true`

Example 2

Determine if the list {1, 2, 3, 4, 5} contains 6 or 7.

Usage

Power Query M

```
List.ContainsAny({1, 2, 3, 4, 5}, {6, 7})
```

Output

```
false
```

Example 3

Determine if the list contains a horse or an owl, while ignoring case.

Usage

Power Query M

```
List.ContainsAny({"dog", "cat", "raccoon", "horse", "rabbit"}, {"Horse", "OWL"},  
Comparer.OrdinalIgnoreCase)
```

Output

```
true
```

Example 4

Determine if the list contains a date of either April 8, 2022 or January 12, 2021.

Usage

Power Query M

```
let  
    Source = {#date(2024, 2, 23), #date(2023, 12, 2), #date(2022, 4, 8),  
#date(2021, 7, 6)},  
    ContainsDates = List.ContainsAny(Source, {Date.From("Apr 8, 2022"),  
Date.From("Jan 11, 2021")})  
in  
    ContainsDates
```

Output

```
true
```

Related content

Equation criteria

List.Count

06/19/2025

Syntax

```
List.Count(list as list) as number
```

About

Returns the number of items in the list `list`.

Example 1

Find the number of values in the list {1, 2, 3}.

Usage

Power Query M

```
List.Count({1, 2, 3})
```

Output

3

List.Covariance

06/19/2025

Syntax

```
List.Covariance(numberList1 as list, numberList2 as list) as nullable number
```

About

Returns the covariance between two lists, `numberList1` and `numberList2`. `numberList1` and `numberList2` must contain the same number of `number` values.

Example 1

Calculate the covariance between two lists.

Usage

Power Query M

```
List.Covariance({1, 2, 3}, {1, 2, 3})
```

Output

```
0.666666666666666607
```

List.Dates

06/19/2025

Syntax

```
List.Dates(start as date, count as number, step as duration) as list
```

About

Returns a list of `date` values of size `count`, starting at `start`. The given increment, `step`, is a `duration` value that is added to every value.

Example 1

Create a list of 5 values starting from New Year's Eve (`#date(2011, 12, 31)`) incrementing by 1 day (`#duration(1, 0, 0, 0)`).

Usage

Power Query M

```
List.Dates(#date(2011, 12, 31), 5, #duration(1, 0, 0, 0))
```

Output

Power Query M

```
{
  #date(2011, 12, 31),
  #date(2012, 1, 1),
  #date(2012, 1, 2),
  #date(2012, 1, 3),
  #date(2012, 1, 4)
}
```

Related content

[#duration](#)

List.DateTimeTimes

06/19/2025

Syntax

```
List.DateTimeTimes(start as datetime, count as number, step as duration) as list
```

About

Returns a list of `datetime` values of size `count`, starting at `start`. The given increment, `step`, is a `duration` value that is added to every value.

Example

Create a list of 10 values starting from 5 minutes before New Year's Day (`#datetime(2011, 12, 31, 23, 55, 0)`) incrementing by 1 minute (`#duration(0, 0, 1, 0)`).

Usage

Power Query M

```
List.DateTimeTimes(#datetime(2011, 12, 31, 23, 55, 0), 10, #duration(0, 0, 1, 0))
```

Output

Power Query M

```
{
  #datetime(2011, 12, 31, 23, 55, 0),
  #datetime(2011, 12, 31, 23, 56, 0),
  #datetime(2011, 12, 31, 23, 57, 0),
  #datetime(2011, 12, 31, 23, 58, 0),
  #datetime(2011, 12, 31, 23, 59, 0),
  #datetime(2012, 1, 1, 0, 0, 0),
  #datetime(2012, 1, 1, 0, 1, 0),
  #datetime(2012, 1, 1, 0, 2, 0),
  #datetime(2012, 1, 1, 0, 3, 0),
  #datetime(2012, 1, 1, 0, 4, 0)
}
```

List.DateTimeZones

06/19/2025

Syntax

```
List.DateTimeZones(start as datetimezone, count as number, step as duration) as list
```

About

Returns a list of `datetimezone` values of size `count`, starting at `start`. The given increment, `step`, is a `duration` value that is added to every value.

Example 1

Create a list of 10 values starting from 5 minutes before New Year's Day (`#datetimezone(2011, 12, 31, 23, 55, 0, -8, 0)`) incrementing by 1 minute (`#duration(0, 0, 1, 0)`).

Usage

Power Query M

```
List.DateTimeZones(#datetimezone(2011, 12, 31, 23, 55, 0, -8, 0), 10, #duration(0, 0, 1, 0))
```

Output

Power Query M

```
{
    #datetimezone(2011, 12, 31, 23, 55, 0, -8, 0),
    #datetimezone(2011, 12, 31, 23, 56, 0, -8, 0),
    #datetimezone(2011, 12, 31, 23, 57, 0, -8, 0),
    #datetimezone(2011, 12, 31, 23, 58, 0, -8, 0),
    #datetimezone(2011, 12, 31, 23, 59, 0, -8, 0),
    #datetimezone(2012, 1, 1, 0, 0, 0, -8, 0),
    #datetimezone(2012, 1, 1, 0, 1, 0, -8, 0),
    #datetimezone(2012, 1, 1, 0, 2, 0, -8, 0),
    #datetimezone(2012, 1, 1, 0, 3, 0, -8, 0),
    #datetimezone(2012, 1, 1, 0, 4, 0, -8, 0)
}
```

List.Difference

06/19/2025

```
List.Difference(list1 as list, list2 as list, optional equationCriteria as any) as list
```

About

Returns the items in list `list1` that do not appear in list `list2`. Duplicate values are supported. An optional equation criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Find the items in list {1, 2, 3, 4, 5} that do not appear in {4, 5, 3}.

Usage

Power Query M

```
List.Difference({1, 2, 3, 4, 5}, {4, 5, 3})
```

Output

```
{1, 2}
```

Example 2

Find the items in the list {1, 2} that do not appear in {1, 2, 3}.

Usage

Power Query M

```
List.Difference({1, 2}, {1, 2, 3})
```

Output

```
{}
```

Related content

[Equation criteria](#)

List.Distinct

06/19/2025

Syntax

```
List.Distinct(list as list, optional equationCriteria as any) as list
```

About

Returns a list that contains all the values in list `list` with duplicates removed. If the list is empty, the result is an empty list.

Example 1

Remove the duplicates from the list {1, 1, 2, 3, 3, 3}.

Usage

Power Query M

```
List.Distinct({1, 1, 2, 3, 3, 3})
```

Output

```
{1, 2, 3}
```

Related content

[Equation criteria](#)

List.Durations

06/19/2025

Syntax

```
List.Durations(start as duration, count as number, step as duration) as list
```

About

Returns a list of `count` `duration` values, starting at `start` and incremented by the given `duration` `step`.

Example 1

Create a list of 5 values starting 1 hour and incrementing by an hour.

Usage

Power Query M

```
List.Durations(#duration(0, 1, 0, 0), 5, #duration(0, 1, 0, 0))
```

Output

Power Query M

```
{
    #duration(0, 1, 0, 0),
    #duration(0, 2, 0, 0),
    #duration(0, 3, 0, 0),
    #duration(0, 4, 0, 0),
    #duration(0, 5, 0, 0)
}
```

List.FindText

06/19/2025

Syntax

```
List.FindText(list as list, text as text) as list
```

About

Returns a list of the values from the list `list` which contained the value `text`.

Example 1

Find the text values in the list {"a", "b", "ab"} that match "a".

Usage

Power Query M

```
List.FindText({"a", "b", "ab"}, "a")
```

Output

```
{"a", "ab"}
```

List.First

06/19/2025

Syntax

```
List.First(list as list, optional defaultValue as any) as any
```

About

Returns the first item in the list `list`, or the optional default value, `defaultValue`, if the list is empty. If the list is empty and a default value is not specified, the function returns `null`.

Example 1

Find the first value in the list {1, 2, 3}.

Usage

```
Power Query M
```

```
List.First({1, 2, 3})
```

Output

```
1
```

Example 2

Find the first value in the list {}. If the list is empty, return -1.

Usage

```
Power Query M
```

```
List.First({}, -1)
```

Output

```
-1
```


List.FirstN

06/19/2025

Syntax

```
List.FirstN(list as list, countOrCondition as any) as any
```

About

- If a number is specified, up to that many items are returned.
- If a condition is specified, all items are returned that initially meet the condition. Once an item fails the condition, no further items are considered.

Example 1

Find the initial values in the list {3, 4, 5, -1, 7, 8, 2} that are greater than 0.

Usage

Power Query M

```
List.FirstN({3, 4, 5, -1, 7, 8, 2}, each _ > 0)
```

Output

```
{3, 4, 5}
```

List.Generate

06/19/2025

Syntax

```
List.Generate(initial as function, condition as function, next as function,  
optional selector as nullable function) as list
```

About

Generates a list of values using the provided functions. The `initial` function generates a starting candidate value, which is then tested against `condition`. If the candidate value is approved, then it's returned as part of the resulting list, and the next candidate value is generated by passing the newly approved value to `next`. Once a candidate value fails to match `condition`, the list generation process stops. An optional parameter, `selector`, may also be provided to transform the items in the resulting list.

Example 1

Create a list by starting at ten, repeatedly decrementing by one, and ensuring each item is greater than zero.

Usage

Power Query M

```
List.Generate(() => 10, each _ > 0, each _ - 1)
```

Output

```
{10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

Example 2

Generate a list of records containing x and y, where x is a value and y is a list. x should remain less than 10 and represent the number of items in the list y. After the list is generated, return only the x values.

Usage

Power Query M

```
List.Generate(  
    () => [x = 1, y = {}],  
    each [x] < 10,  
    each [x = List.Count([y]), y = [y] & {x}],  
    each [x]  
)
```

Output

```
{1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

List.InsertRange

06/19/2025

Syntax

```
List.InsertRange(list as list, index as number, values as list) as list
```

About

Returns a new list produced by inserting the values in `values` into `list` at `index`. The first position in the list is at index 0.

- `list`: The target list where values are to be inserted.
- `index`: The index of the target list(`list`) where the values are to be inserted. The first position in the list is at index 0.
- `values`: The list of values which are to be inserted into `list`.

Example 1

Insert the list ({3, 4}) into the target list ({1, 2, 5}) at index 2.

Usage

Power Query M

```
List.InsertRange({1, 2, 5}, 2, {3, 4})
```

Output

Power Query M

```
{  
    1,  
    2,  
    3,  
    4,  
    5  
}
```

Example 2

Insert a list with a nested list ({1, {1.1, 1.2}}) into a target list ({2, 3, 4}) at index 0.

Usage

Power Query M

```
List.InsertRange({2, 3, 4}, 0, {1, {1.1, 1.2}})
```

Output

Power Query M

```
{  
  1,  
  {  
    1.1,  
    1.2  
  },  
  2,  
  3,  
  4  
}
```

List.Intersect

06/19/2025

Syntax

```
List.Intersect(lists as list, optional equationCriteria as any) as list
```

About

Returns the intersection of the list values found in the input list `lists`. An optional parameter, `equationCriteria`, can be specified.

Example 1

Find the intersection of the lists {1..5}, {2..6}, {3..7}.

Usage

Power Query M

```
List.Intersect({{1..5}, {2..6}, {3..7}})
```

Output

```
{3, 4, 5}
```

Related content

[Equation criteria](#)

List.IsDistinct

06/19/2025

Syntax

```
List.IsDistinct(list as list, optional equationCriteria as any) as logical
```

About

Returns a logical value whether there are duplicates in the list `list`; `true` if the list is distinct, `false` if there are duplicate values.

Example 1

Find if the list {1, 2, 3} is distinct (i.e. no duplicates).

Usage

Power Query M

```
List.IsDistinct({1, 2, 3})
```

Output

true

Example 2

Find if the list {1, 2, 3, 3} is distinct (i.e. no duplicates).

Usage

Power Query M

```
List.IsDistinct({1, 2, 3, 3})
```

Output

false

Related content

[Equation criteria](#)

List.IsEmpty

06/19/2025

Syntax

```
List.IsEmpty(list as list) as logical
```

About

Returns `true` if the list, `list`, contains no values (length 0). If the list contains values (length > 0), returns `false`.

Example 1

Find if the list {} is empty.

Usage

```
Power Query M
```

```
List.IsEmpty({})
```

Output

```
true
```

Example 2

Find if the list {1, 2} is empty.

Usage

```
Power Query M
```

```
List.IsEmpty({1, 2})
```

Output

```
false
```


List.Last

06/19/2025

Syntax

```
List.Last(list as list, optional defaultValue as any) as any
```

About

Returns the last item in the list `list`, or the optional default value, `defaultValue`, if the list is empty. If the list is empty and a default value is not specified, the function returns `null`.

Example 1

Find the last value in the list {1, 2, 3}.

Usage

```
Power Query M
```

```
List.Last({1, 2, 3})
```

Output

```
3
```

Example 2

Find the last value in the list {} or -1 if it empty.

Usage

```
Power Query M
```

```
List.Last({}, -1)
```

Output

```
-1
```


List.LastN

06/19/2025

Syntax

```
List.LastN(list as list, optional countOrCondition as any) as any
```

About

Returns the last item of the list `list`. If the list is empty, an exception is thrown. This function takes an optional parameter, `countOrCondition`, to support gathering multiple items or filtering items. `countOrCondition` can be specified in three ways:

- If a number is specified, up to that many items are returned.
- If a condition is specified, all items are returned that initially meet the condition, starting at the end of the list. Once an item fails the condition, no further items are considered.
- If this parameter is null the last item in the list is returned.

Example 1

Find the last value in the list {3, 4, 5, -1, 7, 8, 2}.

Usage

Power Query M

```
List.LastN({3, 4, 5, -1, 7, 8, 2}, 1)
```

Output

```
{2}
```

Example 2

Find the last values in the list {3, 4, 5, -1, 7, 8, 2} that are greater than 0.

Usage

Power Query M

```
List.LastN({3, 4, 5, -1, 7, 8, 2}, each _ > 0)
```

Output

```
{7, 8, 2}
```

List.MatchesAll

06/19/2025

Syntax

```
List.MatchesAll(list as list, condition as function) as logical
```

About

Returns `true` if the condition function is satisfied by all of the values in the list, otherwise returns `false`.

- `list`: The list containing the values to check.
- `condition`: The condition to check against the values in the list.

Example 1

Determine if all the values in the list {11, 12, 13} are greater than 10.

Usage

Power Query M

```
List.MatchesAll({11, 12, 13}, each _ > 10)
```

Output

```
true
```

Example 2

Determine if all the values in the list {1, 2, 3} are greater than 10.

Usage

Power Query M

```
List.MatchesAll({1, 2, 3}, each _ > 10)
```


Output

```
false
```

Example 3

Determine if all the text values in the list contain "anna" while ignoring case.

Usage

Power Query M

```
let
    Source = {"Savannah", "Annabelle", "Annals", "wannabe", "MANNA"},
    Result = List.MatchesAll(Source, each Text.Contains(_, "anna",
        Comparer.OrdinalIgnoreCase))
in
    Result
```

Output

```
true
```

Example 4

Determine if all the dates contain the year 2021.

Usage

Power Query M

```
let
    Source = {#date(2021, 11, 28), #date(2021, 1, 14), #date(2021, 12, 31),
        #date(2021, 7, 6)},
    Result = List.MatchesAll(Source, each Date.Year(_) = 2021)
in
    Result
```

Output

```
true
```

List.MatchesAny

06/19/2025

Syntax

```
List.MatchesAny(list as list, condition as function) as logical
```

About

Returns `true` if the condition function is satisfied by any of the values in the list, otherwise returns `false`.

- `list`: The list containing the values to check.
- `condition`: The condition to check against the values in the list.

Example 1

Determine if any of the values in the list {9, 10, 11} are greater than 10.

Usage

Power Query M

```
List.MatchesAny({9, 10, 11}, each _ > 10)
```

Output

```
true
```

Example 2

Determine if any of the values in the list {1, 2, 3} are greater than 10.

Usage

Power Query M

```
List.MatchesAny({1, 2, 3}, each _ > 10)
```

Output

```
false
```

Example 3

Determine if any of the text values in the list contain "cat" while ignoring case.

Usage

Power Query M

```
let
    Source = {"A Brown Fox", "A Loyal Dog", "A Curious Cat", "A Wild Horse", "A Rascally Rabbit"},
    Result = List.MatchesAny(Source, each Text.Contains(_, "cat", Comparer.OrdinalIgnoreCase))
in
    Result
```

Output

```
true
```

Example 4

Determine if any of the dates contain the year 2021.

Usage

Power Query M

```
let
    Source = {#date(2024, 11, 28), #date(2023, 1, 14), #date(2021, 12, 31), #date(2025, 7, 6)},
    Result = List.MatchesAny(Source, each Date.Year(_) = 2021)
in
    Result
```

Output

```
true
```

List.Max

06/19/2025

Syntax

```
List.Max(list as list, optional default as any, optional comparisonCriteria as any, optional includeNulls as nullable logical) as any
```

About

Returns the maximum item in the list `list`, or the optional default value `default` if the list is empty. An optional comparisonCriteria value, `comparisonCriteria`, may be specified to determine how to compare the items in the list. If this parameter is null, the default comparer is used.

Example 1

Find the max in the list {1, 4, 7, 3, -2, 5}.

Usage

Power Query M

```
List.Max({1, 4, 7, 3, -2, 5}, 1)
```

Output

7

Example 2

Find the max in the list {} or return -1 if it is empty.

Usage

Power Query M

```
List.Max({}, -1)
```

Output

-1

Related content

[Comparison criteria](#)

List.MaxN

06/19/2025

Syntax

```
List.MaxN(list as list, countOrCondition as any, optional comparisonCriteria as any, optional includeNulls as nullable logical) as list
```

About

Returns the maximum value(s) in the list, `list`. After the rows are sorted, optional parameters may be specified to further filter the result. The optional parameter `countOrCondition` specifies the number of values to return or a filtering condition. The optional parameter `comparisonCriteria` specifies how to compare values in the list.

- `list`: The list of values.
- `countOrCondition`: If a number is specified, a list of up to `countOrCondition` items in ascending order is returned. If a condition is specified, a list of items that initially meet the condition is returned. Once an item fails the condition, no further items are considered.
- `comparisonCriteria`: *[Optional]* An optional `comparisonCriteria` value can be specified to determine how to compare the items in the list. If this parameter is null, the default comparer is used.

Related content

[Comparison criteria](#)

List.Median

06/19/2025

Syntax

```
List.Median(list as list, optional comparisonCriteria as any) as any
```

About

Returns the median item of the list `list`. This function returns `null` if the list contains no non-`null` values. If there is an even number of items, the function chooses the smaller of the two median items unless the list is comprised entirely of datetimes, durations, numbers or times, in which case it returns the average of the two items.

Example 1

Find the median of the list `{5, 3, 1, 7, 9}`.

Usage

Power Query M

```
powerquery-mList.Median({5, 3, 1, 7, 9})
```

Output

5

Related content

[Comparison criteria](#)

List.Min

06/19/2025

Syntax

```
List.Min(list as list, optional default as any, optional comparisonCriteria as any, optional includeNulls as nullable logical) as any
```

About

Returns the minimum item in the list `list`, or the optional default value `default` if the list is empty. An optional comparisonCriteria value, `comparisonCriteria`, may be specified to determine how to compare the items in the list. If this parameter is null, the default comparer is used.

Example 1

Find the min in the list {1, 4, 7, 3, -2, 5}.

Usage

Power Query M

```
List.Min({1, 4, 7, 3, -2, 5})
```

Output

-2

Example 2

Find the min in the list {} or return -1 if it is empty.

Usage

Power Query M

```
List.Min({}, -1)
```


Output

-1

Related content

[Comparison criteria](#)

List.MinN

06/19/2025

Syntax

```
List.MinN(list as list, countOrCondition as any, optional comparisonCriteria as any, optional includeNulls as nullable logical) as list
```

About

Returns the minimum value(s) in the list, `list`. The parameter, `countOrCondition`, specifies the number of values to return or a filtering condition. The optional parameter, `comparisonCriteria`, specifies how to compare values in the list.

- `list`: The list of values.
- `countOrCondition`: If a number is specified, a list of up to `countOrCondition` items in ascending order is returned. If a condition is specified, a list of items that initially meet the condition is returned. Once an item fails the condition, no further items are considered. If this parameter is null, the single smallest value in the list is returned.
- `comparisonCriteria`: *[Optional]* An optional `comparisonCriteria` value can be specified to determine how to compare the items in the list. If this parameter is null, the default comparer is used.

Example 1

Find the 5 smallest values in the list `{3, 4, 5, -1, 7, 8, 2}`.

Usage

Power Query M

```
List.MinN({3, 4, 5, -1, 7, 8, 2}, 5)
```

Output

```
{-1, 2, 3, 4, 5}
```

Related content

Comparison criteria

List.Mode

06/19/2025

Syntax

```
List.Mode(list as list, optional equationCriteria as any) as any
```

About

Returns the item that appears most frequently in `list`. If the list is empty an exception is thrown. If multiple items appear with the same maximum frequency, the last one is chosen. An optional comparison criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Find the item that appears most frequently in the list `{"A", 1, 2, 3, 3, 4, 5}`.

Usage

Power Query M

```
List.Mode({"A", 1, 2, 3, 3, 4, 5})
```

Output

3

Example 2

Find the item that appears most frequently in the list `{"A", 1, 2, 3, 3, 4, 5, 5}`.

Usage

Power Query M

```
List.Mode({"A", 1, 2, 3, 3, 4, 5, 5})
```

Output

5

Related content

[Equation criteria](#)

List.Modes

06/19/2025

Syntax

```
List.Modes(list as list, optional equationCriteria as any) as list
```

About

Returns the items that appear most frequently in `list`. If the list is empty an exception is thrown. If multiple items appear with the same maximum frequency, all of them are returned. An optional comparison criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Find the items that appears most frequently in the list `{"A", 1, 2, 3, 3, 4, 5, 5}`.

Usage

Power Query M

```
List.Modes({"A", 1, 2, 3, 3, 4, 5, 5})
```

Output

```
{3, 5}
```

Related content

[Equation criteria](#)

List.NonNullCount

06/19/2025

Syntax

```
List.NonNullCount(list as list) as number
```

About

Returns the number of non-null items in the list `list`.

List.Numbers

06/19/2025

Syntax

```
List.Numbers(start as number, count as number, optional increment as nullable number) as list
```

About

Returns a list of numbers given an initial value, count, and optional increment value. The default increment value is 1.

- **start**: The initial value in the list.
- **count**: The number of values to create.
- **increment**: *[Optional]* The value to increment by. If omitted values are incremented by 1.

Example 1

Generate a list of 10 consecutive numbers starting at 1.

Usage

Power Query M

```
List.Numbers(1, 10)
```

Output

Power Query M

```
{  
  1,  
  2,  
  3,  
  4,  
  5,  
  6,  
  7,  
  8,  
  9,
```



```
10  
}
```

Example 2

Generate a list of 10 numbers starting at 1, with an increment of 2 for each subsequent number.

Usage

Power Query M

```
List.Numbers(1, 10, 2)
```

Output

Power Query M

```
{  
    1,  
    3,  
    5,  
    7,  
    9,  
    11,  
    13,  
    15,  
    17,  
    19  
}
```

List.Percentile

06/19/2025

Syntax

```
List.Percentile(list as list, percentiles as any, optional options as nullable record) as any
```

About

Returns one or more sample percentiles of the list `list`. If the value `percentiles` is a number between 0.0 and 1.0, it will be treated as a percentile and the result will be a single value corresponding to that probability. If the value `percentiles` is a list of numbers with values between 0.0 and 1.0, the result will be a list of percentiles corresponding to the input probability.

The PercentileMode option in `options` can be used by advanced users to pick a more-specific interpolation method but is not recommended for most uses. Predefined symbols [PercentileMode.ExcelInc](#) and [PercentileMode.ExcelExc](#) match the interpolation methods used by the Excel functions `PERCENTILE.INC` and `PERCENTILE.EXC`. The default behavior matches [PercentileMode.ExcelInc](#). The symbols [PercentileMode.SqlCont](#) and [PercentileMode.SqlDisc](#) match the SQL Server behavior for `PERCENTILE_CONT` and `PERCENTILE_DISC`, respectively.

Example 1

Find the first quartile of the list `{5, 3, 1, 7, 9}`.

Usage

```
Power Query M
```

```
List.Percentile({5, 3, 1, 7, 9}, 0.25)
```

Output

```
3
```

Example 2

Find the quartiles of the list {5, 3, 1, 7, 9} using an interpolation method matching Excel's PERCENTILE.EXC.

Usage

Power Query M

```
List.Percentile({5, 3, 1, 7, 9}, {0.25, 0.5, 0.75},  
[PercentileMode=PercentileMode.ExcelExc])
```

Output

```
{2, 5, 8}
```

List.PositionOf

06/19/2025

Syntax

```
List.PositionOf(list as list, value as any, optional occurrence as nullable number, optional equationCriteria as any) as any
```

About

Returns the offset at which the value `value` appears in the list `list`. Returns -1 if the value doesn't appear. An optional occurrence parameter `occurrence` can be specified.

- `occurrence`: The maximum number of occurrences to report.

Example 1

Find the position in the list {1, 2, 3} at which the value 3 appears.

Usage

Power Query M

```
List.PositionOf({1, 2, 3}, 3)
```

Output

2

Related content

[Equation criteria](#)

List.PositionOfAny

06/19/2025

Syntax

```
List.PositionOfAny(list as list, values as list, optional occurrence as nullable number, optional equationCriteria as any) as any
```

About

Returns the offset in list `list` of the first occurrence of a value in a list `values`. Returns -1 if no occurrence is found. An optional occurrence parameter `occurrence` can be specified.

- `occurrence`: The maximum number of occurrences that can be returned.

Example 1

Find the first position in the list {1, 2, 3} at which the value 2 or 3 appears.

Usage

Power Query M

```
List.PositionOfAny({1, 2, 3}, {2, 3})
```

Output

1

Related content

[Equation criteria](#)

List.Positions

06/19/2025

Syntax

```
List.Positions(list as list) as list
```

About

Returns a list of offsets for the input list `list`. When using List.Transform to change a list, the list of positions can be used to give the transform access to the position.

Example 1

Find the offsets of values in the list {1, 2, 3, 4, null, 5}.

Usage

Power Query M

```
List.Positions({1, 2, 3, 4, null, 5})
```

Output

```
{0, 1, 2, 3, 4, 5}
```

List.Product

06/19/2025

Syntax

```
List.Product(numbersList as list, optional precision as nullable number) as  
nullable number
```

About

Returns the product of the non-null numbers in the list, `numbersList`. Returns null if there are no non-null values in the list.

Example 1

Find the product of the numbers in the list `{1, 2, 3, 3, 4, 5, 5}`.

Usage

Power Query M

```
List.Product({1, 2, 3, 3, 4, 5, 5})
```

Output

1800

List.Random

06/19/2025

Syntax

```
List.Random(count as number, optional seed as nullable number) as list
```

About

Returns a list of random numbers between 0 and 1, given the number of values to generate and an optional seed value.

- **count**: The number of random values to generate.
- **seed**: *[Optional]* A numeric value used to seed the random number generator. If omitted a unique list of random numbers is generated each time you call the function. If you specify the seed value with a number every call to the function generates the same list of random numbers.

Example 1

Create a list of 3 random numbers.

Usage

```
Power Query M
```

```
List.Random(3)
```

Output

```
{0.992332, 0.132334, 0.023592}
```

Example 2

Create a list of 3 random numbers, specifying seed value.

Usage

```
Power Query M
```



```
List.Random(3, 2)
```

Output

```
{0.883002, 0.245344, 0.723212}
```

List.Range

06/19/2025

Syntax

```
List.Range(list as list, offset as number, optional count as nullable number) as list
```

About

Returns a subset of `list` beginning at `offset`. An optional parameter, `count`, sets the maximum number of items in the subset.

Example 1

Find the subset starting at offset 6 of the list of numbers 1 through 10.

Usage

Power Query M

```
List.Range({1..10}, 6)
```

Output

```
{7, 8, 9, 10}
```

Example 2

Find the subset of length 2 from offset 6, from the list of numbers 1 through 10.

Usage

Power Query M

```
List.Range({1..10}, 6, 2)
```

Output

{7, 8}

List.RemoveFirstN

06/19/2025

Syntax

```
List.RemoveFirstN(list as list, optional countOrCondition as any) as list
```

About

Returns a list that removes the first element of list `list`. If `list` is an empty list an empty list is returned. This function takes an optional parameter, `countOrCondition`, to support removing multiple values as listed below.

- If a number is specified, up to that many items are removed.
- If a condition is specified, any consecutive matching items at the start of `list` are removed.
- If this parameter is null, the default behavior is observed.

Example 1

Create a list from {1, 2, 3, 4, 5} without the first 3 numbers.

Usage

Power Query M

```
List.RemoveFirstN({1, 2, 3, 4, 5}, 3)
```

Output

```
{4, 5}
```

Example 2

Create a list from {5, 4, 2, 6, 1} that starts with a number less than 3.

Usage

Power Query M

```
List.RemoveFirstN({5, 4, 2, 6, 1}, each _ > 3)
```

Output

```
{2, 6, 1}
```

List.RemoveItems

06/19/2025

Syntax

```
List.RemoveItems(list1 as list, list2 as list) as list
```

About

Removes all occurrences of the given values in the `list2` from `list1`. If the values in `list2` don't exist in `list1`, the original list is returned.

Example 1

Remove the items in the list {2, 4, 6} from the list {1, 2, 3, 4, 2, 5, 5}.

Usage

Power Query M

```
List.RemoveItems({1, 2, 3, 4, 2, 5, 5}, {2, 4, 6})
```

Output

```
{1, 3, 5, 5}
```

List.RemoveLastN

06/19/2025

Syntax

```
List.RemoveLastN(list as list, optional countOrCondition as any) as list
```

About

Returns a list that removes the last `countOrCondition` elements from the end of list `list`. If `list` has less than `countOrCondition` elements, an empty list is returned.

- If a number is specified, up to that many items are removed.
- If a condition is specified, any consecutive matching items at the end of `list` are removed.
- If this parameter is null, only one item is removed.

Example 1

Create a list from {1, 2, 3, 4, 5} without the last 3 numbers.

Usage

Power Query M

```
List.RemoveLastN({1, 2, 3, 4, 5}, 3)
```

Output

```
{1, 2}
```

Example 2

Create a list from {5, 4, 2, 6, 4} that ends with a number less than 3.

Usage

Power Query M

```
List.RemoveLastN({5, 4, 2, 6, 4}, each _ > 3)
```

Output

```
{5, 4, 2}
```


List.RemoveMatchingItems

06/19/2025

Syntax

```
List.RemoveMatchingItems(list1 as list, list2 as list, optional equationCriteria as any) as list
```

About

Removes all occurrences of the given values in `list2` from the list `list1`. If the values in `list2` don't exist in `list1`, the original list is returned. An optional equation criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Create a list from {1, 2, 3, 4, 5, 5} without {1, 5}.

Usage

Power Query M

```
List.RemoveMatchingItems({1, 2, 3, 4, 5, 5}, {1, 5})
```

Output

```
{2, 3, 4}
```

Related content

[Equation criteria](#)

List.RemoveNulls

06/19/2025

Syntax

```
List.RemoveNulls(list as list) as list
```

About

Removes all occurrences of "null" values in the `list`. If there are no 'null' values in the list, the original list is returned.

Example 1

Remove the "null" values from the list {1, 2, 3, null, 4, 5, null, 6}.

Usage

Power Query M

```
List.RemoveNulls({1, 2, 3, null, 4, 5, null, 6})
```

Output

```
{1, 2, 3, 4, 5, 6}
```

List.RemoveRange

06/19/2025

Syntax

```
List.RemoveRange(list as list, index as number, optional count as nullable number)  
as list
```

About

Removes **count** values in the **list** starting at the specified position, **index**.

Example 1

Remove 3 values in the list {1, 2, 3, 4, -6, -2, -1, 5} starting at index 4.

Usage

Power Query M

```
List.RemoveRange({1, 2, 3, 4, -6, -2, -1, 5}, 4, 3)
```

Output

```
{1, 2, 3, 4, 5}
```

List.Repeat

06/19/2025

Syntax

```
List.Repeat(list as list, count as number) as list
```

About

Returns a list that is **count** repetitions of the original list, **list**.

Example 1

Create a list that has {1, 2} repeated 3 times.

Usage

Power Query M

```
List.Repeat({1, 2}, 3)
```

Output

```
{1, 2, 1, 2, 1, 2}
```

List.ReplaceMatchingItems

06/19/2025

Syntax

```
List.ReplaceMatchingItems(list as list, replacements as list, optional  
equationCriteria as any) as list
```

About

Performs the given replacements to the list `list`. A replacement operation `replacements` consists of a list of two values, the old value and new value, provided in a list. An optional equation criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Create a list from {1, 2, 3, 4, 5} replacing the value 5 with -5, and the value 1 with -1.

Usage

Power Query M

```
List.ReplaceMatchingItems({1, 2, 3, 4, 5}, {{5, -5}, {1, -1}})
```

Output

```
{-1, 2, 3, 4, -5}
```

Related content

[Equation criteria](#)

List.ReplaceRange

06/19/2025

Syntax

```
List.ReplaceRange(list as list, index as number, count as number, replaceWith as list) as list
```

About

Replaces `count` values in the `list` with the list `replaceWith`, starting at specified position, `index`.

Example 1

Replace {7, 8, 9} in the list {1, 2, 7, 8, 9, 5} with {3, 4}.

Usage

Power Query M

```
List.ReplaceRange({1, 2, 7, 8, 9, 5}, 2, 3, {3, 4})
```

Output

```
{1, 2, 3, 4, 5}
```

List.ReplaceValue

06/19/2025

Syntax

```
List.ReplaceValue(list as list, oldValue as any, newValue as any, replacer as function) as list
```

About

Searches a list of values, `list`, for the value `oldValue` and replaces each occurrence with the replacement value `newValue`.

Example 1

Replace all the "a" values in the list {"a", "B", "a", "a"} with "A".

Usage

Power Query M

```
List.ReplaceValue({"a", "B", "a", "a"}, "a", "A", Replacer.ReplaceText)
```

Output

```
{"A", "B", "A", "A"}
```

List.Reverse

06/19/2025

Syntax

```
List.Reverse(list as list) as list
```

About

Returns a list with the values in the list `list` in reversed order.

Example 1

Create a list from {1..10} in reverse order.

Usage

Power Query M

```
List.Reverse({1..10})
```

Output

```
{10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```


List.Select

06/19/2025

Syntax

```
List.Select(list as list, selection as function) as list
```

About

Returns a list of values from the list `list`, that match the selection condition `selection`.

Example 1

Find the values in the list {1, -3, 4, 9, -2} that are greater than 0.

Usage

Power Query M

```
List.Select({1, -3, 4, 9, -2}, each _ > 0)
```

Output

```
{1, 4, 9}
```

List.Single

06/19/2025

Syntax

```
List.Single(list as list) as any
```

About

If there is only one item in the list `list`, returns that item. If there is more than one item or the list is empty, the function throws an exception.

Example 1

Find the single value in the list {1}.

Usage

```
Power Query M
```

```
List.Single({1})
```

Output

```
1
```

Example 2

Find the single value in the list {1, 2, 3}.

Usage

```
Power Query M
```

```
List.Single({1, 2, 3})
```

Output

[Expression.Error] There were too many elements in the enumeration to complete the operation.

List.SingleOrDefault

06/19/2025

Syntax

```
List.SingleOrDefault(list as list, optional default as any) as any
```

About

If there is only one item in the list `list`, returns that item. If the list is empty, the function returns null unless an optional `default` is specified. If there is more than one item in the list, the function returns an error.

Example 1

Find the single value in the list {1}.

Usage

Power Query M

```
List.SingleOrDefault({1})
```

Output

1

Example 2

Find the single value in the list {}.

Usage

Power Query M

```
List.SingleOrDefault({})
```

Output

null

Example 3

Find the single value in the list {}. If is empty, return -1.

Usage

Power Query M

```
List.SingleOrDefault({}, -1)
```

Output

-1

List.Skip

06/19/2025

Syntax

```
List.Skip(list as list, optional countOrCondition as any) as list
```

About

Returns a list that skips the first element of list `list`. If `list` is an empty list an empty list is returned. This function takes an optional parameter, `countOrCondition`, to support skipping multiple values as listed below.

- If a number is specified, up to that many items are skipped.
- If a condition is specified, any consecutive matching items at the start of `list` are skipped.
- If this parameter is null, the default behavior is observed.

Example 1

Create a list from {1, 2, 3, 4, 5} without the first 3 numbers.

Usage

Power Query M

```
List.Skip({1, 2, 3, 4, 5}, 3)
```

Output

```
{4, 5}
```

Example 2

Create a list from {5, 4, 2, 6, 1} that starts with a number less than 3.

Usage

Power Query M

```
List.Skip({5, 4, 2, 6, 1}, each _ > 3)
```

Output

```
{2, 6, 1}
```

List.Sort

06/19/2025

Syntax

```
List.Sort(list as list, optional comparisonCriteria as any) as list
```

About

Sorts a list of data, `list`, according to the optional criteria specified. An optional parameter, `comparisonCriteria`, can be specified as the comparison criterion. This can take the following values:

- To control the order, the comparison criterion can be an Order enum value. ([Order.Descending](#), [Order.Ascending](#)).
- To compute a key to be used for sorting, a function of 1 argument can be used.
- To both select a key and control order, comparison criterion can be a list containing the key and order (`{each 1 / _, Order.Descending}`).
- To completely control the comparison, a function of 2 arguments can be used. This function will be passed two items from the list (any two items, in any order). The function should return one of the following values:
 - `-1`: The first item is less than the second item.
 - `0`: The items are equal.
 - `1`: The first item is greater than the second item.

[Value.Compare](#) is a method that can be used to delegate this logic.

Example 1

Sort the list {2, 3, 1}.

Usage

Power Query M

```
List.Sort({2, 3, 1})
```


Output

```
{1, 2, 3}
```

Example 2

Sort the list {2, 3, 1} in descending order.

Usage

Power Query M

```
List.Sort({2, 3, 1}, Order.Descending)
```

Output

```
{3, 2, 1}
```

Example 3

Sort the list {2, 3, 1} in descending order using the **Value.Compare** method.

Usage

Power Query M

```
List.Sort({2, 3, 1}, (x, y) => Value.Compare(1/x, 1/y))
```

Output

```
{3, 2, 1}
```

List.Split

06/19/2025

Syntax

```
List.Split(list as list, pageSize as number) as list
```

About

Splits `list` into a list of lists where the first element of the output list is a list containing the first `pageSize` elements from the source list, the next element of the output list is a list containing the next `pageSize` elements from the source list, and so on.

List.StandardDeviation

06/19/2025

Syntax

```
List.StandardDeviation(numbersList as list) as nullable number
```

About

Returns a sample based estimate of the standard deviation of the values in the list, **numbersList**. If **numbersList** is a list of numbers, a number is returned. An exception is thrown on an empty list or a list of items that is not type **number**.

Example 1

Find the standard deviation of the numbers 1 through 5.

Usage

Power Query M

```
List.StandardDeviation({1..5})
```

Output

```
1.5811388300841898
```

List.Sum

06/19/2025

Syntax

```
List.Sum(list as list, optional precision as nullable number) as any
```

About

Returns the sum of the non-null values in the list, `list`. Returns null if there are no non-null values in the list.

Example 1

Find the sum of the numbers in the list `{1, 2, 3}`.

Usage

Power Query M

```
List.Sum({1, 2, 3})
```

Output

6

List.Times

06/19/2025

Syntax

```
List.Times(start as time, count as number, step as duration) as list
```

About

Returns a list of `time` values of size `count`, starting at `start`. The given increment, `step`, is a `duration` value that is added to every value.

Example 1

Create a list of 4 values starting from noon (`#time(12, 0, 0)`) incrementing by one hour (`#duration(0, 1, 0, 0)`).

Usage

Power Query M

```
List.Times(#time(12, 0, 0), 4, #duration(0, 1, 0, 0))
```

Output

Power Query M

```
{  
    #time(12, 0, 0),  
    #time(13, 0, 0),  
    #time(14, 0, 0),  
    #time(15, 0, 0)  
}
```

List.Transform

06/19/2025

Syntax

```
List.Transform(list as list, transform as function) as list
```

About

Returns a new list of values by applying the transform function `transform` to the list, `list`.

Example 1

Add 1 to each value in the list {1, 2}.

Usage

Power Query M

```
List.Transform({1, 2}, each _ + 1)
```

Output

```
{2, 3}
```

List.TransformMany

06/19/2025

Syntax

```
List.TransformMany(list as list, collectionTransform as function, resultTransform as function) as list
```

About

Returns a list whose elements are projected from the input list.

The `collectionTransform` function transforms each element into an intermediate list, and the `resultTransform` function receives the original element as well as an item from the intermediate list in order to construct the final result.

The `collectionTransform` function has the signature `(x as any) as list => ...`, where `x` is an element in `list`. The `resultTransform` function projects the shape of the result and has the signature `(x as any, y as any) as any => ...`, where `x` is an element in `list` and `y` is an element from the list generated by passing `x` to `collectionTransform`.

Example 1

Flatten a list of people and their pets.

Usage

Power Query M

```
List.TransformMany(  
    {  
        [Name = "Alice", Pets = {"Scruffy", "Sam"}],  
        [Name = "Bob", Pets = {"Walker"}]  
    },  
    each [Pets],  
    (person, pet) => [Name = person[Name], Pet = pet]  
)
```

Output

Power Query M

```
{  
  [Name = "Alice", Pet = "Scruffy"],  
  [Name = "Alice", Pet = "Sam"],  
  [Name = "Bob", Pet = "Walker"]  
}
```


List.Union

06/19/2025

Syntax

```
List.Union(lists as list, optional equationCriteria as any) as list
```

About

Takes a list of lists `lists`, unions the items in the individual lists and returns them in the output list. As a result, the returned list contains all items in any input lists. This operation maintains traditional bag semantics, so duplicate values are matched as part of the Union. An optional equation criteria value, `equationCriteria`, can be specified to control equality testing.

Example 1

Create a union of the list {1..5}, {2..6}, {3..7}.

Usage

Power Query M

```
List.Union({{1..5}, {2..6}, {3..7}})
```

Output

```
{1, 2, 3, 4, 5, 6, 7}
```

Related content

[Equation criteria](#)

List.Zip

06/19/2025

Syntax

```
List.Zip(lists as list) as list
```

About

Takes a list of lists, `lists`, and returns a list of lists combining items at the same position.

Example 1

Zips the two simple lists {1, 2} and {3, 4}.

Usage

Power Query M

```
List.Zip({{1, 2}, {3, 4}})
```

Output

Power Query M

```
{
    {1, 3},
    {2, 4}
}
```

Example 2

Zips the two simple lists of different lengths {1, 2} and {3}.

Usage

Power Query M

```
List.Zip({{1, 2}, {3}})
```

Output

Power Query M

```
{  
    {1, 3},  
    {2, null}  
}
```

Logical functions

Article • 11/22/2024

These functions create and manipulate logical (that is, `true` or `false`) values.

 Expand table

Name	Description
Logical.From	Creates a logical from the given value.
Logical.FromText	Creates a logical value from the text values "true" and "false".
Logical.ToText	Returns the text "true" or "false" given a logical value.

Feedback

Was this page helpful?  Yes  No

[Provide product feedback](#) | [Ask the community](#) 

Logical.From

06/19/2025

Syntax

```
Logical.From(value as any) as nullable logical
```

About

Returns a `logical` value from the given `value`. If the given `value` is `null`, **Logical.From** returns `null`. If the given `value` is `logical`, `value` is returned.

Values of the following types can be converted to a `logical` value:

- `text`: A `logical` value from the text value, either `"true"` or `"false"`. Refer to [Logical.FromText](#) for details.
- `number`: `false` if `value` equals `0`, `true` otherwise.

If `value` is of any other type, an error is returned.

Example 1

Convert `2` to a `logical` value.

Usage

```
Power Query M
```

```
Logical.From(2)
```

Output

```
true
```

Logical.FromText

06/19/2025

Syntax

```
Logical.FromText(text as nullable text) as nullable logical
```

About

Creates a logical value from the text value `text`, either "true" or "false". If `text` contains a different string, an exception is thrown. The text value `text` is case insensitive.

Example 1

Create a logical value from the text string "true".

Usage

Power Query M

```
Logical.FromText("true")
```

Output

```
true
```

Example 2

Create a logical value from the text string "a".

Usage

Power Query M

```
Logical.FromText("a")
```

Output

[Expression.Error] Could not convert to a logical.

Logical.ToText

06/19/2025

Syntax

```
Logical.ToText(logicalValue as nullable logical) as nullable text
```

About

Creates a text value from the logical value `logicalValue`, either `true` or `false`. If `logicalValue` is not a logical value, an exception is thrown.

Example 1

Create a text value from the logical `true`.

Usage

Power Query M

```
Logical.ToText(true)
```

Output

```
"true"
```


Number functions

Article • 07/18/2023

These functions create and manipulate number values.

Information

Name	Description
Number.IsEven	Returns true if a value is an even number.
Number.IsNaN	Returns true if a value is Number.NaN.
Number.IsOdd	Returns true if a value is an odd number.

Conversion and formatting

Name	Description
Byte.From	Returns an 8-bit integer number value from the given value.
Currency.From	Returns a currency value from the given value.
Decimal.From	Returns a decimal number value from the given value.
Double.From	Returns a Double number value from the given value.
Int8.From	Returns a signed 8-bit integer number value from the given value.
Int16.From	Returns a 16-bit integer number value from the given value.
Int32.From	Returns a 32-bit integer number value from the given value.
Int64.From	Returns a 64-bit integer number value from the given value.
Number.From	Returns a number value from a value.
Number.FromText	Returns a number value from a text value.
Number.ToText	Converts the given number to text.
Percentage.From	Returns a percentage value from the given value.
Single.From	Returns a Single number value from the given value.

Rounding

Name	Description
Number.Round	Returns a nullable number (n) if value is an integer.
Number.RoundAwayFromZero	Returns Number.RoundUp(value) when value ≥ 0 and Number.RoundDown(value) when value < 0 .
Number.RoundDown	Returns the largest integer less than or equal to a number value.
Number.RoundTowardZero	Returns Number.RoundDown(x) when $x \geq 0$ and Number.RoundUp(x) when $x < 0$.
Number.RoundUp	Returns the larger integer greater than or equal to a number value.

Operations

Name	Description
Number.Abs	Returns the absolute value of a number.
Number.Combinations	Returns the number of combinations of a given number of items for the optional combination size.
Number.Exp	Returns a number representing e raised to a power.
Number.Factorial	Returns the factorial of a number.
Number.IntegerDivide	Divides two numbers and returns the whole part of the resulting number.
Number.Ln	Returns the natural logarithm of a number.
Number.Log	Returns the logarithm of a number to the base.
Number.Log10	Returns the base-10 logarithm of a number.
Number.Mod	Divides two numbers and returns the remainder of the resulting number.
Number.Permutations	Returns the number of total permutations of a given number of items for the optional permutation size.
Number.Power	Returns a number raised by a power.
Number.Sign	Returns 1 for positive numbers, -1 for negative numbers or 0 for zero.
Number.Sqrt	Returns the square root of a number.

Random

Name	Description
Number.Random	Returns a random fractional number between 0 and 1.
Number.RandomBetween	Returns a random number between the two given number values.

Trigonometry

Name	Description
Number.Acos	Returns the arccosine of a number.
Number.Asin	Returns the arcsine of a number.
Number.Atan	Returns the arctangent of a number.
Number.Atan2	Returns the arctangent of the division of two numbers.
Number.Cos	Returns the cosine of a number.
Number.Cosh	Returns the hyperbolic cosine of a number.
Number.Sin	Returns the sine of a number.
Number.Sinh	Returns the hyperbolic sine of a number.
Number.Tan	Returns the tangent of a number.
Number.Tanh	Returns the hyperbolic tangent of a number.

Bytes

Name	Description
Number.BitwiseAnd	Returns the result of a bitwise AND operation on the provided operands.
Number.BitwiseNot	Returns the result of a bitwise NOT operation on the provided operands.
Number.BitwiseOr	Returns the result of a bitwise OR operation on the provided operands.
Number.BitwiseShiftLeft	Returns the result of a bitwise shift left operation on the operands.

Name	Description
Number.BitwiseShiftRight	Returns the result of a bitwise shift right operation on the operands.
Number.BitwiseXor	Returns the result of a bitwise XOR operation on the provided operands.

Feedback

Was this page helpful?



Yes



No

[Get help at Microsoft Q&A](#)

Byte.From

06/19/2025

Syntax

```
Byte.From(value as any, optional culture as nullable text, optional roundingMode as nullable number) as nullable number
```

About

Returns an 8-bit integer `number` value from the given `value`. If the given `value` is `null`, **Byte.From** returns `null`. If the given `value` is a `number` within the range of an 8-bit integer without a fractional part, `value` is returned. If it has fractional part, then the number is rounded with the rounding mode specified. The default rounding mode is [RoundingMode.ToEven](#). If `value` is of any other type, it will first be converted to a `number` using [Number.FromText](#). Refer to [Number.Round](#) for the available rounding modes. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the 8-bit integer `number` value of "4".

Usage

```
Power Query M
```

```
Byte.From("4")
```

Output

```
4
```

Example 2

Get the 8-bit integer `number` value of "4.5" using [RoundingMode.AwayFromZero](#).

Usage

Power Query M

```
Byte.From("4.5", null, RoundingMode.AwayFromZero)
```

Output

5

Related content

- [How culture affects text formatting](#)

Currency.From

06/19/2025

Syntax

```
Currency.From(value as any, optional culture as nullable text, optional roundingMode as nullable number) as nullable number
```

About

Returns a `currency` value from the given `value`. If the given `value` is `null`, **Currency.From** returns `null`. If the given `value` is `number` within the range of currency, fractional part of the `value` is rounded to 4 decimal digits and returned. If `value` is of any other type, it will first be converted to a `number` using [Number.FromText](#). Valid range for currency is `-922,337,203,685,477.5808` to `922,337,203,685,477.5807`. Refer to [Number.Round](#) for the available rounding modes. The default is [RoundingMode.ToEven](#). An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the `currency` value of `"1.23455"`.

Usage

```
Power Query M
```

```
Currency.From("1.23455")
```

Output

```
1.2346
```

Example 2

Get the `currency` value of `"1.23455"` using `RoundingMode.Down`.

Usage

Power Query M

```
Currency.From("1.23455", "en-US", RoundingMode.Down)
```

Output

1.2345

Related content

- [How culture affects text formatting](#)

Decimal.From

06/19/2025

Syntax

```
Decimal.From(value as any, optional culture as nullable text) as nullable number
```

About

Returns a Decimal `number` value from the given `value`. If the given `value` is `null`, **Decimal.From** returns `null`. If the given `value` is `number` within the range of Decimal, `value` is returned, otherwise an error is returned. If `value` is of any other type, it will first be converted to a `number` using [Number.FromText](#). An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the Decimal `number` value of "4.5".

Usage

Power Query M

```
Decimal.From("4.5")
```

Output

4.5

Related content

- [How culture affects text formatting](#)

Double.From

06/19/2025

Syntax

```
Double.From(value as any, optional culture as nullable text) as nullable number
```

About

Returns a Double **number** value from the given **value**. If the given **value** is **null**, **Double.From** returns **null**. If the given **value** is **number** within the range of Double, **value** is returned, otherwise an error is returned. If **value** is of any other type, it will first be converted to a **number** using [Number.FromText](#). An optional **culture** may also be provided (for example, "en-US").

Example 1

Get the Double **number** value of **"4"**.

Usage

Power Query M

```
Double.From("4.5")
```

Output

4.5

Related content

- [How culture affects text formatting](#)

Int8.From

06/19/2025

Syntax

```
Int8.From(value as any, optional culture as nullable text, optional roundingMode as nullable number) as nullable number
```

About

Returns a signed 8-bit integer `number` value from the given `value`. If the given `value` is `null`, **Int8.From** returns `null`. If the given `value` is `number` within the range of signed 8-bit integer without a fractional part, `value` is returned. If it has fractional part, then the number is rounded with the rounding mode specified. The default rounding mode is [RoundingMode.ToEven](#). If `value` is of any other type, it will first be converted to a `number` using [Number.FromText](#). Refer to [Number.Round](#) for the available rounding modes. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the signed 8-bit integer `number` value of `"4"`.

Usage

```
Power Query M
```

```
Int8.From("4")
```

Output

```
4
```

Example 2

Get the signed 8-bit integer `number` value of `"4.5"` using [RoundingMode.AwayFromZero](#).

Usage

Power Query M

```
Int8.From("4.5", null, RoundingMode.AwayFromZero)
```

Output

5

Related content

- [How culture affects text formatting](#)

Int16.From

06/19/2025

Syntax

```
Int16.From(value as any, optional culture as nullable text, optional roundingMode as nullable number) as nullable number
```

About

Returns a 16-bit integer `number` value from the given `value`. If the given `value` is `null`, **Int16.From** returns `null`. If the given `value` is `number` within the range of 16-bit integer without a fractional part, `value` is returned. If it has fractional part, then the number is rounded with the rounding mode specified. The default rounding mode is [RoundingMode.ToEven](#). If `value` is of any other type, it will first be converted to a `number` using [Number.FromText](#). Refer to [Number.Round](#) for the available rounding modes. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the 16-bit integer `number` value of "4".

Usage

```
Power Query M
```

```
Int64.From("4")
```

Output

```
4
```

Example 2

Get the 16-bit integer `number` value of "4.5" using `RoundingMode.AwayFromZero`.

Usage

Power Query M

```
Int16.From("4.5", null, RoundingMode.AwayFromZero)
```

Output

5

Related content

- [How culture affects text formatting](#)

Int32.From

06/19/2025

Syntax

```
Int32.From(value as any, optional culture as nullable text, optional roundingMode as nullable number) as nullable number
```

About

Returns a 32-bit integer `number` value from the given `value`. If the given `value` is `null`, **Int32.From** returns `null`. If the given `value` is `number` within the range of 32-bit integer without a fractional part, `value` is returned. If it has fractional part, then the number is rounded with the rounding mode specified. The default rounding mode is [RoundingMode.ToEven](#). If `value` is of any other type, it will first be converted to a `number` using [Number.FromText](#). Refer to [Number.Round](#) for the available rounding modes. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the 32-bit integer `number` value of "4".

Usage

Power Query M

```
Int32.From("4")
```

Output

4

Example 2

Get the 32-bit integer `number` value of "4.5" using `RoundingMode.AwayFromZero`.

Usage

Power Query M

```
Int32.From("4.5", null, RoundingMode.AwayFromZero)
```

Output

5

Related content

- [How culture affects text formatting](#)

Int64.From

06/19/2025

Syntax

```
Int64.From(value as any, optional culture as nullable text, optional roundingMode as nullable number) as nullable number
```

About

Returns a 64-bit integer `number` value from the given `value`. If the given `value` is `null`, **Int64.From** returns `null`. If the given `value` is `number` within the range of 64-bit integer without a fractional part, `value` is returned. If it has fractional part, then the number is rounded with the rounding mode specified. The default rounding mode is [RoundingMode.ToEven](#). If `value` is of any other type, it will first be converted to a `number` using [Number.FromText](#). Refer to [Number.Round](#) for the available rounding modes. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the 64-bit integer `number` value of "4".

Usage

Power Query M

```
Int64.From("4")
```

Output

4

Example 2

Get the 64-bit integer `number` value of "4.5" using `RoundingMode.AwayFromZero`.

Usage

Power Query M

```
Int64.From("4.5", null, RoundingMode.AwayFromZero)
```

Output

5

Related content

- [How culture affects text formatting](#)

Number.Abs

06/19/2025

Syntax

```
Number.Abs(number as nullable number) as nullable number
```

About

Returns the absolute value of `number`. If `number` is null, **Number.Abs** returns null.

- `number`: A `number` for which the absolute value is to be calculated.

Example 1

Absolute value of -3.

Usage

```
Power Query M
```

```
Number.Abs(-3)
```

Output

```
3
```

Number.Acos

06/19/2025

Syntax

```
Number.Acos(number as nullable number) as nullable number
```

About

Returns the arccosine of **number**.

Number.Asin

06/19/2025

Syntax

```
Number.Asin(number as nullable number) as nullable number
```

About

Returns the arcsine of `number`.

Number.Atan

06/19/2025

Syntax

```
Number.Atan(number as nullable number) as nullable number
```

About

Returns the arctangent of `number`.

Number.Atan2

06/19/2025

Syntax

```
Number.Atan2(y as nullable number, x as nullable number) as nullable number
```

About

Returns the angle, in radians, whose tangent is the quotient y/x of the two numbers y and x .

Number.BitwiseAnd

06/19/2025

Syntax

```
Number.BitwiseAnd(number1 as nullable number, number2 as nullable number) as  
nullable number
```

About

Returns the result of performing a bitwise "And" operation between `number1` and `number2`.

Number.BitwiseNot

06/19/2025

Syntax

```
Number.BitwiseNot(number as any) as any
```

About

Returns the result of performing a bitwise "Not" operation on `number`.

Number.BitwiseOr

06/19/2025

Syntax

```
Number.BitwiseOr(number1 as nullable number, number2 as nullable number) as  
nullable number
```

About

Returns the result of performing a bitwise "Or" between `number1` and `number2`.

Number.BitwiseShiftLeft

06/19/2025

Syntax

```
Number.BitwiseShiftLeft(number1 as nullable number, number2 as nullable number) as nullable number
```

About

Returns the result of performing a bitwise shift to the left on `number1`, by the specified number of bits `number2`.

Number.BitwiseShiftRight

06/19/2025

Syntax

```
Number.BitwiseShiftRight(number1 as nullable number, number2 as nullable number)  
as nullable number
```

About

Returns the result of performing a bitwise shift to the right on **number1**, by the specified number of bits **number2**.

Number.BitwiseXor

06/19/2025

Syntax

```
Number.BitwiseXor(number1 as nullable number, number2 as nullable number) as  
nullable number
```

About

Returns the result of performing a bitwise "XOR" (Exclusive-OR) between `number1` and `number2`.

Number.Combinations

06/19/2025

Syntax

```
Number.Combinations(setSize as nullable number, combinationSize as nullable number) as nullable number
```

About

Returns the number of unique combinations from a list of items, **setSize** with specified combination size, **combinationSize**.

- **setSize**: The number of items in the list.
- **combinationSize**: The number of items in each combination.

Example 1

Find the number of combinations from a total of 5 items when each combination is a group of 3.

Usage

Power Query M

```
Number.Combinations(5, 3)
```

Output

10

Number.Cos

06/19/2025

Syntax

```
Number.Cos(number as nullable number) as nullable number
```

About

Returns the cosine of `number`.

Example 1

Find the cosine of the angle 0.

Usage

```
Power Query M
```

```
Number.Cos(0)
```

Output

```
1
```

Number.Cosh

06/19/2025

Syntax

```
Number.Cosh(number as nullable number) as nullable number
```

About

Returns the hyperbolic cosine of `number`.

Number.Exp

06/19/2025

Syntax

```
Number.Exp(number as nullable number) as nullable number
```

About

Returns the result of raising e to the power of **number** (exponential function).

- **number**: A **number** for which the exponential function is to be calculated. If **number** is null, **Number.Exp** returns null.

Example 1

Raise e to the power of 3.

Usage

```
Power Query M
```

```
Number.Exp(3)
```

Output

```
20.085536923187668
```

Number.Factorial

06/19/2025

Syntax

```
Number.Factorial(number as nullable number) as nullable number
```

About

Returns the factorial of the number `number`.

Example 1

Find the factorial of 10.

Usage

```
Power Query M
```

```
Number.Factorial(10)
```

Output

```
3628800
```

Number.From

06/19/2025

Syntax

```
Number.From(value as any, optional culture as nullable text) as nullable number
```

About

Returns a **number** value from the given **value**. An optional **culture** may also be provided (for example, "en-US"). If the given **value** is **null**, **Number.From** returns **null**. If the given **value** is **number**, **value** is returned. Values of the following types can be converted to a **number** value:

- **text**: A **number** value from textual representation. Common text formats are handled ("15", "3,423.10", "5.0E-10"). Refer to [Number.FromText](#) for details.
- **logical**: 1 for **true**, 0 for **false**.
- **datetime**: A double-precision floating-point number that contains an OLE Automation date equivalent.
- **datetimezone**: A double-precision floating-point number that contains an OLE Automation date equivalent of the local date and time of **value**.
- **date**: A double-precision floating-point number that contains an OLE Automation date equivalent.
- **time**: Expressed in fractional days.
- **duration**: Expressed in whole and fractional days.

If **value** is of any other type, an error is returned.

Example 1

Get the **number** value of "4".

Usage

```
Power Query M
```

```
Number.From("4")
```

Output

Example 2

Get the `number` value of `#datetime(2020, 3, 20, 6, 0, 0)`.

Usage

Power Query M

```
Number.From(#datetime(2020, 3, 20, 6, 0, 0))
```

Output

43910.25

Example 3

Get the `number` value of `"12.3%"`.

Usage

Power Query M

```
Number.From("12.3%")
```

Output

0.123

Related content

- [How culture affects text formatting](#)
- [Standard numeric format strings](#)
- [Custom numeric format strings](#)

Number.FromText

06/19/2025

Syntax

```
Number.FromText(text as nullable text, optional culture as nullable text) as nullable number
```

About

Returns a `number` value from the given text value, `text`.

- `text`: The textual representation of a number value. The representation must be in a common number format, such as "15", "3,423.10", or "5.0E-10".
- `culture`: An optional culture that controls how `text` is interpreted (for example, "en-US").

Example 1

Get the number value of `"4"`.

Usage

```
Power Query M
```

```
Number.FromText("4")
```

Output

```
4
```

Example 2

Get the number value of `"5.0e-10"`.

Usage

```
Power Query M
```

```
Number.FromText("5.0e-10")
```

Output

5E-10

Related content

- [How culture affects text formatting](#)
- [Standard numeric format strings](#)
- [Custom numeric format strings](#)

Number.IntegerDivide

06/19/2025

Syntax

```
Number.IntegerDivide(number1 as nullable number, number2 as nullable number,  
optional precision as nullable number) as nullable number
```

About

Returns the integer portion of the result from dividing a number, `number1`, by another number, `number2`. If `number1` or `number2` are null, **Number.IntegerDivide** returns null.

- `number1`: The dividend.
- `number2`: The divisor.

Example 1

Divide 6 by 4.

Usage

```
Power Query M
```

```
Number.IntegerDivide(6, 4)
```

Output

```
1
```

Example 2

Divide 8.3 by 3.

Usage

```
Power Query M
```

```
Number.IntegerDivide(8.3, 3)
```

Output

2

Number.IsEven

06/19/2025

Syntax

```
Number.IsEven(number as number) as logical
```

About

Indicates if the value, `number`, is even by returning `true` if it is even, `false` otherwise.

Example 1

Check if 625 is an even number.

Usage

```
Power Query M
```

```
Number.IsEven(625)
```

Output

```
false
```

Example 2

Check if 82 is an even number.

Usage

```
Power Query M
```

```
Number.IsEven(82)
```

Output

```
true
```

Number.IsNaN

06/19/2025

Syntax

```
Number.IsNaN(number as number) as logical
```

About

Indicates if the value is NaN (Not a number). Returns `true` if `number` is equivalent to `Number.IsNaN`, `false` otherwise.

Example 1

Check if 0 divided by 0 is NaN.

Usage

```
Power Query M
```

```
Number.IsNaN(0/0)
```

Output

```
true
```

Example 2

Check if 1 divided by 0 is NaN.

Usage

```
Power Query M
```

```
Number.IsNaN(1/0)
```

Output

```
false
```


Number.IsOdd

06/19/2025

Syntax

```
Number.IsOdd(number as number) as logical
```

About

Indicates if the value is odd. Returns `true` if `number` is an odd number, `false` otherwise.

Example 1

Check if 625 is an odd number.

Usage

```
Power Query M
```

```
Number.IsOdd(625)
```

Output

```
true
```

Example 2

Check if 82 is an odd number.

Usage

```
Power Query M
```

```
Number.IsOdd(82)
```

Output

```
false
```

Number.Ln

06/19/2025

Syntax

```
Number.Ln(number as nullable number) as nullable number
```

About

Returns the natural logarithm of a number, `number`. If `number` is null `Number.Ln` returns null.

Example 1

Get the natural logarithm of 15.

Usage

```
Power Query M
```

```
Number.Ln(15)
```

Output

```
2.70805020110221
```

Number.Log

06/19/2025

Syntax

```
Number.Log(number as nullable number, optional base as nullable number) as nullable number
```

About

Returns the logarithm of a number, `number`, to the specified `base` base. If `base` is not specified, the default value is Number.E. If `number` is null **Number.Log** returns null.

Example 1

Get the base 10 logarithm of 2.

Usage

```
Power Query M
```

```
Number.Log(2, 10)
```

Output

```
0.3010299956639812
```

Example 2

Get the base e logarithm of 2.

Usage

```
Power Query M
```

```
Number.Log(2)
```

Output

0.69314718055994529

Number.Log10

06/19/2025

Syntax

```
Number.Log10(number as nullable number) as nullable number
```

About

Returns the base 10 logarithm of a number, `number`. If `number` is null **Number.Log10** returns null.

Example 1

Get the base 10 logarithm of 2.

Usage

```
Power Query M
```

```
Number.Log10(2)
```

Output

```
0.3010299956639812
```


Number.Mod

06/19/2025

Syntax

```
Number.Mod(number as nullable number, divisor as nullable number, optional  
precision as nullable number) as nullable number
```

About

Returns the remainder resulting from the integer division of `number` by `divisor`. If `number` or `divisor` are null, `Number.Mod` returns null.

- `number`: The dividend.
- `divisor`: The divisor.

Example 1

Find the remainder when you divide 5 by 3.

Usage

```
Power Query M
```

```
Number.Mod(5, 3)
```

Output

```
2
```

Number.Permutations

06/19/2025

Syntax

```
Number.Permutations(setSize as nullable number, permutationSize as nullable number) as nullable number
```

About

Returns the number of permutations that can be generated from a number of items, `setSize`, with a specified permutation size, `permutationSize`.

Example 1

Find the number of permutations from a total of 5 items in groups of 3.

Usage

Power Query M

```
Number.Permutations(5, 3)
```

Output

60

Number.Power

06/19/2025

Syntax

```
Number.Power(number as nullable number, power as nullable number) as nullable number
```

About

Returns the result of raising `number` to the power of `power`. If `number` or `power` are null, **Number.Power** returns null.

- `number`: The base.
- `power`: The exponent.

Example 1

Find the value of 5 raised to the power of 3 (5 cubed).

Usage

```
Power Query M
```

```
Number.Power(5, 3)
```

Output

```
125
```

Number.Random

06/19/2025

Syntax

```
Number.Random() as number
```

About

Returns a random number between 0 and 1.

Example 1

Get a random number.

Usage

```
Power Query M
```

```
Number.Random()
```

Output

```
0.919303
```

Number.RandomBetween

06/19/2025

Syntax

```
Number.RandomBetween(bottom as number, top as number) as number
```

About

Returns a random number between **bottom** and **top**.

Example 1

Get a random number between 1 and 5.

Usage

Power Query M

```
Number.RandomBetween(1, 5)
```

Output

2.546797

Number.Round

06/19/2025

Syntax

```
Number.Round(number as nullable number, optional digits as nullable number,  
optional roundingMode as nullable number) as nullable number
```

About

Returns the result of rounding `number` to the nearest number. If `number` is null, **Number.Round** returns null.

By default, `number` is rounded to the nearest integer, and ties are broken by rounding to the nearest even number (using [RoundingMode.ToEven](#), also known as "banker's rounding").

However, these defaults can be overridden via the following optional parameters.

- `digits`: Causes `number` to be rounded to the specified number of decimal digits.
- `roundingMode`: Overrides the default tie-breaking behavior when `number` is at the midpoint between two potential rounded values (refer to [RoundingMode.Type](#) for possible values).

Example 1

Round 1.234 to the nearest integer.

Usage

```
Power Query M
```

```
Number.Round(1.234)
```

Output

```
1
```

Example 2

Round 1.56 to the nearest integer.

Usage

Power Query M

```
Number.Round(1.56)
```

Output

2

Example 3

Round 1.2345 to two decimal places.

Usage

Power Query M

```
Number.Round(1.2345, 2)
```

Output

1.23

Example 4

Round 1.2345 to three decimal places (Rounding up).

Usage

Power Query M

```
Number.Round(1.2345, 3, RoundingMode.Up)
```

Output

1.235

Example 5

Round 1.2345 to three decimal places (Rounding down).

Usage

Power Query M

```
Number.Round(1.2345, 3, RoundingMode.Down)
```

Output

1.234

Number.RoundAwayFromZero

06/19/2025

Syntax

```
Number.RoundAwayFromZero(number as nullable number, optional digits as nullable number) as nullable number
```

About

Returns the result of rounding `number` based on the sign of the number. This function will round positive numbers up and negative numbers down. If `digits` is specified, `number` is rounded to the `digits` number of decimal digits.

Example 1

Round the number -1.2 away from zero.

Usage

```
Power Query M
```

```
Number.RoundAwayFromZero(-1.2)
```

Output

```
-2
```

Example 2

Round the number 1.2 away from zero.

Usage

```
Power Query M
```

```
Number.RoundAwayFromZero(1.2)
```

Output

Example 3

Round the number -1.234 to two decimal places away from zero.

Usage

Power Query M

```
Number.RoundAwayFromZero(-1.234, 2)
```

Output

-1.24

Number.RoundDown

06/19/2025

Syntax

```
Number.RoundDown(number as nullable number, optional digits as nullable number) as nullable number
```

About

Returns the result of rounding `number` down to the previous highest integer. If `number` is null, this function returns null. If `digits` is provided, `number` is rounded to the specified number of decimal digits.

Example 1

Round down 1.234 to integer.

Usage

```
Power Query M
```

```
Number.RoundDown(1.234)
```

Output

1

Example 2

Round down 1.999 to integer.

Usage

```
Power Query M
```

```
Number.RoundDown(1.999)
```

Output

Example 3

Round down 1.999 to two decimal places.

Usage

Power Query M

```
Number.RoundDown(1.999, 2)
```

Output

1.99

Number.RoundTowardZero

06/19/2025

Syntax

```
Number.RoundTowardZero(number as nullable number, optional digits as nullable number) as nullable number
```

About

Returns the result of rounding `number` based on the sign of the number. This function will round positive numbers down and negative numbers up. If `digits` is specified, `number` is rounded to the `digits` number of decimal digits.

Example 1

Round the number -1.2 toward zero.

Usage

```
Power Query M
```

```
Number.RoundTowardZero(-1.2)
```

Output

```
-1
```

Example 2

Round the number 1.2 toward zero.

Usage

```
Power Query M
```

```
Number.RoundTowardZero(1.2)
```

Output

Example 3

Round the number -1.234 to two decimal places toward zero.

Usage

Power Query M

```
Number.RoundTowardZero(-1.234, 2)
```

Output

-1.23

Number.RoundUp

06/19/2025

Syntax

```
Number.RoundUp(number as nullable number, optional digits as nullable number) as nullable number
```

About

Returns the result of rounding `number` up to the next highest integer. If `number` is null, this function returns null. If `digits` is provided, `number` is rounded to the specified number of decimal digits.

Example 1

Round up 1.234 to integer.

Usage

```
Power Query M
```

```
Number.RoundUp(1.234)
```

Output

```
2
```

Example 2

Round up 1.999 to integer.

Usage

```
Power Query M
```

```
Number.RoundUp(1.999)
```

Output

Example 3

Round up 1.234 to two decimal places.

Usage

Power Query M

```
Number.RoundUp(1.234, 2)
```

Output

1.24

Number.Sign

06/19/2025

Syntax

```
Number.Sign(number as nullable number) as nullable number
```

About

Returns 1 for if `number` is a positive number, -1 if it is a negative number, and 0 if it is zero. If `number` is null, `Number.Sign` returns null.

Example 1

Determine the sign of 182.

Usage

```
Power Query M
```

```
Number.Sign(182)
```

Output

```
1
```

Example 2

Determine the sign of -182.

Usage

```
Power Query M
```

```
Number.Sign(-182)
```

Output

```
-1
```

Example 3

Determine the sign of 0.

Usage

```
Power Query M
```

```
Number.Sign(0)
```

Output

```
0
```

Number.Sin

06/19/2025

Syntax

```
Number.Sin(number as nullable number) as nullable number
```

About

Returns the sine of `number`.

Example 1

Find the sine of the angle 0.

Usage

```
Power Query M
```

```
Number.Sin(0)
```

Output

```
0
```

Number.Sinh

06/19/2025

Syntax

```
Number.Sinh(number as nullable number) as nullable number
```

About

Returns the hyperbolic sine of `number`.

Number.Sqrt

06/19/2025

Syntax

```
Number.Sqrt(number as nullable number) as nullable number
```

About

Returns the square root of `number`. If `number` is null, **Number.Sqrt** returns null. If it is a negative value, **Number.NaN** is returned (Not a number).

Example 1

Find the square root of 625.

Usage

```
Power Query M
```

```
Number.Sqrt(625)
```

Output

```
25
```

Example 2

Find the square root of 85.

Usage

```
Power Query M
```

```
Number.Sqrt(85)
```

Output

```
9.2195444572928871
```


Number.Tan

06/19/2025

Syntax

```
Number.Tan(number as nullable number) as nullable number
```

About

Returns the tangent of `number`.

Example 1

Find the tangent of the angle 1.

Usage

```
Power Query M
```

```
Number.Tan(1)
```

Output

```
1.5574077246549023
```

Number.Tanh

06/19/2025

Syntax

```
Number.Tanh(number as nullable number) as nullable number
```

About

Returns the hyperbolic tangent of `number`.

Number.ToText

06/19/2025

Syntax

```
Number.ToText(number as nullable number, optional format as nullable text,  
optional culture as nullable text) as nullable text
```

About

Converts the numeric value `number` to a text value according to the format specified by `format`.

The format is a text value indicating how the number should be converted. For more details on the supported format values, go to [Standard numeric format strings](#) and [Custom numeric format strings](#).

An optional `culture` may also be provided (for example, "en-US") to control the culture-dependent behavior of `format`.

Example 1

Convert a number to text without specifying a format.

Usage

```
Power Query M
```

```
Number.ToText(4)
```

Output

```
"4"
```

Example 2

Convert a number to exponential format.

Usage

```
Power Query M
```

```
Number.ToText(4, "e")
```

Output

```
"4.000000e+000"
```

Example 3

Convert a number to percentage format with only one decimal place.

Usage

```
Power Query M
```

```
Number.ToText(-0.1234, "P1")
```

Output

```
"-12.3 %"
```

Related content

- [How culture affects text formatting](#)
- [Standard numeric format strings](#)
- [Custom numeric format strings](#)

Percentage.From

06/19/2025

Syntax

```
Percentage.From(value as any, optional culture as nullable text) as nullable number
```

About

Returns a `percentage` value from the given `value`. If the given `value` is `null`, **Percentage.From** returns `null`. If the given `value` is `text` with a trailing percent symbol, then the converted decimal number will be returned. Otherwise, the value will be converted to a `number` using [Number.From](#). An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the `percentage` value of "12.3%".

Usage

```
Power Query M
```

```
Percentage.From("12.3%")
```

Output

```
0.123
```

Related content

- [How culture affects text formatting](#)

Single.From

06/19/2025

Syntax

```
Single.From(value as any, optional culture as nullable text) as nullable number
```

About

Returns a Single `number` value from the given `value`. If the given `value` is `null`, **Single.From** returns `null`. If the given `value` is `number` within the range of Single, `value` is returned, otherwise an error is returned. If `value` is of any other type, it will first be converted to a `number` using [Number.FromText](#). An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the Single `number` value of `"1.5"`.

Usage

Power Query M

```
Single.From("1.5")
```

Output

```
1.5
```

Related content

- [How culture affects text formatting](#)

Record functions

06/19/2025

These functions create and manipulate record values.

Information

 Expand table

Name	Description
Record.FieldCount	Returns the number of fields in a record.
Record.HasFields	Returns true if the field name or field names are present in a record.

Transformations

 Expand table

Name	Description
Geography.FromWellKnownText	Translates text representing a geographic value in Well-Known Text (WKT) format into a structured record.
Geography.ToWellKnownText	Translates a structured geographic point value into its Well-Known Text (WKT) representation.
GeographyPoint.From	Creates a record representing a geographic point from parts.
Geometry.FromWellKnownText	Translates text representing a geometric value in Well-Known Text (WKT) format into a structured record.
Geometry.ToWellKnownText	Translates a structured geometric point value into its Well-Known Text (WKT) representation.
GeometryPoint.From	Creates a record representing a geometric point from parts.
Record.AddField	Adds a field from a field name and value.
Record.Combine	Combines the records in a list.
Record.RemoveFields	Removes the specified field(s) from the input record.
Record.RenameFields	Returns a new record that renames the fields specified. The resultant fields will retain their original order. This function supports swapping

Name	Description
	and chaining field names. However, all target names plus remaining field names must constitute a unique set or an error will occur.
Record.ReorderFields	Reorders record fields to match the order of a list of field names.
Record.TransformFields	Transforms fields by applying transformOperations. For more more information about values supported by transformOperations, go to Parameter Values .

Selection

 Expand table

Name	Description
Record.Field	Returns the value of the given field. This function can be used to dynamically create field lookup syntax for a given record. In that way it is a dynamic version of the record[field] syntax.
Record.FieldNames	Returns a list of field names in order of the record's fields.
Record.FieldOrDefault	Returns the value of a field from a record, or the default value if the field does not exist.
Record.FieldValues	Returns a list of field values in order of the record's fields.
Record.SelectFields	Returns a new record that contains the fields selected from the input record. The original order of the fields is maintained.

Serialization

 Expand table

Name	Description
Record.FromList	Returns a record given a list of field values and a set of fields.
Record.FromTable	Returns a record from a table of records containing field names and values.
Record.ToList	Returns a list of values containing the field values of the input record.
Record.ToTable	Returns a table of records containing field names and values from an input record.

Parameter Values

The following type definitions are used to describe the parameter values that are referenced in the Record functions above.

 Expand table

Type Definition	Description
MissingField option	More information: MissingField.Type
Transform operations	Transform operations can be specified by either of the following values: • A list value of two items, first item being the field name and the second item being the transformation function applied to that field to produce a new value. • A list of transformations can be provided by providing a list value, and each item being the list value of 2 items as described above. For examples, go to the description of Record.TransformFields
Rename operations	Rename operations for a record can be specified as either of: A single rename operation, which is represented by a list of two field names, old and new. For examples, go to the description of Record.RenameFields.

Geography.FromWellKnownText

06/19/2025

Syntax

```
Geography.FromWellKnownText(input as nullable text) as nullable record
```

About

Translates text representing a geographic value in Well-Known Text (WKT) format into a structured record. WKT is a standard format defined by the Open Geospatial Consortium (OGC) and is the typical serialization format used by databases including SQL Server.

Geography.ToWellKnownText

06/19/2025

Syntax

```
Geography.ToWellKnownText(input as nullable record, optional omitSRID as nullable logical) as nullable text
```

About

Translates a structured geographic point value into its Well-Known Text (WKT) representation as defined by the Open Geospatial Consortium (OGC), also the serialization format used by many databases including SQL Server.

GeographyPoint.From

06/19/2025

Syntax

```
GeographyPoint.From(longitude as number, latitude as number, optional z as  
nullable number, optional m as nullable number, optional srid as nullable number)  
as record
```

About

Creates a record representing a geographic point from its constituent parts, such as longitude, latitude, and if present, elevation (Z) and measure (M). An optional spatial reference identifier (SRID) can be given if different from the default value (4326).

Geometry.FromWellKnownText

06/19/2025

Syntax

```
Geometry.FromWellKnownText(input as nullable text) as nullable record
```

About

Translates text representing a geometric value in Well-Known Text (WKT) format into a structured record. WKT is a standard format defined by the Open Geospatial Consortium (OGC) and is the typical serialization format used by databases including SQL Server.

Geometry.ToWellKnownText

06/19/2025

Syntax

```
Geometry.ToWellKnownText(input as nullable record, optional omitSRID as nullable logical) as nullable text
```

About

Translates a structured geometric point value into its Well-Known Text (WKT) representation as defined by the Open Geospatial Consortium (OGC), also the serialization format used by many databases including SQL Server.

GeometryPoint.From

06/19/2025

Syntax

```
GeometryPoint.From(x as number, y as number, optional z as nullable number,  
optional m as nullable number, optional srid as nullable number) as record
```

About

Creates a record representing a geometric point from its constituent parts, such as X coordinate, Y coordinate, and if present, Z coordinate and measure (M). An optional spatial reference identifier (SRID) can be given if different from the default value (0).

Record.AddField

06/19/2025

Syntax

```
Record.AddField(record as record, fieldName as text, value as any, optional  
delayed as nullable logical) as record
```

About

Adds a field to a record `record`, given the name of the field `fieldName` and the value `value`.

Example 1

Add the field Address to the record.

Usage

Power Query M

```
Record.AddField([CustomerID = 1, Name = "Bob", Phone = "123-4567"], "Address",  
"123 Main St.")
```

Output

```
[CustomerID = 1, Name = "Bob", Phone = "123-4567", Address = "123 Main St."]
```

Record.Combine

06/19/2025

Syntax

```
Record.Combine(records as list) as record
```

About

Combines the records in the given `records`. If the `records` contains non-record values, an error is returned.

Example 1

Create a combined record from the records.

Usage

Power Query M

```
Record.Combine({  
    [CustomerID = 1, Name = "Bob"],  
    [Phone = "123-4567"]  
})
```

Output

```
[CustomerID = 1, Name = "Bob", Phone = "123-4567"]
```

Record.Field

06/19/2025

Syntax

```
Record.Field(record as record, field as text) as any
```

About

Returns the value of the specified **field** in the **record**. If the field is not found, an exception is thrown.

Example 1

Find the value of field "CustomerID" in the record.

Usage

Power Query M

```
Record.Field([CustomerID = 1, Name = "Bob", Phone = "123-4567"], "CustomerID")
```

Output

1

Record.FieldCount

06/19/2025

Syntax

```
Record.FieldCount(record as record) as number
```

About

Returns the number of fields in the record `record`.

Example 1

Find the number of fields in the record.

Usage

Power Query M

```
Record.FieldCount([CustomerID = 1, Name = "Bob"])
```

Output

2

Record.FieldNames

06/19/2025

Syntax

```
Record.FieldNames(record as record) as list
```

About

Returns the names of the fields in the record `record` as text.

Example 1

Find the names of the fields in the record.

Usage

Power Query M

```
Record.FieldNames([OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0])
```

Output

```
{"OrderID", "CustomerID", "Item", "Price"}
```

Record.FieldOrDefault

06/19/2025

Syntax

```
Record.FieldOrDefault(record as nullable record, field as text, optional defaultValue as any) as any
```

About

Returns the value of the specified field `field` in the record `record`. If the field is not found, the optional `defaultValue` is returned.

Example 1

Find the value of field "Phone" in the record, or return null if it doesn't exist.

Usage

Power Query M

```
Record.FieldOrDefault([CustomerID = 1, Name = "Bob"], "Phone")
```

Output

null

Example 2

Find the value of field "Phone" in the record, or return the default if it doesn't exist.

Usage

Power Query M

```
Record.FieldOrDefault([CustomerID = 1, Name = "Bob"], "Phone", "123-4567")
```

Output

"123-4567"

Record.FieldValues

06/19/2025

Syntax

```
Record.FieldValues(record as record) as list
```

About

Returns a list of the field values in record `record`.

Example 1

Find the field values in the record.

Usage

Power Query M

```
Record.FieldValues([CustomerID = 1, Name = "Bob", Phone = "123-4567"])
```

Output

```
{1, "Bob", "123-4567"}
```

Record.FromList

06/19/2025

Syntax

```
Record.FromList(list as list, fields as any) as record
```

About

Returns a record given a `list` of field values and a set of fields. The `fields` can be specified either by a list of text values, or a record type. An error is thrown if the fields are not unique.

Example 1

Build a record from a list of field values and a list of field names.

Usage

Power Query M

```
Record.FromList({1, "Bob", "123-4567"}, {"CustomerID", "Name", "Phone"})
```

Output

```
[CustomerID = 1, Name = "Bob", Phone = "123-4567"]
```

Example 2

Build a record from a list of field values and a record type.

Usage

Power Query M

```
Record.FromList({1, "Bob", "123-4567"}, type [CustomerID = number, Name = text,  
Phone = number])
```

Output

```
[CustomerID = 1, Name = "Bob", Phone = "123-4567"]
```

Record.FromTable

06/19/2025

Syntax

```
Record.FromTable(table as table) as record
```

About

Returns a record from a table of records `table` containing field names and value names `{[Name = name, Value = value]}`. An exception is thrown if the field names are not unique.

Example 1

Create a record from the table of the form `Table.FromRecords({[Name = "CustomerID", Value = 1], [Name = "Name", Value = "Bob"], [Name = "Phone", Value = "123-4567"]})`.

Usage

Power Query M

```
Record.FromTable(  
    Table.FromRecords(  
        [Name = "CustomerID", Value = 1],  
        [Name = "Name", Value = "Bob"],  
        [Name = "Phone", Value = "123-4567"]  
    )  
)
```

Output

```
[CustomerID = 1, Name = "Bob", Phone = "123-4567"]
```


Record.HasFields

06/19/2025

Syntax

```
Record.HasFields(record as record, fields as any) as logical
```

About

Indicates whether the record `record` has the fields specified in `fields`, by returning a logical value (true or false). Multiple field values can be specified using a list.

Example 1

Check if the record has the field "CustomerID".

Usage

Power Query M

```
Record.HasFields([CustomerID = 1, Name = "Bob", Phone = "123-4567"], "CustomerID")
```

Output

```
true
```

Example 2

Check if the record has the field "CustomerID" and "Address".

Usage

Power Query M

```
Record.HasFields([CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
{ "CustomerID", "Address" })
```

Output

false

Record.RemoveFields

06/19/2025

Syntax

```
Record.RemoveFields(record as record, fields as any, optional missingField as nullable number) as record
```

About

Returns a record that removes all the fields specified in list **fields** from the input **record**. If the field specified does not exist, an exception is thrown.

Example 1

Remove the field "Price" from the record.

Usage

Power Query M

```
Record.RemoveFields([CustomerID = 1, Item = "Fishing rod", Price = 18.00],  
"Price")
```

Output

```
[CustomerID = 1, Item = "Fishing rod"]
```

Example 2

Remove the fields "Price" and "Item" from the record.

Usage

Power Query M

```
Record.RemoveFields([CustomerID = 1, Item = "Fishing rod", Price = 18.00],  
{ "Price", "Item" })
```

Output

```
[CustomerID = 1]
```

Record.RenameFields

06/19/2025

Syntax

```
Record.RenameFields(record as record, renames as list, optional missingField as nullable number) as record
```

About

Returns a record after renaming fields in the input `record` to the new field names specified in list `renames`. For multiple renames, a nested list can be used (`{ {old1, new1}, {old2, new2} }`).

Example 1

Rename the field "UnitPrice" to "Price" from the record.

Usage

Power Query M

```
Record.RenameFields(  
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", UnitPrice = 100.0],  
    {"UnitPrice", "Price"}  
)
```

Output

```
[OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0]
```

Example 2

Rename the fields "UnitPrice" to "Price" and "OrderNum" to "OrderID" from the record.

Usage

Power Query M

```
Record.RenameFields(  
    [OrderNum = 1, CustomerID = 1, Item = "Fishing rod", UnitPrice = 100.0],  
    {
```

```
        {"UnitPrice", "Price"},  
        {"OrderNum", "OrderID"}  
    }  
)
```

Output

```
[OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0]
```

Record.ReorderFields

06/19/2025

Syntax

```
Record.ReorderFields(record as record, fieldOrder as list, optional missingField as nullable number) as record
```

About

Reorders the fields of a record to match the order of a list of field names.

- **record**: The record containing the fields to reorder.
- **fieldOrder**: A list containing the new order of the fields to apply to the record. Field values are maintained and fields not listed in this parameter are left in their original positions.
- **missingField**: Specifies the expected action for missing values in a row that contains fewer fields than expected. The following values are valid:
 - **MissingField.Error**: (Default) Indicates that missing fields should result in an error. If no value is entered for the **missingField** parameter, this value is used.
 - **MissingField.Ignore**: Indicates that missing fields should be ignored.
 - **MissingField.UseNull**: Indicates that missing fields should be included as **null** values.

Example 1

Reorder some of the fields in the record.

Usage

Power Query M

```
Record.ReorderFields(  
    [CustomerID = 1, OrderID = 1, Item = "Fishing rod", Price = 100.0],  
    {"OrderID", "CustomerID"}  
)
```

Output

```
[OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0]
```

Example 2

Reorder some of the fields in the record and include `null` for any missing fields.

Usage

Power Query M

```
let
    Source = [CustomerID = 3, First Name = "Paul", Phone = "543-7890", Purchase = "Fishing Rod"],
    reorderedRecord = Record.ReorderFields(
        Source,
        {"Purchase", "Last Name", "First Name"},
        MissingField.UseNull
    )
in
    reorderedRecord
```

Output

```
[CustomerID = 3, Purchase = "Fishing Rod", Phone = "543-7890", Last Name = null, First Name = "Paul"]
```


Record.SelectFields

06/19/2025

Syntax

```
Record.SelectFields(record as record, fields as any, optional missingField as nullable number) as record
```

About

Returns a record which includes only the fields specified in list `fields` from the input `record`.

Example 1

Select the fields "Item" and "Price" in the record.

Usage

Power Query M

```
Record.SelectFields(  
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],  
    {"Item", "Price"}  
)
```

Output

```
[Item = "Fishing rod", Price = 100]
```

Record.ToList

06/19/2025

Syntax

```
Record.ToList(record as record) as list
```

About

Returns a list of values containing the field values from the input `record`.

Example 1

Extract the field values from a record.

Usage

Power Query M

```
Record.ToList([A = 1, B = 2, C = 3])
```

Output

```
{1, 2, 3}
```

Record.ToTable

06/19/2025

Syntax

```
Record.ToTable(record as record) as table
```

About

Returns a table containing the columns `Name` and `Value` with a row for each field in `record`.

Example 1

Return the table from the record.

Usage

Power Query M

```
Record.ToTable([OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0])
```

Output

Power Query M

```
Table.FromRecords({  
    [Name = "OrderID", Value = 1],  
    [Name = "CustomerID", Value = 1],  
    [Name = "Item", Value = "Fishing rod"],  
    [Name = "Price", Value = 100]  
})
```

Record.TransformFields

06/19/2025

Syntax

```
Record.TransformFields(record as record, transformOperations as list, optional  
missingField as nullable number) as record
```

About

Returns a record after applying transformations specified in list `transformOperations` to `record`. One or more fields may be transformed at a given time.

In the case of a single field being transformed, `transformOperations` is expected to be a list with two items. The first item in `transformOperations` specifies a field name, and the second item in `transformOperations` specifies the function to be used for transformation. For example, `{"Quantity", Number.FromText}`

In the case of a multiple fields being transformed, `transformOperations` is expected to be a list of lists, where each inner list is a pair of field name and transformation operation. For example, `{{"Quantity", Number.FromText}, {"UnitPrice", Number.FromText}}`

Example 1

Convert "Price" field to number.

Usage

Power Query M

```
Record.TransformFields(  
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = "100.0"],  
    {"Price", Number.FromText}  
)
```

Output

```
[OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100]
```

Example 2

Convert "OrderID" and "Price" fields to numbers.

Usage

Power Query M

```
Record.TransformFields(  
    [OrderID = "1", CustomerID = 1, Item = "Fishing rod", Price = "100.0"],  
    {"OrderID", Number.FromText}, {"Price", Number.FromText}  
)
```

Output

```
[OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100]
```

Replacer functions

Article • 08/04/2022

These functions are used by other functions in the library to replace a given value.

Name	Description
Replacer.ReplaceText	This function be provided to List.ReplaceValue or Table.ReplaceValue to do replace of text values in list and table values respectively.
Replacer.ReplaceValue	This function be provided to List.ReplaceValue or Table.ReplaceValue to do replace values in list and table values respectively.

Feedback

Was this page helpful?

 Yes

 No

Get help at [Microsoft Q&A](#)

Replacer.ReplaceText

06/19/2025

Syntax

```
Replacer.ReplaceText(text as nullable text, old as text, new as text) as nullable text
```

About

Replaces the `old` text in the original `text` with the `new` text. This replacer function can be used in `List.ReplaceValue` and `Table.ReplaceValue`.

Example 1

Replace the text "hE" with "He" in the string "hEllo world".

Usage

Power Query M

```
Replacer.ReplaceText("hEllo world", "hE", "He")
```

Output

```
"Hello world"
```

Replacer.ReplaceValue

06/19/2025

Syntax

```
Replacer.ReplaceValue(value as any, old as any, new as any) as any
```

About

Replaces the `old` value in the original `value` with the `new` value. This replacer function can be used in `List.ReplaceValue` and `Table.ReplaceValue`.

Example 1

Replace the value 11 with the value 10.

Usage

```
Power Query M
```

```
Replacer.ReplaceValue(11, 11, 10)
```

Output

```
10
```


Splitter functions

Article • 08/04/2022

These functions split text.

Name	Description
Splitter.SplitByNothing	Returns a function that does no splitting, returning its argument as a single element list.
Splitter.SplitTextByCharacterTransition	Returns a function that splits text into a list of text according to a transition from one kind of character to another.
Splitter.SplitTextByAnyDelimiter	Returns a function that splits text by any supported delimiter.
Splitter.SplitTextByDelimiter	Returns a function that will split text according to a delimiter.
Splitter.SplitTextByEachDelimiter	Returns a function that splits text by each delimiter in turn.
Splitter.SplitTextByLengths	Returns a function that splits text according to the specified lengths.
Splitter.SplitTextByPositions	Returns a function that splits text according to the specified positions.
Splitter.SplitTextByRanges	Returns a function that splits text according to the specified ranges.
Splitter.SplitTextByRepeatedLengths	Returns a function that splits text into a list of text after the specified length repeatedly.
Splitter.SplitTextByWhitespace	Returns a function that splits text according to whitespace.

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Splitter.SplitByNothing

06/19/2025

Syntax

```
Splitter.SplitByNothing() as function
```

About

Returns a function that does no splitting, returning its argument as a single element list.

Splitter.SplitTextByAnyDelimiter

06/19/2025

Syntax

```
Splitter.SplitTextByAnyDelimiter(delimiters as list, optional quoteStyle as nullable number, optional startAtEnd as nullable logical) as function
```

About

Returns a function that splits text into a list of text at any of the specified delimiters.

Example 1

Split the input by comma or semicolon, ignoring quotes and quoted delimiters and starting from the beginning of the input.

Usage

Power Query M

```
Splitter.SplitTextByAnyDelimiter({"", ",", ";"}, QuoteStyle.Csv)("a,b;""c,d;e""",f")
```

Output

```
{"a", "b", "c,d;e", "f"}
```

Example 2

Split the input by comma or semicolon, ignoring quotes and quoted delimiters and starting from the end of the input.

Usage

Power Query M

```
let
    startAtEnd = true
in
```

```
Splitter.SplitTextByAnyDelimiter({"", ";"}, QuoteStyle.Csv, startAtEnd)
("a,""b;c,d")
```

Output

```
{"a,b", "c", "d"}
```

Splitter.SplitTextByCharacterTransition

06/19/2025

Syntax

```
Splitter.SplitTextByCharacterTransition(before as anynonnull, after as anynonnull)  
as function
```

About

Returns a function that splits text into a list of text according to a transition from one kind of character to another. The `before` and `after` parameters can either be a list of characters, or a function that takes a character and returns true/false.

Example 1

Split the input whenever an upper or lowercase letter is followed by a digit.

Usage

Power Query M

```
Splitter.SplitTextByCharacterTransition({"A".."Z", "a".."z"}, {"0".."9"})  
("Abc123")
```

Output

```
{"Abc", "123"}
```

Splitter.SplitTextByDelimiter

06/19/2025

Syntax

```
Splitter.SplitTextByDelimiter(delimiter as text, optional quoteStyle as nullable number, optional csvStyle as nullable number) as function
```

About

Returns a function that splits text into a list of text according to the specified delimiter.

Example 1

Split the input by comma, ignoring quoted commas.

Usage

Power Query M

```
Splitter.SplitTextByDelimiter(",", QuoteStyle.Csv)("a, ""b,c"",d")
```

Output

```
{"a", "b,c", "d"}
```

Splitter.SplitTextByEachDelimiter

06/19/2025

Syntax

```
Splitter.SplitTextByEachDelimiter(delimiters as list, optional quoteStyle as nullable number, optional startAtEnd as nullable logical) as function
```

About

Returns a function that splits text into a list of text at each specified delimiter in sequence.

Example 1

Split the input by comma, then semicolon, starting from the beginning of the input.

Usage

Power Query M

```
Splitter.SplitTextByEachDelimiter({"", ";"})( "a,b;c,d" )
```

Output

```
{"a", "b", "c,d"}
```

Example 2

Split the input by comma, then semicolon, treating quotes like any other character and starting from the end of the input.

Usage

Power Query M

```
let
    startAtEnd = true
in
    Splitter.SplitTextByEachDelimiter({"", ";"}, QuoteStyle.None, startAtEnd)
    ( "a,""b;c"" ,d" )
```

Output

```
{"a","b", "c"", "d"}
```


Splitter.SplitTextByLengths

06/19/2025

Syntax

```
Splitter.SplitTextByLengths(lengths as list, optional startAtEnd as nullable logical) as function
```

About

Returns a function that splits text into a list of text by each specified length.

Example 1

Split the input into the first two characters followed by the next three, starting from the beginning of the input.

Usage

Power Query M

```
Splitter.SplitTextByLengths({2, 3})("AB123")
```

Output

```
{"AB", "123"}
```

Example 2

Split the input into the first three characters followed by the next two, starting from the end of the input.

Usage

Power Query M

```
let  
    startAtEnd = true
```

in

```
Splitter.SplitTextByLengths({5, 2}, startAtEnd)("RedmondWA98052")
```

```
{"WA", "98052"}
```

Splitter.SplitTextByPositions

06/19/2025

Syntax

```
Splitter.SplitTextByPositions(positions as list, optional startAtEnd as nullable logical) as function
```

About

Returns a function that splits text into a list of text at each specified position.

Example 1

Split the input at the specified positions, starting from the beginning of the input.

Usage

Power Query M

```
Splitter.SplitTextByPositions({0, 3, 4})("ABC|12345")
```

Output

```
{"ABC", "|", "12345"}
```

Example 2

Split the input at the specified positions, starting from the end of the input.

Usage

Power Query M

```
let
    startAtEnd = true
in
    Splitter.SplitTextByPositions({0, 5}, startAtEnd)("Redmond98052")
```

Output

```
{"Redmond", "98052"}
```

Splitter.SplitTextByRanges

06/19/2025

Syntax

```
Splitter.SplitTextByRanges(ranges as list, optional startAtEnd as nullable  
logical) as function
```

About

Returns a function that splits text into a list of text according to the specified offsets and lengths. A null length indicates that all remaining input should be included.

Example 1

Split the input by the specified position and length pairs, starting from the beginning of the input. Note that the ranges in this example overlap.

Usage

Power Query M

```
Splitter.SplitTextByRanges({{0, 4}, {2, 10}})("codelimiter")
```

Output

```
{"code", "delimiter"}
```

Example 2

Split the input by the specified position and length pairs, starting from the end of the input.

Usage

Power Query M

```
let  
    startAtEnd = true
```

in

```
Splitter.SplitTextByRanges({{0, 5}, {6, 2}}, startAtEnd)("RedmondWA?98052")
```

Output

```
{"WA", "98052"}
```

Splitter.SplitTextByRepeatedLengths

06/19/2025

Syntax

```
Splitter.SplitTextByRepeatedLengths(length as number, optional startAtEnd as nullable logical) as function
```

About

Returns a function that splits text into a list of text after the specified length repeatedly.

Example 1

Repeatedly split the input into chunks of three characters, starting from the beginning of the input.

Usage

Power Query M

```
Splitter.SplitTextByRepeatedLengths(3)("12345678")
```

Output

```
{"123", "456", "78"}
```

Example 2

Repeatedly split the input into chunks of three characters, starting from the end of the input.

Usage

Power Query M

```
let
    startAtEnd = true
in
    Splitter.SplitTextByRepeatedLengths(3, startAtEnd)("87654321")
```

Output

```
{"87", "654", "321"}
```


Splitter.SplitTextByWhitespace

06/19/2025

Syntax

```
Splitter.SplitTextByWhitespace(optional quoteStyle as nullable number) as function
```

About

Returns a function that splits text into a list of text at whitespace.

Example 1

Split the input by whitespace characters, treating quotes like any other character.

Usage

Power Query M

```
Splitter.SplitTextByWhitespace(QuoteStyle.None)("a b#(tab)c")
```

Output

```
{"a", "b", "c"}
```

Table functions

06/19/2025

These functions create and manipulate table values.

Table construction

 Expand table

Name	Description
#table	Creates a table value from columns and rows.
ItemExpression.From	Returns the abstract syntax tree (AST) for the body of a function.
ItemExpression.Item	An abstract syntax tree (AST) node representing the item in an item expression.
RowExpression.Column	Returns an abstract syntax tree (AST) that represents access to a column within a row expression.
RowExpression.From	Returns the abstract syntax tree (AST) for the body of a function.
RowExpression.Row	An abstract syntax tree (AST) node representing the row in a row expression.
Table.FromColumns	Creates a table from a list of columns and specified values.
Table.FromList	Converts a list into a table by applying the specified splitting function to each item in the list.
Table.FromRecords	Converts a list of records into a table.
Table.FromRows	Creates a table from a list of row values and optional columns.
Table.FromValue	Creates a table with a column from the provided value or values.
Table.WithErrorContext	This function is intended for internal use only.
Table.View	Creates or extends a table with user-defined handlers for query and action operations.
Table.ViewError	Creates a modified error record that won't trigger a fallback when thrown by a handler defined on a view (via Table.View).
Table.ViewFunction	Creates a function that can be intercepted by a handler defined on a view (via Table.View).

Conversions

[Expand table](#)

Name	Description
Table.ToColumns	Creates a list of nested lists of column values from a table.
Table.ToList	Converts a table into a list by applying the specified combining function to each row of values in the table.
Table.ToRecords	Converts a table to a list of records.
Table.ToRows	Creates a list of nested lists of row values from a table.

Information

[Expand table](#)

Name	Description
Table.ApproximateRowCount	Returns the approximate number of rows in the table.
Table.ColumnCount	Returns the number of columns in the table.
Table.IsEmpty	Indicates whether the table contains any rows.
Table.PartitionValues	Returns information about how a table is partitioned.
Table.Profile	Returns a profile of the columns of a table.
Table.RowCount	Returns the number of rows in the table.
Table.Schema	Returns a table containing a description of the columns (that is, the schema) of the specified table.
Tables.GetRelationships	Gets the relationships among a set of tables.

Row operations

[Expand table](#)

Name	Description
Table.AlternateRows	Keeps the initial offset then alternates taking and skipping the following rows.
Table.Combine	Returns a table that is the result of merging a list of tables.

Name	Description
Table.FindText	Returns all the rows that contain the given text in the table.
Table.First	Returns the first row or a specified default value.
Table.FirstN	Returns the first count rows specified.
Table.FirstValue	Returns the first column of the first row of the table or a specified default value.
Table.FromPartitions	Returns a table that is the result of combining a set of partitioned tables.
Table.InsertRows	Inserts a list of rows into the table at the specified position.
Table.Last	Returns the last row or a specified default value.
Table.LastN	Returns the last specified number of rows.
Table.MatchesAllRows	Indicates whether all the rows in the table meet the given condition.
Table.MatchesAnyRows	Indicates whether any the rows in the table meet the given condition.
Table.Partition	Partitions the table into a list of tables based on the number of groups and column specified.
Table.Range	Returns the rows beginning at the specified offset.
Table.RemoveFirstN	Returns a table with the specified number of rows removed from the table starting at the first row.
Table.RemoveLastN	Returns a table with the specified number of rows removed from the table starting at the last row.
Table.RemoveRows	Removes the specified number of rows.
Table.RemoveRowsWithErrors	Returns a table with the rows removed from the input table that contain an error in at least one of the cells. If a columns list is specified, then only the cells in the specified columns are inspected for errors.
Table.Repeat	Repeats the rows of the tables a specified number of times.
Table.ReplaceRows	Replaces the specified range of rows with the provided row or rows.
Table.ReverseRows	Returns a table with the rows in reverse order.
Table.SelectRows	Selects the rows that meet the condition function.
Table.SelectRowsWithErrors	Returns a table with only those rows of the input table that contain an error in at least one of the cells. If a columns list is specified, then only the cells in the specified columns are inspected for errors.
Table.SingleRow	Returns a single row in the table.

Name	Description
Table.Skip	Returns a table with the first specified number of rows skipped.
Table.SplitAt	Returns a list containing the first count rows specified and the remaining rows.

Column operations

 Expand table

Name	Description
Table.Column	Returns a specified column of data from the table as a list.
Table.ColumnNames	Returns the column names as a list.
Table.ColumnsOfType	Returns a list with the names of the columns that match the specified types.
Table.DemoteHeaders	Demotes the column headers to the first row of values.
Table.DuplicateColumn	Duplicates a column with the specified name. Values and type are copied from the source column.
Table.HasColumns	Indicates whether the table contains the specified column or columns.
Table.Pivot	Given a pair of columns representing attribute-value pairs, rotates the data in the attribute column into a column headings.
Table.PrefixColumns	Returns a table where the columns have all been prefixed with the given text.
Table.PromoteHeaders	Promotes the first row of values as the new column headers (that is, as column names).
Table.RemoveColumns	Removes the specified columns.
Table.ReorderColumns	Returns a table with the columns in the specified order.
Table.RenameColumns	Returns a table with the columns renamed as specified.
Table.SelectColumns	Returns a table with only the specified columns.
Table.TransformColumnNames	Transforms column names by using the given function.
Table.Unpivot	Translates a set of columns in a table into attribute-value pairs.
Table.UnpivotOtherColumns	Translates all columns other than a specified set into attribute-value pairs.

Transformation

 Expand table

Name	Description
Table.AddColumn	Adds a column with the specified name. The value is computed using the specified selection function with each row taken as an input.
Table.AddFuzzyClusterColumn	Adds a new column with representative values obtained by fuzzy grouping values of the specified column in the table.
Table.AddIndexColumn	Appends a column with explicit position values.
Table.AddJoinColumn	Performs a join between tables on supplied columns and produces the join result in a new column.
Table.AddKey	Adds a key to a table.
Table.AggregateTableColumn	Aggregates a column of tables into multiple columns in the containing table.
Table.CombineColumns	Combines the specified columns into a new column using the specified combiner function.
Table.CombineColumnsToRecord	Combines the specified columns into a new record-valued column where each record has field names and values corresponding to the column names and values of the columns that were combined.
Table.ConformToPageReader	This function is intended for internal use only.
Table.ExpandListColumn	Given a column of lists in a table, create a copy of a row for each value in its list.
Table.ExpandRecordColumn	Expands a column of records into columns with each of the values.
Table.ExpandTableColumn	Expands a column of records or a column of tables into multiple columns in the containing table.
Table.FillDown	Propagates the value of a previous cell to the null-valued cells below in the column.
Table.FillUp	Propagates the value of a cell to the null-valued cells above in the column.
Table.FilterWithDataTable	This function is intended for internal use only.
Table.FuzzyGroup	Groups rows in the table based on fuzzy matching of keys.
Table.FuzzyJoin	Joins the rows from the two tables that fuzzy match based on the given keys.

Name	Description
Table.FuzzyNestedJoin	Performs a fuzzy join between tables on supplied columns and produces the join result in a new column.
Table.Group	Groups rows in the table that have the same key.
Table.Join	Joins the rows from the two tables that match based on the given keys.
Table.Keys	Returns the keys of the specified table.
Table.NestedJoin	Performs a join between tables on supplied columns and produces the join result in a new column.
Table.ReplaceErrorValues	Replaces the error values in the specified columns with the corresponding specified value.
Table.ReplaceKeys	Replaces the keys of the specified table.
Table.ReplaceRelationshipIdentity	This function is intended for internal use only.
Table.ReplaceValue	Replaces one value with another in the specified columns.
Table.Split	Splits the specified table into a list of tables using the specified page size.
Table.SplitColumn	Splits the specified column into a set of additional columns using the specified splitter function.
Table.TransformColumns	Transforms the values of one or more columns.
Table.TransformColumnTypes	Applies type transformation(s) of the form { column, type } using a specific culture.
Table.TransformRows	Transforms the rows of the table using the specified transform function.
Table.Transpose	Makes columns into rows and rows into columns.

Membership

 Expand table

Name	Description
Table.Contains	Indicates whether the specified record appears as a row in the table.
Table.ContainsAll	Indicates whether all of the specified records appear as rows in the table.

Name	Description
Table.ContainsAny	Indicates whether any of the specified records appear as rows in the table.
Table.Distinct	Removes duplicate rows from the table.
Table.IsDistinct	Indicates whether the table contains only distinct rows (no duplicates).
Table.PositionOf	Returns the position or positions of the row within the table.
Table.PositionOfAny	Returns the position or positions of any of the specified rows within the table.
Table.RemoveMatchingRows	Removes all occurrences of the specified rows from the table.
Table.ReplaceMatchingRows	Replaces all the specified rows with the provided row or rows.

Ordering

 Expand table

Name	Description
Table.AddRankColumn	Appends a column with the ranking of one or more other columns.
Table.Max	Returns the largest row or default value using the given criteria.
Table.MaxN	Returns the largest row or rows using the given criteria.
Table.Min	Returns the smallest row or a default value using the given criteria.
Table.MinN	Returns the smallest row or rows using the given criteria.
Table.Sort	Sorts the table using one or more column names and comparison criteria.

Other

 Expand table

Name	Description
Table.Buffer	Buffers a table in memory, isolating it from external changes during evaluation.
Table.StopFolding	Prevents any downstream operations from being run against the original source of the data.

Parameter Values

Naming output columns

This parameter is a list of text values specifying the column names of the resulting table. This parameter is generally used in the [Table construction functions](#), such as [Table.FromRows](#) and [Table.FromList](#).

Comparison criteria

Comparison criterion can be provided as either of the following values:

- A number value to specify a sort order. More information: [Sort order](#)
- To compute a key to be used for sorting, a function of one argument can be used.
- To both select a key and control order, comparison criterion can be a list containing the key and order.
- To completely control the comparison, a function of two arguments can be used that returns -1, 0, or 1 given the relationship between the left and right inputs. [Value.Compare](#) can be used to delegate this logic.

For examples, go to the description of [Table.Sort](#).

Count or Condition criteria

This criteria is generally used in ordering or row operations. It determines the number of rows returned in the table and can take two forms, a number or a condition.

- A number indicates how many values to return inline with the appropriate function.
- If a condition is specified, the rows containing values that initially meet the condition is returned. Once a value fails the condition, no further values are considered.

More information: [Table.FirstN](#), [Table.MaxN](#)

Handling of extra values

Extra values are used to indicate how the function should handle extra values in a row. This parameter is specified as a number, which maps to the following options:

```
ExtraValues.List = 0
```

```
ExtraValues.Error = 1
```

```
ExtraValues.Ignore = 2
```

More information: [Table.FromList](#), [ExtraValues.Type](#)

Missing column handling

This parameter is used to indicate how the function should handle missing columns. This parameter is specified as a number, which maps to the following options:

```
MissingField.Error = 0
```

```
MissingField.Ignore = 1
```

```
MissingField.UseNull = 2;
```

This parameter is used in column or transformation operations, for examples, in [Table.TransformColumns](#). More information: [MissingField.Type](#)

Sort Order

Sort ordering is used to indicate how the results should be sorted. This parameter is specified as a number, which maps to the following options:

```
Order.Ascending = 0
```

```
Order.Descending = 1
```

More information: [Order.Type](#)

Equation criteria

Equation criteria for tables can be specified as either:

- A function value that is either:
 - A key selector that determines the column in the table to apply the equality criteria.
 - A comparer function that is used to specify the kind of comparison to apply. Built-in comparer functions can be specified. More information: [Comparer functions](#)
- A list of the columns in the table to apply the equality criteria.

For examples, go to the description of [Table.Distinct](#).

ItemExpression.From

06/19/2025

Syntax

```
ItemExpression.From(function as function) as record
```

About

Returns the abstract syntax tree (AST) for the body of `function`, normalized into an *item expression*:

- The function must be a 1-argument lambda.
- All references to the function parameter are replaced with [ItemExpression.Item](#).
- The AST will be simplified to contain only nodes of the kinds:
 - `Constant`
 - `Invocation`
 - `Unary`
 - `Binary`
 - `If`
 - `FieldAccess`

An error is raised if an item expression AST cannot be returned for the body of `function`.

This function is identical to [RowExpression.From](#).

Example 1

Returns the AST for the body of the function `each _ <> null`.

Usage

Power Query M

```
ItemExpression.From(each _ <> null)
```

Output

Power Query M

```
[
  Kind = "Binary",
  Operator = "NotEquals",
  Left = ItemExpression.Item,
  Right =
  [
    Kind = "Constant",
    Value = null
  ]
]
```

ItemExpression.Item

06/19/2025

About

An abstract syntax tree (AST) node representing the item in an item expression.

This value is identical to [RowExpression.Row](#).

RowExpression.Column

06/19/2025

Syntax

```
RowExpression.Column(columnName as text) as record
```

About

Returns an abstract syntax tree (AST) that represents access to column `columnName` of the row within a row expression.

Example 1

Creates an AST representing access of column "CustomerName".

Usage

Power Query M

```
RowExpression.Column("CustomerName")
```

Output

Power Query M

```
[
  Kind = "FieldAccess",
  Expression = RowExpression.Row,
  MemberName = "CustomerName"
]
```

RowExpression.From

06/19/2025

Syntax

```
RowExpression.From(function as function) as record
```

About

Returns the abstract syntax tree (AST) for the body of `function`, normalized into a *row expression*:

- The function must be a 1-argument lambda.
- All references to the function parameter are replaced with [RowExpression.Row](#).
- All references to columns are replaced with [RowExpression.Column\(columnName\)](#).
- The AST will be simplified to contain only nodes of the kinds:
 - `Constant`
 - `Invocation`
 - `Unary`
 - `Binary`
 - `If`
 - `FieldAccess`

An error is raised if a row expression AST cannot be returned for the body of `function`.

This function is identical to [ItemExpression.From](#).

Example 1

Returns the AST for the body of the function `each [CustomerID] = "ALFKI"`.

Usage

Power Query M

```
RowExpression.From(each [CustomerName] = "ALFKI")
```

Output

Power Query M

```
[
    Kind = "Binary",
    Operator = "Equals",
    Left = RowExpression.Column("CustomerName"),
    Right =
    [
        Kind = "Constant",
        Value = "ALFKI"
    ]
]
```


RowExpression.Row

06/19/2025

About

An abstract syntax tree (AST) node representing the row in a row expression.

This value is identical to [ItemExpression.Item](#).

Table.AddColumn

06/19/2025

Syntax

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

About

Adds a column named `newColumnName` to the table `table`. The values for the column are computed using the specified selection function `columnGenerator` with each row taken as an input.

Example 1

Add a number column named "TotalPrice" to the table, with each value being the sum of the [Price] and [Shipping] columns.

Usage

Power Query M

```
Table.AddColumn(  
    Table.FromRecords({  
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0,  
Shipping = 10.00],  
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0, Shipping  
= 15.00],  
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0, Shipping  
= 10.00]  
    }),  
    "TotalPrice",  
    each [Price] + [Shipping],  
    type number  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100, Shipping =  
10, TotalPrice = 110],  
    [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5, Shipping = 15,  
TotalPrice = 20],  
    [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25, Shipping = 10,  
TotalPrice = 35]  
})
```

Related content

- [Types and type conversion](#)

Table.AddFuzzyClusterColumn

06/19/2025

Syntax

```
Table.AddFuzzyClusterColumn(table as table, columnName as text, newColumnName as text, optional options as nullable record) as table
```

About

Adds a new column `newColumnName` to `table` with representative values of `columnName`. The representatives are obtained by fuzzily matching values in `columnName`, for each row.

An optional set of `options` may be included to specify how to compare the key columns.

Options include:

- `Culture`: Allows grouping records based on culture-specific rules. It can be any valid culture name. For example, a Culture option of "ja-JP" groups records based on the Japanese culture. The default value is "", which groups based on the Invariant English culture.
- `IgnoreCase`: A logical (true/false) value that allows case-insensitive key grouping. For example, when true, "Grapes" is grouped with "grapes". The default value is true.
- `IgnoreSpace`: A logical (true/false) value that allows combining of text parts in order to find groups. For example, when true, "Gra pes" is grouped with "Grapes". The default value is true.
- `SimilarityColumnName`: A name for the column that shows the similarity between an input value and the representative value for that input. The default value is null, in which case a new column for similarities will not be added.
- `Threshold`: A number between 0.00 and 1.00 that specifies the similarity score at which two values will be grouped. For example, "Grapes" and "Graes" (missing the "p") are grouped together only if this option is set to less than 0.90. A threshold of 1.00 only allows exact matches. (Note that a fuzzy "exact match" might ignore differences like casing, word order, and punctuation.) The default value is 0.80.
- `TransformationTable`: A table that allows grouping records based on custom value mappings. It should contain "From" and "To" columns. For example, "Grapes" is grouped with "Raisins" if a transformation table is provided with the "From" column containing "Grapes" and the "To" column containing "Raisins". Note that the transformation will be

applied to all occurrences of the text in the transformation table. With the above transformation table, "Grapes are sweet" will also be grouped with "Raisins are sweet".

Example 1

Find the representative values for the location of the employees.

Usage

Power Query M

```
Table.AddFuzzyClusterColumn(  
    Table.FromRecords(  
        {  
            [EmployeeID = 1, Location = "Seattle"],  
            [EmployeeID = 2, Location = "seattl"],  
            [EmployeeID = 3, Location = "Vancouver"],  
            [EmployeeID = 4, Location = "Seatle"],  
            [EmployeeID = 5, Location = "vancouver"],  
            [EmployeeID = 6, Location = "Seattle"],  
            [EmployeeID = 7, Location = "Vancouver"]  
        },  
        type table [EmployeeID = nullable number, Location = nullable text]  
    ),  
    "Location",  
    "Location_Cleaned",  
    [IgnoreCase = true, IgnoreSpace = true]  
)
```

Output

Power Query M

```
Table.FromRecords(  
    {  
        [EmployeeID = 1, Location = "Seattle", Location_Cleaned = "Seattle"],  
        [EmployeeID = 2, Location = "seattl", Location_Cleaned = "Seattle"],  
        [EmployeeID = 3, Location = "Vancouver", Location_Cleaned = "Vancouver"],  
        [EmployeeID = 4, Location = "Seatle", Location_Cleaned = "Seattle"],  
        [EmployeeID = 5, Location = "vancouver", Location_Cleaned = "Vancouver"],  
        [EmployeeID = 6, Location = "Seattle", Location_Cleaned = "Seattle"],  
        [EmployeeID = 7, Location = "Vancouver", Location_Cleaned = "Vancouver"]  
    },  
    type table [EmployeeID = nullable number, Location = nullable text,  
        Location_Cleaned = nullable text]  
)
```

Related content

- How culture affects text formatting

Table.AddIndexColumn

06/19/2025

Syntax

```
Table.AddIndexColumn(table as table, newColumnName as text, optional initialValue as nullable number, optional increment as nullable number, optional columnType as nullable type) as table
```

About

Appends a column named `newColumnName` to the `table` with explicit position values. An optional value, `initialValue`, the initial index value. An optional value, `increment`, specifies how much to increment each index value.

Example 1

Add an index column named "Index" to the table.

Usage

Power Query M

```
Table.AddIndexColumn(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Index"
)
```

Output

Power Query M

```
Table.FromRecords({
    [CustomerID = 1, Name = "Bob", Phone = "123-4567", Index = 0],
    [CustomerID = 2, Name = "Jim", Phone = "987-6543", Index = 1],
    [CustomerID = 3, Name = "Paul", Phone = "543-7890", Index = 2],
```

```
[CustomerID = 4, Name = "Ringo", Phone = "232-1550", Index = 3]
})
```

Example 2

Add an index column named "index", starting at value 10 and incrementing by 5, to the table.

Usage

Power Query M

```
Table.AddIndexColumn(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Index",
    10,
    5
)
```

Output

Power Query M

```
Table.FromRecords({
    [CustomerID = 1, Name = "Bob", Phone = "123-4567", Index = 10],
    [CustomerID = 2, Name = "Jim", Phone = "987-6543", Index = 15],
    [CustomerID = 3, Name = "Paul", Phone = "543-7890", Index = 20],
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550", Index = 25]
})
```

Related content

- [Types and type conversion](#)

Table.AddJoinColumn

06/19/2025

Syntax

```
Table.AddJoinColumn(table1 as table, key1 as any, table2 as function, key2 as any,
newColumnName as text) as table
```

About

Joins the rows of `table1` with the rows of `table2` based on the equality of the values of the key columns selected by `key1` (for `table1`) and `key2` (for `table2`). The results are entered into the column named `newColumnName`. This function behaves similarly to [Table.Join](#) with a `JoinKind` of `LeftOuter` except that the join results are presented in a nested rather than flattened fashion.

Example 1

Add a join column to `({[saleID = 1, item = "Shirt"], [saleID = 2, item = "Hat"]})` named "price/stock" from the table `({[saleID = 1, price = 20], [saleID = 2, price = 10]})` joined on `[saleID]`.

Usage

Power Query M

```
Table.AddJoinColumn(
    Table.FromRecords({
        [saleID = 1, item = "Shirt"],
        [saleID = 2, item = "Hat"]
    }),
    "saleID",
    () => Table.FromRecords({
        [saleID = 1, price = 20, stock = 1234],
        [saleID = 2, price = 10, stock = 5643]
    }),
    "saleID",
    "price"
)
```

Output

```
Table.FromRecords({  
    [  
        saleID = 1,  
        item = "Shirt",  
        price = Table.FromRecords({[saleID = 1, price = 20, stock = 1234]})  
    ],  
    [  
        saleID = 2,  
        item = "Hat",  
        price = Table.FromRecords({[saleID = 2, price = 10, stock = 5643]})  
    ]  
})
```

Table.AddKey

06/19/2025

Syntax

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

About

Adds a key to `table`, where `columns` is the list of column names that define the key, and `isPrimary` specifies whether the key is primary.

Example 1

Add a single-column primary key to a table.

Usage

Power Query M

```
let
    table = Table.FromRecords({
        [Id = 1, Name = "Hello There"],
        [Id = 2, Name = "Good Bye"]
    }),
    resultTable = Table.AddKey(table, {"Id"}, true)
in
    resultTable
```

Output

Power Query M

```
Table.FromRecords({
    [Id = 1, Name = "Hello There"],
    [Id = 2, Name = "Good Bye"]
})
```

Table.AddRankColumn

06/19/2025

Syntax

```
Table.AddRankColumn(table as table, newColumnName as text, comparisonCriteria as any, optional options as nullable record) as table
```

About

Appends a column named `newColumnName` to the `table` with the ranking of one or more other columns described by `comparisonCriteria`. The `RankKind` option in `options` can be used by advanced users to pick a more-specific ranking method.

Example 1

Add a column named **RevenueRank** to the table which ranks the **Revenue** column from highest to lowest.

Usage

Power Query M

```
Table.AddRankColumn(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Revenue = 200],
        [CustomerID = 2, Name = "Jim", Revenue = 100],
        [CustomerID = 3, Name = "Paul", Revenue = 200],
        [CustomerID = 4, Name = "Ringo", Revenue = 50]
    }),
    "RevenueRank",
    {"Revenue", Order.Descending},
    [RankKind = RankKind.Competition]
)
```

Output

Power Query M

```
Table.FromRecords({
    [CustomerID = 1, Name = "Bob", Revenue = 200, RevenueRank = 1],
    [CustomerID = 3, Name = "Paul", Revenue = 200, RevenueRank = 1],
    [CustomerID = 2, Name = "Jim", Revenue = 100, RevenueRank = 2],
    [CustomerID = 4, Name = "Ringo", Revenue = 50, RevenueRank = 3]
})
```

```
[CustomerID = 2, Name = "Jim", Revenue = 100, RevenueRank = 3],  
[CustomerID = 4, Name = "Ringo", Revenue = 50, RevenueRank = 4]  
})
```

Related content

[Comparison criteria](#)

Table.AggregateTableColumn

06/19/2025

Syntax

```
Table.AggregateTableColumn(table as table, column as text, aggregations as list)  
as table
```

About

Aggregates tables in `table[column]` into multiple columns containing aggregate values for the tables. `aggregations` is used to specify the columns containing the tables to aggregate, the aggregation functions to apply to the tables to generate their values, and the names of the aggregate columns to create.

Example 1

Aggregate table columns in `[t]` in the table `{[t = {[a=1, b=2, c=3], [a=2,b=4,c=6]}, b = 2]}` into the sum of `[t.a]`, the min and max of `[t.b]`, and the count of values in `[t.a]`.

Usage

Power Query M

```
Table.AggregateTableColumn(  
    Table.FromRecords(  
        {  
            [  
                t = Table.FromRecords({  
                    [a = 1, b = 2, c = 3],  
                    [a = 2, b = 4, c = 6]  
                }),  
                b = 2  
            ]  
        },  
        type table [t = table [a = number, b = number, c = number], b = number]  
    ),  
    "t",  
    {  
        {"a", List.Sum, "sum of t.a"},  
        {"b", List.Min, "min of t.b"},  
        {"b", List.Max, "max of t.b"},  
        {"a", List.Count, "count of t.a"}  
    }  
)
```

```
}  
)
```

Output

```
Table.FromRecords({[#"sum of t.a" = 3, #"min of t.b" = 2, #"max of t.b" = 4, #"count of  
t.a" = 2, b = 2]})
```

Table.AlternateRows

06/19/2025

Syntax

```
Table.AlternateRows(table as table, offset as number, skip as number, take as number) as table
```

About

Keeps the initial offset then alternates taking and skipping the following rows.

- **table**: The input table.
- **offset**: The number of rows to keep before starting iterations.
- **skip**: The number of rows to remove in each iteration.
- **take**: The number of rows to keep in each iteration.

Example 1

Return a table from the table that, starting at the first row, skips 1 value and then keeps 1 value.

Usage

Power Query M

```
Table.AlternateRows(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
    }),  
    1,  
    1,  
    1  
)
```

Output

Power Query M


```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
})
```

Table.ApproximateRowCount

06/19/2025

Syntax

```
Table.ApproximateRowCount(table as table) as number
```

About

Returns the approximate number of rows in the `table`, or an error if the data source doesn't support approximation.

Example 1

Estimate the number of distinct combinations of city and state in a large table, which can be used as a cardinality estimate for the columns. Cardinality estimates are important enough that various data sources (such as SQL Server) support this particular approximation, often using an algorithm called HyperLogLog.

Usage

Power Query M

```
Table.ApproximateRowCount(Table.Distinct(Table.SelectColumns(sqlTable, {"city",  
"state"})))
```

Output

number

Table.Buffer

06/19/2025

Syntax

```
Table.Buffer(table as table, optional options as nullable record) as table
```

About

Buffers a table in memory, isolating it from external changes during evaluation. Buffering is shallow. It forces the evaluation of any scalar cell values, but leaves non-scalar values (records, lists, tables, and so on) as-is.

- **table**: The table to buffer in memory.
- **options**: [Optional] The following options record values can be used:
 - **BufferMode**: The buffer mode that describes the type of buffering to be performed. This option can be either [BufferMode.Eager](#) or [BufferMode.Delayed](#).

Using this function might or might not make your queries run faster. In some cases, it can make your queries run more slowly due to the added cost of reading all the data and storing it in memory, as well as the fact that buffering prevents downstream folding. If the data doesn't need to be buffered but you just want to prevent downstream folding, use [Table.StopFolding](#) instead.

Example 1

Load all the rows of a SQL table into memory, so that any downstream operations are no longer able to query the SQL server.

Usage

Power Query M

```
let
    Source = Sql.Database("SomeSQLServer", "MyDb"),
    MyTable = Source{[Item="MyTable"]}[Data],
    BufferMyTable = Table.Buffer(MyTable)
in
    BufferMyTable
```

Output

Power Query M
table

Table.Column

06/19/2025

Syntax

```
Table.Column(table as table, column as text) as list
```

About

Returns the column of data specified by `column` from the table `table` as a list.

Example 1

Returns the values from the [Name] column in the table.

Usage

Power Query M

```
Table.Column(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    "Name"  
)
```

Output

```
{"Bob", "Jim", "Paul", "Ringo"}
```

Table.ColumnCount

06/19/2025

Syntax

```
Table.ColumnCount(table as table) as number
```

About

Returns the number of columns in the table `table`.

Example 1

Find the number of columns in the table.

Usage

Power Query M

```
Table.ColumnCount(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
    })  
)
```

Output

3

Table.ColumnNames

06/19/2025

Syntax

```
Table.ColumnNames(table as table) as list
```

About

Returns the column names in the table `table` as a list of text.

Example 1

Find the column names of the table.

Usage

Power Query M

```
Table.ColumnNames(  
    Table.FromRecords(  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    )  
)
```

Output

```
{"CustomerID", "Name", "Phone"}
```

Table.ColumnsOfType

06/19/2025

Syntax

```
Table.ColumnsOfType(table as table, listOfTypes as list) as list
```

About

Returns a list with the names of the columns from table `table` that match the types specified in `listOfTypes`.

Example 1

Return the names of columns of type `Number.Type` from the table.

Usage

Power Query M

```
Table.ColumnsOfType(  
    Table.FromRecords(  
        {[a = 1, b = "hello"]},  
        type table[a = Number.Type, b = Text.Type]  
    ),  
    {type number}  
)
```

Output

```
{"a"}
```

Related content

- [Types and type conversion](#)

Table.Combine

06/19/2025

Syntax

```
Table.Combine(tables as list, optional columns as any) as table
```

About

Returns a table that is the result of merging a list of tables, `tables`. The resulting table will have a row type structure defined by `columns` or by a union of the input types if `columns` is not specified.

Example 1

Merge the three tables together.

Usage

Power Query M

```
Table.Combine({  
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),  
    Table.FromRecords({[CustomerID = 2, Name = "Jim", Phone = "987-6543"]}),  
    Table.FromRecords({[CustomerID = 3, Name = "Paul", Phone = "543-7890"]})  
})
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
})
```

Example 2

Merge three tables with different structures.

Usage

Power Query M

```
Table.Combine({
    Table.FromRecords({[Name = "Bob", Phone = "123-4567"]}),
    Table.FromRecords({[Fax = "987-6543", Phone = "838-7171"]}),
    Table.FromRecords({[Cell = "543-7890"]})
})
```

Output

Power Query M

```
Table.FromRecords({
    [Name = "Bob", Phone = "123-4567", Fax = null, Cell = null],
    [Name = null, Phone = "838-7171", Fax = "987-6543", Cell = null],
    [Name = null, Phone = null, Fax = null, Cell = "543-7890"]
})
```

Example 3

Merge two tables and project onto the given type.

Usage

Power Query M

```
Table.Combine(
    {
        Table.FromRecords({[Name = "Bob", Phone = "123-4567"]}),
        Table.FromRecords({[Fax = "987-6543", Phone = "838-7171"]}),
        Table.FromRecords({[Cell = "543-7890"]})
    },
    {"CustomerID", "Name"}
)
```

Output

Power Query M

```
Table.FromRecords({
    [CustomerID = null, Name = "Bob"],
    [CustomerID = null, Name = null],
    [CustomerID = null, Name = null]
})
```


Table.CombineColumns

06/19/2025

Syntax

```
Table.CombineColumns(table as table, sourceColumns as list, combiner as function,  
column as text) as table
```

About

Combines the specified columns into a new column using the specified combiner function.

Example 1

Combine the last and first names into a new column, separated by a comma.

Usage

Power Query M

```
Table.CombineColumns(  
    Table.FromRecords({[FirstName = "Bob", LastName = "Smith"]}),  
    {"LastName", "FirstName"},  
    Combiner.CombineTextByDelimiter(",", QuoteStyle.None),  
    "FullName"  
)
```

Output

Power Query M

```
Table.FromRecords({[FullName = "Smith,Bob"]})
```

Table.CombineColumnsToRecord

06/19/2025

Syntax

```
Table.CombineColumnsToRecord(table as table, newColumnName as text, sourceColumns as list, optional options as nullable record) as table
```

About

Combines the specified columns of `table` into a new record-valued column named `newColumnName` where each record has field names and values corresponding to the column names and values of the columns that were combined. If a record is specified for `options`, the following options may be provided:

- `DisplayNameColumn`: When specified as text, indicates that the given column name should be treated as the display name of the record. This need not be one of the columns in the record itself.
- `TypeName`: When specified as text, supplies a logical type name for the resulting record which can be used during data load to drive behavior by the loading environment.

Related content

- [Types and type conversion](#)

Table.ConformToPageReader

06/19/2025

Syntax

```
Table.ConformToPageReader(table as table, shapingFunction as function) as table
```

About

This function is intended for internal use only.

Table.Contains

06/19/2025

Syntax

```
Table.Contains(table as table, row as record, optional equationCriteria as any) as logical
```

About

Indicates whether the specified record, `row`, appears as a row in the `table`. An optional parameter `equationCriteria` may be specified to control comparison between the rows of the table.

Example 1

Determine if the table contains the row.

Usage

Power Query M

```
Table.Contains(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    [Name = "Bob"]  
)
```

Output

```
true
```

Example 2

Determine if the table contains the row.

Usage

```
Table.Contains(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    [Name = "Ted"]  
)
```

Output

```
false
```

Example 3

Determine if the table contains the row comparing only the column [Name].

Usage

```
Table.Contains(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    [CustomerID = 4, Name = "Bob"],  
    "Name"  
)
```

Output

```
true
```

Related content

[Equation criteria](#)

Table.ContainsAll

06/19/2025

Syntax

```
Table.ContainsAll(table as table, rows as list, optional equationCriteria as any)  
as logical
```

About

Indicates whether all the specified records in the list of records **rows**, appear as rows in the **table**. An optional parameter **equationCriteria** may be specified to control comparison between the rows of the table.

Example 1

Determine if the table contains all the rows, comparing only the column [CustomerID].

Usage

Power Query M

```
Table.ContainsAll(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    {  
        [CustomerID = 1, Name = "Bill"],  
        [CustomerID = 2, Name = "Fred"]  
    },  
    "CustomerID"  
)
```

Output

true

Example 2

Determine if the table contains all the rows.

Usage

Power Query M

```
Table.ContainsAll(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    {  
        [CustomerID = 1, Name = "Bill"],  
        [CustomerID = 2, Name = "Fred"]  
    }  
)
```

Output

false

Related content

[Equation criteria](#)

Table.ContainsAny

06/19/2025

Syntax

```
Table.ContainsAny(table as table, rows as list, optional equationCriteria as any)  
as logical
```

About

Indicates whether any the specified records in the list of records **rows**, appear as rows in the **table**. An optional parameter **equationCriteria** may be specified to control comparison between the rows of the table.

Example 1

Determine if the table `({[a = 1, b = 2], [a = 3, b = 4]})` contains the rows `[a = 1, b = 2]` or `[a = 3, b = 5]`.

Usage

Power Query M

```
Table.ContainsAny(  
    Table.FromRecords({  
        [a = 1, b = 2],  
        [a = 3, b = 4]  
    }),  
    {  
        [a = 1, b = 2],  
        [a = 3, b = 5]  
    }  
)
```

Output

true

Example 2

Determine if the table `({[a = 1, b = 2], [a = 3, b = 4]})` contains the rows `[a = 1, b = 3]` or `[a = 3, b = 5]`.

Usage

Power Query M

```
Table.ContainsAny(  
    Table.FromRecords(  
        [a = 1, b = 2],  
        [a = 3, b = 4]  
    )),  
    {  
        [a = 1, b = 3],  
        [a = 3, b = 5]  
    }  
)
```

Output

false

Example 3

Determine if the table `(Table.FromRecords({[a = 1, b = 2], [a = 3, b = 4]}))` contains the rows `[a = 1, b = 3]` or `[a = 3, b = 5]` comparing only the column `[a]`.

Usage

Power Query M

```
Table.ContainsAny(  
    Table.FromRecords(  
        [a = 1, b = 2],  
        [a = 3, b = 4]  
    )),  
    {  
        [a = 1, b = 3],  
        [a = 3, b = 5]  
    },  
    "a"  
)
```

Output

true

Related content

[Equation criteria](#)

Table.DemoteHeaders

06/19/2025

Syntax

```
Table.DemoteHeaders(table as table) as table
```

About

Demotes the column headers (i.e. column names) to the first row of values. The default column names are "Column1", "Column2" and so on.

Example 1

Demote the first row of values in the table.

Usage

Power Query M

```
Table.DemoteHeaders(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"]  
    })  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [Column1 = "CustomerID", Column2 = "Name", Column3 = "Phone"],  
    [Column1 = 1, Column2 = "Bob", Column3 = "123-4567"],  
    [Column1 = 2, Column2 = "Jim", Column3 = "987-6543"]  
})
```

Table.Distinct

06/19/2025

Syntax

```
Table.Distinct(table as table, optional equationCriteria as any) as table
```

About

Removes duplicate rows from the table. An optional parameter, `equationCriteria`, specifies which columns of the table are tested for duplication. If `equationCriteria` is not specified, all columns are tested.

Because Power Query sometimes offloads certain operations to backend data sources (known as *folding*), and also sometimes optimizes queries by skipping operations that aren't strictly necessary, in general there's no guarantee which specific duplicate will be preserved. For example, you can't assume that the first row with a unique set of column values will remain, and rows further down in the table will be removed. If you want the duplicate removal to behave predictably, first buffer the table using [Table.Buffer](#).

Example 1

Remove the duplicate rows from the table.

Usage

Power Query M

```
Table.Distinct(  
    Table.FromRecords({  
        [a = "A", b = "a"],  
        [a = "B", b = "b"],  
        [a = "A", b = "a"]  
    })  
)
```

Output

Power Query M

```
Table.FromRecords({
    [a = "A", b = "a"],
    [a = "B", b = "b"]
})
```

Example 2

Remove the duplicate rows from column [b] in the table `({[a = "A", b = "a"], [a = "B", b = "a"], [a = "A", b = "b"]})`.

Usage

Power Query M

```
Table.Distinct(
    Table.FromRecords({
        [a = "A", b = "a"],
        [a = "B", b = "a"],
        [a = "A", b = "b"]
    }),
    "b"
)
```

Output

Power Query M

```
Table.FromRecords({
    [a = "A", b = "a"],
    [a = "A", b = "b"]
})
```

Related content

[Equation criteria](#)

Table.DuplicateColumn

06/19/2025

Syntax

```
Table.DuplicateColumn(table as table, columnName as text, newColumnName as text,  
optional columnType as nullable type) as table
```

About

Duplicate the column named `columnName` to the table `table`. The values and type for the column `newColumnName` are copied from column `columnName`.

Example

Duplicate the column "a" to a column named "copied column" in the table `([a = 1, b = 2], [a = 3, b = 4])`.

Usage

Power Query M

```
Table.DuplicateColumn(  
    Table.FromRecords({  
        [a = 1, b = 2],  
        [a = 3, b = 4]  
    }),  
    "a",  
    "copied column"  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [a = 1, b = 2, #"copied column" = 1],  
    [a = 3, b = 4, #"copied column" = 3]  
})
```

Related content

- [Types and type conversion](#)

Table.ExpandListColumn

06/19/2025

Syntax

```
Table.ExpandListColumn(table as table, column as text) as table
```

About

Given a `table` where `column` contains a list of values, splits the list into a row for each value. Values in the other columns are duplicated in each new row created. This function can also expand nested tables by treating them as lists of records.

Example 1

Split the list column [Name].

Usage

Power Query M

```
Table.ExpandListColumn(  
    Table.FromRecords({[Name = {"Bob", "Jim", "Paul"}, Discount = .15]}),  
    "Name"  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [Name = "Bob", Discount = 0.15],  
    [Name = "Jim", Discount = 0.15],  
    [Name = "Paul", Discount = 0.15]  
})
```

Example 2

Split the nested table column [Components].

Usage

Power Query M

```
Table.ExpandListColumn(  
    #table(  
        {"Part", "Components"},  
        {  
            {"Tool", #table({"Name", "Quantity"}, {"Thingamajig", 2}, {"Widget",  
3}})}  
        }  
    ),  
    "Components"  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [Part = "Tool", Components = [Name = "Thingamajig", Quantity = 2]],  
    [Part = "Tool", Components = [Name = "Widget", Quantity = 3]]  
})
```

Table.ExpandRecordColumn

06/19/2025

Syntax

```
Table.ExpandRecordColumn(table as table, column as text, fieldNames as list,  
optional newColumnNames as nullable list) as table
```

About

Given the `column` of records in the input `table`, creates a table with a column for each field in the record. Optionally, `newColumnNames` may be specified to ensure unique names for the columns in the new table.

- `table`: The original table with the record column to expand.
- `column`: The column to expand.
- `fieldNames`: The list of fields to expand into columns in the table.
- `newColumnNames`: The list of column names to give the new columns. The new column names cannot duplicate any column in the new table.

Example 1

Expand column [a] in the table `{{[a = [aa = 1, bb = 2, cc = 3], b = 2]}}` into 3 columns "aa", "bb" and "cc".

Usage

Power Query M

```
Table.ExpandRecordColumn(  
    Table.FromRecords(  
        [  
            a = [aa = 1, bb = 2, cc = 3],  
            b = 2  
        ]  
    ),  
    "a",  
    {"aa", "bb", "cc"}  
)
```

Output

```
Table.FromRecords({[aa = 1, bb = 2, cc = 3, b = 2]})
```

Table.ExpandTableColumn

06/19/2025

Syntax

```
Table.ExpandTableColumn(table as table, column as text, columnNames as list,  
optional newColumnNames as nullable list) as table
```

About

Expands tables in `table[column]` into multiple rows and columns. `columnNames` is used to select the columns to expand from the inner table. Specify `newColumnNames` to avoid conflicts between existing columns and new columns.

Example 1

Expand table columns in `[a]` in the table `({[t = {[a=1, b=2, c=3], [a=2,b=4,c=6]}, b = 2])` into 3 columns `[t.a]`, `[t.b]` and `[t.c]`.

Usage

Power Query M

```
Table.ExpandTableColumn(  
    Table.FromRecords(  
        [  
            t = Table.FromRecords(  
                [a = 1, b = 2, c = 3],  
                [a = 2, b = 4, c = 6]  
            )),  
        b = 2  
    ]  
    ),  
    "t",  
    {"a", "b", "c"},  
    {"t.a", "t.b", "t.c"}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [t.a = 1, t.b = 2, t.c = 3, b = 2],  
    [t.a = 2, t.b = 4, t.c = 6, b = 2]  
})
```


Table.FillDown

06/19/2025

Syntax

```
Table.FillDown(table as table, columns as list) as table
```

About

Returns a table from the `table` specified where the value of a previous cell is propagated to the null-valued cells below in the `columns` specified.

Example 1

Return a table with the null values in column [Place] filled with the value above them from the table.

Usage

Power Query M

```
Table.FillDown(  
    Table.FromRecords({  
        [Place = 1, Name = "Bob"],  
        [Place = null, Name = "John"],  
        [Place = 2, Name = "Brad"],  
        [Place = 3, Name = "Mark"],  
        [Place = null, Name = "Tom"],  
        [Place = null, Name = "Adam"]  
    }),  
    {"Place"}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [Place = 1, Name = "Bob"],  
    [Place = 1, Name = "John"],  
    [Place = 2, Name = "Brad"],  
    [Place = 3, Name = "Mark"],  
    [Place = 3, Name = "Tom"],
```

```
[Place = 3, Name = "Adam"]  
})
```

Table.FillUp

06/19/2025

Syntax

```
Table.FillUp(table as table, columns as list) as table
```

About

Returns a table from the `table` specified where the value of the next cell is propagated to the null-valued cells above in the `columns` specified.

Example 1

Return a table with the null values in column [Column2] filled with the value below them from the table.

Usage

Power Query M

```
Table.FillUp(  
    Table.FromRecords({  
        [Column1 = 1, Column2 = 2],  
        [Column1 = 3, Column2 = null],  
        [Column1 = 5, Column2 = 3]  
    }),  
    {"Column2"}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [Column1 = 1, Column2 = 2],  
    [Column1 = 3, Column2 = 3],  
    [Column1 = 5, Column2 = 3]  
})
```

Table.FilterWithDataTable

06/19/2025

Syntax

```
Table.FilterWithDataTable(table as table, dataTableIdentifier as text) as any
```

About

This function is intended for internal use only.

Table.FindText

06/19/2025

Syntax

```
Table.FindText(table as table, text as text) as table
```

About

Returns the rows in the table `table` that contain the text `text`. If the text is not found, an empty table is returned.

Example 1

Find the rows in the table that contain "Bob".

Usage

Power Query M

```
Table.FindText(  
    Table.FromRecords(  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    ),  
    "Bob"  
)
```

Output

```
Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]})
```

Table.First

06/19/2025

Syntax

```
Table.First(table as table, optional default as any) as any
```

About

Returns the first row of the `table` or an optional default value, `default`, if the table is empty.

Example 1

Find the first row of the table.

Usage

Power Query M

```
Table.First(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
    })  
)
```

Output

```
[CustomerID = 1, Name = "Bob", Phone = "123-4567"]
```

Example 2

Find the first row of the table `{}` or return `[a = 0, b = 0]` if empty.

Usage

Power Query M

```
Table.First(Table.FromRecords({}), [a = 0, b = 0])
```

Output

```
[a = 0, b = 0]
```

Table.FirstN

06/19/2025

Syntax

```
Table.FirstN(table as table, countOrCondition as any) as table
```

About

Returns the first row(s) of the table `table`, depending on the value of `countOrCondition`:

- If `countOrCondition` is a number, that many rows (starting at the top) will be returned.
- If `countOrCondition` is a condition, the rows that meet the condition will be returned until a row does not meet the condition.

Example 1

Find the first two rows of the table.

Usage

Power Query M

```
Table.FirstN(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
    }),  
    2  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"]  
})
```


Example 2

Find the first rows where $[a] > 0$ in the table.

Usage

Power Query M

```
Table.FirstN(  
    Table.FromRecords({  
        [a = 1, b = 2],  
        [a = 3, b = 4],  
        [a = -5, b = -6]  
    }),  
    each [a] > 0  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [a = 1, b = 2],  
    [a = 3, b = 4]  
})
```

Table.FirstValue

06/19/2025

Syntax

```
Table.FirstValue(table as table, optional default as any) as any
```

About

Returns the first column of the first row of the table `table` or a specified default value.

Table.FromColumns

06/19/2025

Syntax

```
Table.FromColumns(lists as list, optional columns as any) as table
```

About

Creates a table of type `columns` from a list `lists` containing nested lists with the column names and values. If some columns have more values than others, the missing values will be filled with the default value, 'null', if the columns are nullable.

Example 1

Return a table from a list of customer names in a list. Each value in the customer list item becomes a row value, and each list becomes a column.

Usage

Power Query M

```
Table.FromColumns({  
    {1, "Bob", "123-4567"},  
    {2, "Jim", "987-6543"},  
    {3, "Paul", "543-7890"}  
})
```

Output

Power Query M

```
Table.FromRecords({  
    [Column1 = 1, Column2 = 2, Column3 = 3],  
    [Column1 = "Bob", Column2 = "Jim", Column3 = "Paul"],  
    [Column1 = "123-4567", Column2 = "987-6543", Column3 = "543-7890"]  
})
```

Example 2

Create a table from a given list of columns and a list of column names.

Usage

Power Query M

```
Table.FromColumns(  
    {  
        {1, "Bob", "123-4567"},  
        {2, "Jim", "987-6543"},  
        {3, "Paul", "543-7890"}  
    },  
    {"CustomerID", "Name", "Phone"}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = 2, Phone = 3],  
    [CustomerID = "Bob", Name = "Jim", Phone = "Paul"],  
    [CustomerID = "123-4567", Name = "987-6543", Phone = "543-7890"]  
})
```

Example 3

Create a table with different number of columns per row. The missing row value is null.

Usage

Power Query M

```
Table.FromColumns(  
    {  
        {1, 2, 3},  
        {4, 5},  
        {6, 7, 8, 9}  
    },  
    {"column1", "column2", "column3"}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [column1 = 1, column2 = 4, column3 = 6],  
    [column1 = 2, column2 = 5, column3 = 7],  
    [column1 = 3, column2 = null, column3 = 8],  
    [column1 = null, column2 = null, column3 = 9]  
})
```

Table.FromList

06/19/2025

Syntax

```
Table.FromList(list as list, optional splitter as nullable function, optional columns as any, optional default as any, optional extraValues as nullable number) as table
```

About

Converts a list, `list` into a table by applying the optional [splitting function](#), `splitter`, to each item in the list. By default, the list is assumed to be a list of text values that is split by commas. Optional `columns` may be the number of columns, a list of columns or a `TableType`. Optional `default` and `extraValues` may also be specified.

Example 1

Create a table from a list using the default splitter.

Usage

Power Query M

```
Table.FromList(  
    {"a,apple", "b,ball", "c,cookie", "d,door"},  
    null,  
    {"Letter", "Example Word"}  
)
```

Output

Power Query M

```
Table.FromRecords(  
    [Letter = "a", #"Example Word" = "apple"],  
    [Letter = "b", #"Example Word" = "ball"],  
    [Letter = "c", #"Example Word" = "cookie"],  
    [Letter = "d", #"Example Word" = "door"]  
)
```

Example 2

Create a table from a list using a custom splitter.

Usage

Power Query M

```
Table.FromList(  
    {"a,apple", "b,ball", "c,cookie", "d,door"},  
    Splitter.SplitByNothing(),  
    {"Letter and Example Word"}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [#"Letter and Example Word" = "a,apple"],  
    [#"Letter and Example Word" = "b,ball"],  
    [#"Letter and Example Word" = "c,cookie"],  
    [#"Letter and Example Word" = "d,door"]  
})
```

Example 3

Create a table from the list using the [Record.FieldValues](#) splitter.

Usage

Power Query M

```
Table.FromList(  
    {  
        [CustomerID = 1, Name = "Bob"],  
        [CustomerID = 2, Name = "Jim"]  
    },  
    Record.FieldValues,  
    {"CustomerID", "Name"}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob"],  
    [CustomerID = 2, Name = "Jim"]  
})
```


Table.FromPartitions

06/19/2025

Syntax

```
Table.FromPartitions(partitionColumn as text, partitions as list, optional  
partitionColumnType as nullable type) as table
```

About

Returns a table that is the result of combining a set of partitioned tables, `partitions`. `partitionColumn` is the name of the column to add. The type of the column defaults to `any`, but can be specified by `partitionColumnType`.

Example 1

Find item type from the list `{number}`.

Usage

Power Query M

```
Table.FromPartitions(  
    "Year",  
    {  
        {  
            1994,  
            Table.FromPartitions(  
                "Month",  
                {  
                    {  
                        "Jan",  
                        Table.FromPartitions(  
                            "Day",  
                            {  
                                {1, #table({"Foo"}, {"Bar"})}},  
                                {2, #table({"Foo"}, {"Bar"})}}  
                            }  
                        )  
                    },  
                    {  
                        "Feb",  
                        Table.FromPartitions(  
                            "Day",
```

```

    {
        {3, #table({"Foo"}, {"Bar"})},
        {4, #table({"Foo"}, {"Bar"})}
    }
)

```

Output

Power Query M

```

Table.FromRecords({
    [
        Foo = "Bar",
        Day = 1,
        Month = "Jan",
        Year = 1994
    ],
    [
        Foo = "Bar",
        Day = 2,
        Month = "Jan",
        Year = 1994
    ],
    [
        Foo = "Bar",
        Day = 3,
        Month = "Feb",
        Year = 1994
    ],
    [
        Foo = "Bar",
        Day = 4,
        Month = "Feb",
        Year = 1994
    ]
})

```

Related content

- [Types and type conversion](#)

Table.FromRecords

06/19/2025

Syntax

```
Table.FromRecords(records as list, optional columns as any, optional missingField as nullable number) as table
```

About

Converts **records**, a list of records, into a table.

Example 1

Create a table from records, using record field names as column names.

Usage

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
})
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
})
```

Example 2

Create a table from records with typed columns and select the number columns.

Usage

Power Query M

```
Table.ColumnsOfType(  
    Table.FromRecords(  
        {[CustomerID = 1, Name = "Bob"]},  
        type table[CustomerID = Number.Type, Name = Text.Type]  
    ),  
    {type number}  
)
```

Output

```
{"CustomerID"}
```

Related content

[Missing field](#)

Table.FromRows

06/19/2025

Syntax

```
Table.FromRows(rows as list, optional columns as any) as table
```

About

Creates a table from the list `rows` where each element of the list is an inner list that contains the column values for a single row. An optional list of column names, a table type, or a number of columns could be provided for `columns`.

Example 1

Return a table with column [CustomerID] with values {1, 2}, column [Name] with values {"Bob", "Jim"}, and column [Phone] with values {"123-4567", "987-6543"}.

Usage

Power Query M

```
Table.FromRows(  
    {  
        {1, "Bob", "123-4567"},  
        {2, "Jim", "987-6543"}  
    },  
    {"CustomerID", "Name", "Phone"}
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"]  
})
```

Example 2

Return a table with column [CustomerID] with values {1, 2}, column [Name] with values {"Bob", "Jim"}, and column [Phone] with values {"123-4567", "987-6543"}, where [CustomerID] is number type, and [Name] and [Phone] are text types.

Usage

Power Query M

```
Table.FromRows(  
    {  
        {1, "Bob", "123-4567"},  
        {2, "Jim", "987-6543"}  
    },  
    type table [CustomerID = number, Name = text, Phone = text]  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"]  
})
```

Table.FromValue

06/19/2025

Syntax

```
Table.FromValue(value as any, optional options as nullable record) as table
```

About

Creates a table with a column containing the provided value or list of values, `value`. An optional record parameter, `options`, may be specified to control the following options:

- `DefaultColumnName`: The column name used when constructing a table from a list or scalar value.

Example 1

Create a table from the value 1.

Usage

```
Power Query M
```

```
Table.FromValue(1)
```

Output

```
Table.FromRecords({[Value = 1]})
```

Example 2

Create a table from the list.

Usage

```
Power Query M
```

```
Table.FromValue({1, "Bob", "123-4567"})
```

Output

Power Query M

```
Table.FromRecords({  
    [Value = 1],  
    [Value = "Bob"],  
    [Value = "123-4567"]  
})
```

Example 3

Create a table from the value 1, with a custom column name.

Usage

Power Query M

```
Table.FromValue(1, [DefaultColumnName = "MyValue"])
```

Output

```
Table.FromRecords({[MyValue = 1]})
```


Table.FuzzyGroup

06/19/2025

Syntax

```
Table.FuzzyGroup(table as table, key as any, aggregatedColumns as list, optional options as nullable record) as table
```

About

Groups the rows of `table` by fuzzily matching values in the specified column, `key`, for each row. For each group, a record is constructed containing the key columns (and their values) along with any aggregated columns specified by `aggregatedColumns`. This function cannot guarantee to return a fixed order of rows.

An optional set of `options` may be included to specify how to compare the key columns. Options include:

- `Culture`: Allows grouping records based on culture-specific rules. It can be any valid culture name. For example, a Culture option of "ja-JP" groups records based on the Japanese culture. The default value is "", which groups based on the Invariant English culture.
- `IgnoreCase`: A logical (true/false) value that allows case-insensitive key grouping. For example, when true, "Grapes" is grouped with "grapes". The default value is true.
- `IgnoreSpace`: A logical (true/false) value that allows combining of text parts in order to find groups. For example, when true, "Gra pes" is grouped with "Grapes". The default value is true.
- `SimilarityColumnName`: A name for the column that shows the similarity between an input value and the representative value for that input. The default value is null, in which case a new column for similarities will not be added.
- `Threshold`: A number between 0.00 and 1.00 that specifies the similarity score at which two values will be grouped. For example, "Grapes" and "Graes" (missing the "p") are grouped together only if this option is set to less than 0.90. A threshold of 1.00 only allows exact matches. (Note that a fuzzy "exact match" might ignore differences like casing, word order, and punctuation.) The default value is 0.80.
- `TransformationTable`: A table that allows grouping records based on custom value mappings. It should contain "From" and "To" columns. For example, "Grapes" is grouped with "Raisins" if a transformation table is provided with the "From" column containing

"Grapes" and the "To" column containing "Raisins". Note that the transformation will be applied to all occurrences of the text in the transformation table. With the above transformation table, "Grapes are sweet" will also be grouped with "Raisins are sweet".

Example 1

Group the table adding an aggregate column [Count] that contains the number of employees in each location (`each Table.RowCount(_)`).

Usage

Power Query M

```
Table.FuzzyGroup(
    Table.FromRecords(
        {
            [EmployeeID = 1, Location = "Seattle"],
            [EmployeeID = 2, Location = "seattl"],
            [EmployeeID = 3, Location = "Vancouver"],
            [EmployeeID = 4, Location = "Seatle"],
            [EmployeeID = 5, Location = "vancouver"],
            [EmployeeID = 6, Location = "Seattle"],
            [EmployeeID = 7, Location = "Vancouver"]
        },
        type table [EmployeeID = nullable number, Location = nullable text]
    ),
    "Location",
    {"Count", each Table.RowCount(_)},
    [IgnoreCase = true, IgnoreSpace = true]
)
```

Output

Power Query M

```
Table.FromRecords({
    [Location = "Seattle", Count = 4],
    [Location = "Vancouver", Count = 3]
})
```

Related content

- [How culture affects text formatting](#)

Table.FuzzyJoin

06/19/2025

Syntax

```
Table.FuzzyJoin(table1 as table, key1 as any, table2 as table, key2 as any,  
optional joinKind as nullable number, optional joinOptions as nullable record) as  
table
```

About

Joins the rows of `table1` with the rows of `table2` based on a fuzzy matching of the values of the key columns selected by `key1` (for `table1`) and `key2` (for `table2`).

Fuzzy matching is a comparison based on similarity of text rather than equality of text.

By default, an inner join is performed, however an optional `joinKind` may be included to specify the type of join. Options include:

- `JoinKind.Inner`
- `JoinKind.LeftOuter`
- `JoinKind.RightOuter`
- `JoinKind.FullOuter`
- `JoinKind.LeftAnti`
- `JoinKind.RightAnti`
- `JoinKind.LeftSemi`
- `JoinKind.RightSemi`

An optional set of `joinOptions` may be included to specify how to compare the key columns. Options include:

- `ConcurrentRequests`: A number between 1 and 8 that specifies the number of parallel threads to use for fuzzy matching. The default value is 1.
- `Culture`: Allows matching records based on culture-specific rules. It can be any valid culture name. For example, a Culture option of "ja-JP" matches records based on the Japanese culture. The default value is "", which matches based on the Invariant English culture.
- `IgnoreCase`: A logical (true/false) value that allows case-insensitive key matching. For example, when true, "Grapes" is matched with "grapes". The default value is true.

- **IgnoreSpace**: A logical (true/false) value that allows combining of text parts in order to find matches. For example, when true, "Gra pes" is matched with "Grapes". The default value is true.
- **NumberOfMatches**: A whole number that specifies the maximum number of matching rows that can be returned for every input row. For example, a value of 1 will return at most one matching row for each input row. If this option is not provided, all matching rows are returned.
- **SimilarityColumnName**: A name for the column that shows the similarity between an input value and the representative value for that input. The default value is null, in which case a new column for similarities will not be added.
- **Threshold**: A number between 0.00 and 1.00 that specifies the similarity score at which two values will be matched. For example, "Grapes" and "Graes" (missing the "p") are matched only if this option is set to less than 0.90. A threshold of 1.00 only allows exact matches. (Note that a fuzzy "exact match" might ignore differences like casing, word order, and punctuation.) The default value is 0.80.
- **TransformationTable**: A table that allows matching records based on custom value mappings. It should contain "From" and "To" columns. For example, "Grapes" is matched with "Raisins" if a transformation table is provided with the "From" column containing "Grapes" and the "To" column containing "Raisins". Note that the transformation will be applied to all occurrences of the text in the transformation table. With the above transformation table, "Grapes are sweet" will also be matched with "Raisins are sweet".

Example 1

Left inner fuzzy join of two tables based on [FirstName]

Usage

Power Query M

```
Table.FuzzyJoin(
    Table.FromRecords(
        {
            [CustomerID = 1, FirstName1 = "Bob", Phone = "555-1234"],
            [CustomerID = 2, FirstName1 = "Robert", Phone = "555-4567"]
        },
        type table [CustomerID = nullable number, FirstName1 = nullable text,
Phone = nullable text]
    ),
    {"FirstName1"},
    Table.FromRecords(
        {
            [CustomerStateID = 1, FirstName2 = "Bob", State = "TX"],
            [CustomerStateID = 2, FirstName2 = "bOB", State = "CA"]
        },
        type table [CustomerStateID = nullable number, FirstName2 = nullable text,
State = nullable text]
```

```

        type table [CustomerStateID = nullable number, FirstName2 = nullable text,
State = nullable text]
    ),
    {"FirstName2"},
    JoinKind.LeftOuter,
    [IgnoreCase = true, IgnoreSpace = false]
)

```

Output

Power Query M

```

Table.FromRecords({
    [
        CustomerID = 1,
        FirstName1 = "Bob",
        Phone = "555-1234",
        CustomerStateID = 1,
        FirstName2 = "Bob",
        State = "TX"
    ],
    [
        CustomerID = 1,
        FirstName1 = "Bob",
        Phone = "555-1234",
        CustomerStateID = 2,
        FirstName2 = "bOB",
        State = "CA"
    ],
    [
        CustomerID = 2,
        FirstName1 = "Robert",
        Phone = "555-4567",
        CustomerStateID = null,
        FirstName2 = null,
        State = null
    ]
})

```

Related content

- [How culture affects text formatting](#)

Table.FuzzyNestedJoin

06/19/2025

Syntax

```
Table.FuzzyNestedJoin(table1 as table, key1 as any, table2 as table, key2 as any,  
newColumnName as text, optional joinKind as nullable number, optional joinOptions  
as nullable record) as table
```

About

Joins the rows of `table1` with the rows of `table2` based on a fuzzy matching of the values of the key columns selected by `key1` (for `table1`) and `key2` (for `table2`). The results are returned in a new column named `newColumnName`.

Fuzzy matching is a comparison based on similarity of text rather than equality of text.

The optional `joinKind` specifies the kind of join to perform. By default, a left outer join is performed if a `joinKind` is not specified. Options include:

- `JoinKind.Inner`
- `JoinKind.LeftOuter`
- `JoinKind.RightOuter`
- `JoinKind.FullOuter`
- `JoinKind.LeftAnti`
- `JoinKind.RightAnti`
- `JoinKind.LeftSemi`
- `JoinKind.RightSemi`

An optional set of `joinOptions` may be included to specify how to compare the key columns. Options include:

- `ConcurrentRequests`: A number between 1 and 8 that specifies the number of parallel threads to use for fuzzy matching. The default value is 1.
- `Culture`: Allows matching records based on culture-specific rules. It can be any valid culture name. For example, a Culture option of "ja-JP" matches records based on the Japanese culture. The default value is "", which matches based on the Invariant English culture.

- **IgnoreCase**: A logical (true/false) value that allows case-insensitive key matching. For example, when true, "Grapes" is matched with "grapes". The default value is true.
- **IgnoreSpace**: A logical (true/false) value that allows combining of text parts in order to find matches. For example, when true, "Gra pes" is matched with "Grapes". The default value is true.
- **NumberOfMatches**: A whole number that specifies the maximum number of matching rows that can be returned for every input row. For example, a value of 1 will return at most one matching row for each input row. If this option is not provided, all matching rows are returned.
- **SimilarityColumnName**: A name for the column that shows the similarity between an input value and the representative value for that input. The default value is null, in which case a new column for similarities will not be added.
- **Threshold**: A number between 0.00 and 1.00 that specifies the similarity score at which two values will be matched. For example, "Grapes" and "Graes" (missing the "p") are matched only if this option is set to less than 0.90. A threshold of 1.00 only allows exact matches. (Note that a fuzzy "exact match" might ignore differences like casing, word order, and punctuation.) The default value is 0.80.
- **TransformationTable**: A table that allows matching records based on custom value mappings. It should contain "From" and "To" columns. For example, "Grapes" is matched with "Raisins" if a transformation table is provided with the "From" column containing "Grapes" and the "To" column containing "Raisins". Note that the transformation will be applied to all occurrences of the text in the transformation table. With the above transformation table, "Grapes are sweet" will also be matched with "Raisins are sweet".

Example 1

Left inner fuzzy join of two tables based on [FirstName]

Usage

Power Query M

```
Table.FuzzyNestedJoin(
    Table.FromRecords(
        {
            [CustomerID = 1, FirstName1 = "Bob", Phone = "555-1234"],
            [CustomerID = 2, FirstName1 = "Robert", Phone = "555-4567"]
        },
        type table [CustomerID = nullable number, FirstName1 = nullable text,
Phone = nullable text]
    ),
    {"FirstName1"},
    Table.FromRecords(
        {
```

```

        [CustomerStateID = 1, FirstName2 = "Bob", State = "TX"],
        [CustomerStateID = 2, FirstName2 = "bOB", State = "CA"]
    },
    type table [CustomerStateID = nullable number, FirstName2 = nullable text,
State = nullable text]
),
{"FirstName2"},
"NestedTable",
JoinKind.LeftOuter,
[IgnoreCase = true, IgnoreSpace = false]
)

```

Output

Power Query M

```

Table.FromRecords({
    [
        CustomerID = 1,
        FirstName1 = "Bob",
        Phone = "555-1234",
        NestedTable = Table.FromRecords({
            [
                CustomerStateID = 1,
                FirstName2 = "Bob",
                State = "TX"
            ],
            [
                CustomerStateID = 2,
                FirstName2 = "bOB",
                State = "CA"
            ]
        })
    ],
    [
        CustomerID = 2,
        FirstName1 = "Robert",
        Phone = "555-4567",
        NestedTable = Table.FromRecords({})
    ]
})

```

Related content

- [How culture affects text formatting](#)

Table.Group

06/19/2025

Syntax

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

About

Groups the rows of `table` by the key columns defined by `key`. The `key` can either be a single column name, or a list of column names. For each group, a record is constructed containing the key columns (and their values), along with any aggregated columns specified by `aggregatedColumns`. Optionally, `groupKind` and `comparer` may also be specified.

If the data is already sorted by the key columns, then a `groupKind` of `GroupKind.Local` can be provided. This may improve the performance of grouping in certain cases, since all the rows with a given set of key values are assumed to be contiguous.

When passing a `comparer`, note that if it treats differing keys as equal, a row may be placed in a group whose keys differ from its own.

This function does not guarantee the ordering of the rows it returns.

Example 1

Group the table adding an aggregate column [total] which contains the sum of prices ("each List.Sum([price])").

Usage

Power Query M

```
Table.Group(  
    Table.FromRecords({  
        [CustomerID = 1, price = 20],  
        [CustomerID = 2, price = 10],  
        [CustomerID = 2, price = 20],  
        [CustomerID = 1, price = 10],  
        [CustomerID = 3, price = 20],  
        [CustomerID = 3, price = 5]  
    })  
)
```

```
"CustomerID",  
{"total", each List.Sum([price])}  
)
```

Output

Power Query M

```
Table.FromRecords(  
    {  
        [CustomerID = 1, total = 30],  
        [CustomerID = 2, total = 30],  
        [CustomerID = 3, total = 25]  
    },  
    {"CustomerID", "total"}  
)
```

Related content

[Comparer functions](#)

Table.HasColumns

06/19/2025

Syntax

```
Table.HasColumns(table as table, columns as any) as logical
```

About

Indicates whether the `table` contains the specified column(s), `columns`. Returns `true` if the table contains the column(s), `false` otherwise.

Example 1

Determine if the table has the column [Name].

Usage

Power Query M

```
Table.HasColumns(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    "Name"  
)
```

Output

```
true
```

Example 2

Find if the table has the column [Name] and [PhoneNumber].

Usage

Power Query M

```
Table.HasColumns(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    {"Name", "PhoneNumber"}  
)
```

Output

false

Table.InsertRows

06/19/2025

Syntax

```
Table.InsertRows(table as table, offset as number, rows as list) as table
```

About

Returns a table with the list of rows, `rows`, inserted into the `table` at the given position, `offset`. Each column in the row to insert must match the column types of the table.

Example 1

Insert the row into the table at position 1.

Usage

Power Query M

```
Table.InsertRows(  
    Table.FromRecords(  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"]  
    ),  
    1,  
    {[CustomerID = 3, Name = "Paul", Phone = "543-7890"]}  
)
```

Output

Power Query M

```
Table.FromRecords(  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"]  
)
```

Example 2

Insert two rows into the table at position 1.

Usage

Power Query M

```
Table.InsertRows(  
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),  
    1,  
    {  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
    }  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
})
```

Table.IsDistinct

06/19/2025

Syntax

```
Table.IsDistinct(table as table, optional comparisonCriteria as any) as logical
```

About

Indicates whether the `table` contains only distinct rows (no duplicates). Returns `true` if the rows are distinct, `false` otherwise. An optional parameter, `comparisonCriteria`, specifies which columns of the table are tested for duplication. If `comparisonCriteria` is not specified, all columns are tested.

Example 1

Determine if the table is distinct.

Usage

Power Query M

```
Table.IsDistinct(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    })  
)
```

Output

```
true
```

Example 2

Determine if the table is distinct in column.

Usage

```
Table.IsDistinct(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 5, Name = "Bob", Phone = "232-1550"]  
    }),  
    "Name"  
)
```

Output

false

Related content

[Comparison criteria](#)

Table.IsEmpty

06/19/2025

Syntax

```
Table.IsEmpty(table as table) as logical
```

About

Indicates whether the `table` contains any rows. Returns `true` if there are no rows (i.e. the table is empty), `false` otherwise.

Example 1

Determine if the table is empty.

Usage

Power Query M

```
Table.IsEmpty(  
    Table.FromRecords(  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
    )  
)
```

Output

```
false
```

Example 2

Determine if the table `({})` is empty.

Usage

Power Query M

```
Table.IsEmpty(Table.FromRecords({}))
```

Output

```
true
```

Table.Join

06/19/2025

Syntax

```
Table.Join(table1 as table, key1 as any, table2 as table, key2 as any, optional  
joinKind as nullable number, optional joinAlgorithm as nullable number, optional  
keyEqualityComparers as nullable list) as table
```

About

Joins the rows of `table1` with the rows of `table2` based on the equality of the values of the key columns selected by `key1` (for `table1`) and `key2` (for `table2`).

By default, an inner join is performed, however an optional `joinKind` may be included to specify the type of join. Options include:

- [JoinKind.Inner](#)
- [JoinKind.LeftOuter](#)
- [JoinKind.RightOuter](#)
- [JoinKind.FullOuter](#)
- [JoinKind.LeftAnti](#)
- [JoinKind.RightAnti](#)
- [JoinKind.LeftSemi](#)
- [JoinKind.RightSemi](#)

An optional set of `keyEqualityComparers` may be included to specify how to compare the key columns. This parameter is currently intended for internal use only.

Example 1

Join two tables using a single key column.

Usage

Power Query M

```
Table.Join(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
    })  
)
```

```

[CustomerID = 3, Name = "Paul", Phone = "543-7890"],
[CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
}),
"CustomerID",
Table.FromRecords({
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
    [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
    [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
    [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
    [OrderID = 5, CustomerID = 3, Item = "Band-aids", Price = 2.0],
    [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
    [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25]
}),
"CustomerID"
)

```

Output

Power Query M

```

Table.FromRecords({
    [CustomerID = 1, Name = "Bob", Phone = "123-4567", OrderID = 1, Item =
"Fishing rod", Price = 100],
    [CustomerID = 1, Name = "Bob", Phone = "123-4567", OrderID = 2, Item = "1 lb.
worms", Price = 5],
    [CustomerID = 2, Name = "Jim", Phone = "987-6543", OrderID = 3, Item =
"Fishing net", Price = 25],
    [CustomerID = 3, Name = "Paul", Phone = "543-7890", OrderID = 4, Item = "Fish
tazer", Price = 200],
    [CustomerID = 3, Name = "Paul", Phone = "543-7890", OrderID = 5, Item =
"Band-aids", Price = 2],
    [CustomerID = 1, Name = "Bob", Phone = "123-4567", OrderID = 6, Item = "Tackle
box", Price = 20]
})

```

Example 2

Join two tables that have conflicting column names, using multiple key columns.

Usage

Power Query M

```

let
    customers = Table.FromRecords({
        [TenantID = 1, CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [TenantID = 1, CustomerID = 2, Name = "Jim", Phone = "987-6543"]
    }),
    orders = Table.FromRecords({
        [TenantID = 1, OrderID = 1, CustomerID = 1, Name = "Fishing rod", Price =

```

```

100.0],
    [TenantID = 1, OrderID = 2, CustomerID = 1, Name = "1 lb. worms", Price =
5.0],
    [TenantID = 1, OrderID = 3, CustomerID = 2, Name = "Fishing net", Price =
25.0]
    })
in
    Table.Join(
        customers,
        {"TenantID", "CustomerID"},
        Table.PrefixColumns(orders, "Order"),
        {"Order.TenantID", "Order.CustomerID"}
    )

```

Power Query M

```

Table.FromRecords({
    [TenantID = 1, CustomerID = 1, Name = "Bob", Phone = "123-4567",
Order.TenantID = 1, Order.OrderID = 1, Order.CustomerID = 1, Order.Name = "Fishing
rod", Order.Price = 100],
    [TenantID = 1, CustomerID = 1, Name = "Bob", Phone = "123-4567",
Order.TenantID = 1, Order.OrderID = 2, Order.CustomerID = 1, Order.Name = "1 lb.
worms", Order.Price = 5],
    [TenantID = 1, CustomerID = 2, Name = "Jim", Phone = "987-6543",
Order.TenantID = 1, Order.OrderID = 3, Order.CustomerID = 2, Order.Name = "Fishing
net", Order.Price = 25]
})

```

Table.Keys

06/19/2025

Syntax

```
Table.Keys(table as table) as list
```

About

Returns the keys of the specified table.

Example 1

Get the list of keys for a table.

Usage

Power Query M

```
let
    table = Table.FromRecords({
        [Id = 1, Name = "Hello There"],
        [Id = 2, Name = "Good Bye"]
    }),
    tableWithKeys = Table.AddKey(table, {"Id"}, true),
    keys = Table.Keys(tableWithKeys)
in
    keys
```

Output

```
{[Columns = {"Id"}, Primary = true]}
```

Table.Last

06/19/2025

Syntax

```
Table.Last(table as table, optional default as any) as any
```

About

Returns the last row of the `table` or an optional default value, `default`, if the table is empty.

Example 1

Find the last row of the table.

Usage

Power Query M

```
Table.Last(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
    })  
)
```

Output

```
[CustomerID = 3, Name = "Paul", Phone = "543-7890"]
```

Example 2

Find the last row of the table `{{}}` or return `[a = 0, b = 0]` if empty.

Usage

Power Query M

```
Table.Last(Table.FromRecords({}), [a = 0, b = 0])
```

Output

```
[a = 0, b = 0]
```


Table.LastN

06/19/2025

Syntax

```
Table.LastN(table as table, countOrCondition as any) as table
```

About

Returns the last row(s) from the table, **table**, depending on the value of **countOrCondition**:

- If **countOrCondition** is a number, that many rows will be returned starting from position (end - **countOrCondition**).
- If **countOrCondition** is a condition, the rows that meet the condition will be returned in ascending position until a row does not meet the condition.

Example 1

Find the last two rows of the table.

Usage

Power Query M

```
Table.LastN(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
    }),  
    2  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
})
```

Example 2

Find the last rows where $[a] > 0$ in the table.

Usage

Power Query M

```
Table.LastN(  
    Table.FromRecords({  
        [a = -1, b = -2],  
        [a = 3, b = 4],  
        [a = 5, b = 6]  
    }),  
    each _ [a] > 0
```

Output

Power Query M

```
Table.FromRecords({  
    [a = 3, b = 4],  
    [a = 5, b = 6]  
})
```

Table.MatchesAllRows

06/19/2025

Syntax

```
Table.MatchesAllRows(table as table, condition as function) as logical
```

About

Indicates whether all the rows in the `table` match the given `condition`. Returns `true` if all of the rows match, `false` otherwise.

Example 1

Determine whether all of the row values in column [a] are even in the table.

Usage

Power Query M

```
Table.MatchesAllRows(  
    Table.FromRecords({  
        [a = 2, b = 4],  
        [a = 6, b = 8]  
    }),  
    each Number.Mod([a], 2) = 0  
)
```

Output

```
true
```

Example 2

Find if all of the row values are [a = 1, b = 2], in the table `({[a = 1, b = 2], [a = 3, b = 4]})`.

Usage

Power Query M

```
Table.MatchesAllRows(  
  Table.FromRecords({  
    [a = 1, b = 2],  
    [a = -3, b = 4]  
  }),  
  each _ = [a = 1, b = 2]  
)
```

Output

false

Table.MatchesAnyRows

06/19/2025

Syntax

```
Table.MatchesAnyRows(table as table, condition as function) as logical
```

About

Indicates whether any the rows in the `table` match the given `condition`. Returns `true` if any of the rows match, `false` otherwise.

Example 1

Determine whether any of the row values in column [a] are even in the table `{{[a = 2, b = 4], [a = 6, b = 8]}}`.

Usage

Power Query M

```
Table.MatchesAnyRows(  
    Table.FromRecords(  
        [a = 1, b = 4],  
        [a = 3, b = 8]  
    ),  
    each Number.Mod([a], 2) = 0  
)
```

Output

```
false
```

Example 2

Determine whether any of the row values are [a = 1, b = 2], in the table `{{[a = 1, b = 2], [a = 3, b = 4]}}`.

Usage

Power Query M

```
Table.MatchesAnyRows(  
    Table.FromRecords({  
        [a = 1, b = 2],  
        [a = -3, b = 4]  
    }),  
    each _ = [a = 1, b = 2]  
)
```

Output

true

Table.Max

06/19/2025

Syntax

```
Table.Max(table as table, comparisonCriteria as any, optional default as any) as any
```

About

Returns the largest row in the `table`, given the `comparisonCriteria`. If the table is empty, the optional `default` value is returned.

Example 1

Find the row with the largest value in column [a] in the table `([a = 2, b = 4], [a = 6, b = 8])`.

Usage

Power Query M

```
Table.Max(  
    Table.FromRecords({  
        [a = 2, b = 4],  
        [a = 6, b = 8]  
    }),  
    "a"  
)
```

Output

```
[a = 6, b = 8]
```

Example 2

Find the row with the largest value in column [a] in the table `({})`. Return -1 if empty.

Usage

Power Query M

```
Table.Max(#table({"a"}, {}), "a", -1)
```

Output

-1

Related content

[Comparison criteria](#)

Table.MaxN

06/19/2025

Syntax

```
Table.MaxN(table as table, comparisonCriteria as any, countOrCondition as any) as table
```

About

Returns the largest row(s) in the `table`, given the `comparisonCriteria`. After the rows are sorted, the `countOrCondition` parameter must be specified to further filter the result. Note the sorting algorithm cannot guarantee a fixed sorted result. The `countOrCondition` parameter can take multiple forms:

- If a number is specified, a list of up to `countOrCondition` items in ascending order is returned.
- If a condition is specified, a list of items that initially meet the condition is returned. Once an item fails the condition, no further items are considered.

Example 1

Find the row with the largest value in column [a] with the condition [a] > 0, in the table. The rows are sorted before the filter is applied.

Usage

Power Query M

```
Table.MaxN(  
    Table.FromRecords({  
        [a = 2, b = 4],  
        [a = 0, b = 0],  
        [a = 6, b = 2]  
    }),  
    "a",  
    each [a] > 0  
)
```

Output

```
Table.FromRecords({  
    [a = 6, b = 2],  
    [a = 2, b = 4]  
})
```

Example 2

Find the row with the largest value in column [a] with the condition [b] > 0, in the table. The rows are sorted before the filter is applied.

Usage

```
Table.MaxN(  
    Table.FromRecords({  
        [a = 2, b = 4],  
        [a = 8, b = 0],  
        [a = 6, b = 2]  
    }),  
    "a",  
    each [b] > 0  
)
```

Output

```
Table.FromRecords({})
```

Related content

[Comparison criteria](#)

Table.Min

06/19/2025

Syntax

```
Table.Min(table as table, comparisonCriteria as any, optional default as any) as any
```

About

Returns the smallest row in the `table`, given the `comparisonCriteria`. If the table is empty, the optional `default` value is returned.

Example 1

Find the row with the smallest value in column [a] in the table.

Usage

Power Query M

```
Table.Min(  
    Table.FromRecords({  
        [a = 2, b = 4],  
        [a = 6, b = 8]  
    }),  
    "a"  
)
```

Output

```
[a = 2, b = 4]
```

Example 2

Find the row with the smallest value in column [a] in the table. Return -1 if empty.

Usage

Power Query M

```
Table.Min(#table({"a"}, {}), "a", -1)
```

Output

```
-1
```

Related content

[Comparison criteria](#)

Table.MinN

06/19/2025

Syntax

```
Table.MinN(table as table, comparisonCriteria as any, countOrCondition as any) as table
```

About

Returns the smallest row(s) in the `table`, given the `comparisonCriteria`. After the rows are sorted, the `countOrCondition` parameter must be specified to further filter the result. Note the sorting algorithm cannot guarantee a fixed sorted result. The `countOrCondition` parameter can take multiple forms:

- If a number is specified, a list of up to `countOrCondition` items in ascending order is returned.
- If a condition is specified, a list of items that initially meet the condition is returned. Once an item fails the condition, no further items are considered.

Example 1

Find the row with the smallest value in column [a] with the condition [a] < 3, in the table. The rows are sorted before the filter is applied.

Usage

Power Query M

```
Table.MinN(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 0, b = 0],
        [a = 6, b = 4]
    }),
    "a",
    each [a] < 3
)
```

Output

```
Table.FromRecords({  
    [a = 0, b = 0],  
    [a = 2, b = 4]  
})
```

Example 2

Find the row with the smallest value in column [a] with the condition [b] < 0, in the table. The rows are sorted before the filter is applied.

Usage

```
Table.MinN(  
    Table.FromRecords({  
        [a = 2, b = 4],  
        [a = 8, b = 0],  
        [a = 6, b = 2]  
    }),  
    "a",  
    each [b] < 0  
)
```

Output

```
Table.FromRecords({})
```

Related content

[Comparison criteria](#)

Table.NestedJoin

06/19/2025

Syntax

```
Table.NestedJoin(table1 as table, key1 as any, table2 as any, key2 as any,  
newColumnName as text, optional joinKind as nullable number, optional  
keyEqualityComparers as nullable list) as table
```

About

Joins the rows of `table1` with the rows of `table2` based on the equality of the values of the key columns selected by `key1` (for `table1`) and `key2` (for `table2`). The results are entered into the column named `newColumnName`.

The optional `joinKind` specifies the kind of join to perform. By default, a left outer join is performed if a `joinKind` is not specified.

An optional set of `keyEqualityComparers` may be included to specify how to compare the key columns. This `keyEqualityComparers` feature is currently intended for internal use only.

Example 1

Join two tables using a single key column.

Usage

Power Query M

```
Table.NestedJoin(  
    Table.FromRecords({  
        [CustomerToCall = 1],  
        [CustomerToCall = 3]  
    }),  
    {"CustomerToCall"},  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    {"CustomerID"},
```

```
"CustomerDetails"  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerToCall = 1, CustomerDetails = Table.FromRecords({[CustomerID = 1,  
Name = "Bob", Phone = "123-4567"]})],  
    [CustomerToCall = 3, CustomerDetails = Table.FromRecords({[CustomerID = 3,  
Name = "Paul", Phone = "543-7890"]})]  
})
```

Related content

[Join kind](#)

Table.Partition

06/19/2025

Syntax

```
Table.Partition(table as table, column as text, groups as number, hash as function) as list
```

About

Partitions the `table` into a list of `groups` number of tables, based on the value of the `column` and a `hash` function. The `hash` function is applied to the value of the `column` row to obtain a hash value for the row. The hash value modulo `groups` determines in which of the returned tables the row will be placed.

- `table`: The table to partition.
- `column`: The column to hash to determine which returned table the row is in.
- `groups`: The number of tables the input table will be partitioned into.
- `hash`: The function applied to obtain a hash value.

Example 1

Partition the table `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})` into 2 tables on column `[a]`, using the value of the columns as the hash function.

Usage

Power Query M

```
Table.Partition(  
    Table.FromRecords({  
        [a = 2, b = 4],  
        [a = 1, b = 4],  
        [a = 2, b = 4],  
        [a = 1, b = 4]  
    }),  
    "a",  
    2,  
    each _  
)
```

Output

Power Query M

```
{
  Table.FromRecords({
    [a = 2, b = 4],
    [a = 2, b = 4]
  }),
  Table.FromRecords({
    [a = 1, b = 4],
    [a = 1, b = 4]
  })
}
```

Table.PartitionValues

06/19/2025

Syntax

```
Table.PartitionValues(table as table) as table
```

About

Returns information about how a table is partitioned. A table is returned where each column is a partition column in the original table, and each row corresponds to a partition in the original table.

Table.Pivot

06/19/2025

Syntax

```
Table.Pivot(table as table, pivotValues as list, attributeColumn as text,  
valueColumn as text, optional aggregationFunction as nullable function) as table
```

About

Given a pair of columns representing attribute-value pairs, rotates the data in the attribute column into a column headings.

Example 1

Take the values "a", "b", and "c" in the attribute column of table `({ [ key = "x", attribute = "a", value = 1 ], [ key = "x", attribute = "c", value = 3 ], [ key = "y", attribute = "a", value = 2 ], [ key = "y", attribute = "b", value = 4 ] })` and pivot them into their own column.

Usage

Power Query M

```
Table.Pivot(  
    Table.FromRecords({  
        [key = "x", attribute = "a", value = 1],  
        [key = "x", attribute = "c", value = 3],  
        [key = "y", attribute = "a", value = 2],  
        [key = "y", attribute = "b", value = 4]  
    }),  
    {"a", "b", "c"},  
    "attribute",  
    "value"  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [key = "x", a = 1, b = null, c = 3],
```

```
[key = "y", a = 2, b = 4, c = null]
})
```

Example 2

Take the values "a", "b", and "c" in the attribute column of table `({ [ key = "x", attribute = "a", value = 1 ], [ key = "x", attribute = "c", value = 3 ], [ key = "x", attribute = "c", value = 5 ], [ key = "y", attribute = "a", value = 2 ], [ key = "y", attribute = "b", value = 4 ] })` and pivot them into their own column. The attribute "c" for key "x" has multiple values associated with it, so use the function `List.Max` to resolve the conflict.

Usage

Power Query M

```
Table.Pivot(
    Table.FromRecords({
        [key = "x", attribute = "a", value = 1],
        [key = "x", attribute = "c", value = 3],
        [key = "x", attribute = "c", value = 5],
        [key = "y", attribute = "a", value = 2],
        [key = "y", attribute = "b", value = 4]
    }),
    {"a", "b", "c"},
    "attribute",
    "value",
    List.Max
)
```

Output

Power Query M

```
Table.FromRecords({
    [key = "x", a = 1, b = null, c = 5],
    [key = "y", a = 2, b = 4, c = null]
})
```

Table.PositionOf

06/19/2025

Syntax

```
Table.PositionOf(table as table, row as record, optional occurrence as any,  
optional equationCriteria as any) as any
```

About

Returns the row position of the first occurrence of the **row** in the **table** specified. Returns -1 if no occurrence is found.

- **table**: The input table.
- **row**: The row in the table to find the position of.
- **occurrence**: *[Optional]* Specifies which occurrences of the row to return.
- **equationCriteria**: *[Optional]* Controls the comparison between the table rows.

Example 1

Find the position of the first occurrence of [a = 2, b = 4] in the table ({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]}).

Usage

Power Query M

```
Table.PositionOf(  
    Table.FromRecords({  
        [a = 2, b = 4],  
        [a = 1, b = 4],  
        [a = 2, b = 4],  
        [a = 1, b = 4]  
    }),  
    [a = 2, b = 4]  
)
```

Output

0

Example 2

Find the position of the second occurrence of [a = 2, b = 4] in the table ({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]}).

Usage

Power Query M

```
Table.PositionOf(  
    Table.FromRecords({  
        [a = 2, b = 4],  
        [a = 1, b = 4],  
        [a = 2, b = 4],  
        [a = 1, b = 4]  
    }),  
    [a = 2, b = 4],  
    1  
)
```

Output

2

Example 3

Find the position of all the occurrences of [a = 2, b = 4] in the table ({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]}).

Usage

Power Query M

```
Table.PositionOf(  
    Table.FromRecords({  
        [a = 2, b = 4],  
        [a = 1, b = 4],  
        [a = 2, b = 4],  
        [a = 1, b = 4]  
    }),  
    [a = 2, b = 4],  
    Occurrence.All  
)
```

Output

{0, 2}

Related content

[Equation criteria](#)

Table.PositionOfAny

06/19/2025

Syntax

```
Table.PositionOfAny(table as table, rows as list, optional occurrence as nullable number, optional equationCriteria as any) as any
```

About

Returns the row(s) position(s) from the `table` of the first occurrence of the list of `rows`. Returns -1 if no occurrence is found.

- `table`: The input table.
- `rows`: The list of rows in the table to find the positions of.
- `occurrence`: *[Optional]* Specifies which occurrences of the row to return.
- `equationCriteria`: *[Optional]* Controls the comparison between the table rows.

Example 1

Find the position of the first occurrence of [a = 2, b = 4] or [a = 6, b = 8] in the table `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})`.

Usage

Power Query M

```
Table.PositionOfAny(  
    Table.FromRecords(  
        [a = 2, b = 4],  
        [a = 1, b = 4],  
        [a = 2, b = 4],  
        [a = 1, b = 4]  
    )),  
    {  
        [a = 2, b = 4],  
        [a = 6, b = 8]  
    }  
)
```

Output

0

Example 2

Find the position of all the occurrences of [a = 2, b = 4] or [a = 6, b = 8] in the table ({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]}).

Usage

Power Query M

```
Table.PositionOfAny(  
    Table.FromRecords({  
        [a = 2, b = 4],  
        [a = 6, b = 8],  
        [a = 2, b = 4],  
        [a = 1, b = 4]  
    }),  
    {  
        [a = 2, b = 4],  
        [a = 6, b = 8]  
    },  
    Occurrence.All  
)
```

Output

{0, 1, 2}

Related content

[Equation criteria](#)

Table.PrefixColumns

06/19/2025

Syntax

```
Table.PrefixColumns(table as table, prefix as text) as table
```

About

Returns a table where all the column names from the `table` provided are prefixed with the given text, `prefix`, plus a period in the form `prefix.ColumnName`.

Example 1

Prefix the columns with "MyTable" in the table.

Usage

Power Query M

```
Table.PrefixColumns(  
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),  
    "MyTable"  
)
```

Output

```
Table.FromRecords({[MyTable.CustomerID = 1, MyTable.Name = "Bob", MyTable.Phone = "123-4567"]})
```

Table.Profile

06/19/2025

Syntax

```
Table.Profile(table as table, optional additionalAggregates as nullable list) as  
table
```

About

Returns a profile for the columns in `table`.

The following information is returned for each column (when applicable):

- minimum
- maximum
- average
- standard deviation
- count
- null count
- distinct count

Table.PromoteHeaders

06/19/2025

Syntax

```
Table.PromoteHeaders(table as table, optional options as nullable record) as table
```

About

Promotes the first row of values as the new column headers (i.e. column names). By default, only text or number values are promoted to headers. Valid options:

- **PromoteAllScalars**: If set to **true**, all the scalar values in the first row are promoted to headers using the **Culture**, if specified (or current document locale). For values that cannot be converted to text, a default column name will be used.
- **Culture**: A culture name specifying the culture for the data.

Example 1

Promote the first row of values in the table.

Usage

Power Query M

```
Table.PromoteHeaders(  
    Table.FromRecords({  
        [Column1 = "CustomerID", Column2 = "Name", Column3 = #date(1980, 1, 1)],  
        [Column1 = 1, Column2 = "Bob", Column3 = #date(1980, 1, 1)]  
    })  
)
```

Output

```
Table.FromRecords({[CustomerID = 1, Name = "Bob", Column3 = #date(1980, 1, 1)]})
```

Example 2

Promote all the scalars in the first row of the table to headers.

Usage

Power Query M

```
Table.PromoteHeaders(  
    Table.FromRecords(  
        [Rank = 1, Name = "Name", Date = #date(1980, 1, 1)],  
        [Rank = 1, Name = "Bob", Date = #date(1980, 1, 1)]  
    ),  
    [PromoteAllScalars = true, Culture = "en-US"]  
)
```

Output

```
Table.FromRecords({[1 = 1, Name = "Bob", #"1/1/1980" = #date(1980, 1, 1)]})
```

Related content

- [How culture affects text formatting](#)

Table.Range

06/19/2025

Syntax

```
Table.Range(table as table, offset as number, optional count as nullable number)  
as table
```

About

Returns the rows from the `table` starting at the specified `offset`. An optional parameter, `count`, specifies how many rows to return. By default, all the rows after the offset are returned.

Example 1

Return all the rows starting at offset 1 in the table.

Usage

Power Query M

```
Table.Range(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    1  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
})
```

Example 2

Return one row starting at offset 1 in the table.

Usage

Power Query M

```
Table.Range(  
    Table.FromRecords(  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    ),  
    1,  
    1  
)
```

Output

```
Table.FromRecords({[CustomerID = 2, Name = "Jim", Phone = "987-6543"]})
```


Table.RemoveColumns

06/19/2025

Syntax

```
Table.RemoveColumns(table as table, columns as any, optional missingField as nullable number) as table
```

About

Removes the specified `columns` from the `table` provided. If the specified column doesn't exist, an error is raised unless the optional parameter `missingField` specifies an alternative behavior (for example, [MissingField.UseNull](#) or [MissingField.Ignore](#)).

Example 1

Remove column [Phone] from the table.

Usage

Power Query M

```
Table.RemoveColumns(  
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),  
    "Phone"  
)
```

Output

```
Table.FromRecords({[CustomerID = 1, Name = "Bob"]})
```

Example 2

Try to remove a non-existent column from the table.

Usage

Power Query M

```
Table.RemoveColumns(  
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
```

```
"Address"  
)
```

Output

```
[Expression.Error] The column 'Address' of the table wasn't found.
```

Table.RemoveFirstN

06/19/2025

Syntax

```
Table.RemoveFirstN(table as table, optional countOrCondition as any) as table
```

About

Returns a table that does not contain the first specified number of rows, **countOrCondition**, of the table **table**. The number of rows removed depends on the optional parameter **countOrCondition**.

- If **countOrCondition** is omitted only the first row is removed.
- If **countOrCondition** is a number, that many rows (starting at the top) will be removed.
- If **countOrCondition** is a condition, the rows that meet the condition will be removed until a row does not meet the condition.

Example 1

Remove the first row of the table.

Usage

Power Query M

```
Table.RemoveFirstN(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    1  
)
```

Output

Power Query M

```
Table.FromRecords({
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
})
```

Example 2

Remove the first two rows of the table.

Usage

Power Query M

```
Table.RemoveFirstN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    2
)
```

Output

Power Query M

```
Table.FromRecords({
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
})
```

Example 3

Remove the first rows where [CustomerID] <=2 of the table.

Usage

Power Query M

```
Table.RemoveFirstN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"] ,
    })
)
```

```
[CustomerID = 3, Name = "Paul", Phone = "543-7890"] ,  
[CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
}),  
each [CustomerID] <= 2  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
})
```

Table.RemoveLastN

06/19/2025

Syntax

```
Table.RemoveLastN(table as table, optional countOrCondition as any) as table
```

About

Returns a table that does not contain the last **countOrCondition** rows of the table **table**. The number of rows removed depends on the optional parameter **countOrCondition**.

- If **countOrCondition** is omitted only the last row is removed.
- If **countOrCondition** is a number, that many rows (starting at the bottom) will be removed.
- If **countOrCondition** is a condition, the rows that meet the condition will be removed until a row does not meet the condition.

Example 1

Remove the last row of the table.

Usage

Power Query M

```
Table.RemoveLastN(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    1  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
```

```
[CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
[CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
})
```

Example 2

Remove the last rows where [CustomerID] > 2 of the table.

Usage

Power Query M

```
Table.RemoveLastN(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    each [CustomerID] >= 2  
)
```

Output

```
Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]})
```

Table.RemoveMatchingRows

06/19/2025

Syntax

```
Table.RemoveMatchingRows(table as table, rows as list, optional equationCriteria as any) as table
```

About

Removes all occurrences of the specified **rows** from the **table**. An optional parameter **equationCriteria** may be specified to control the comparison between the rows of the table.

Example 1

Remove any rows where [a = 1] from the table `{{[a = 1, b = 2], [a = 3, b = 4], [a = 1, b = 6]}}`.

Usage

Power Query M

```
Table.RemoveMatchingRows(  
    Table.FromRecords(  
        [a = 1, b = 2],  
        [a = 3, b = 4],  
        [a = 1, b = 6]  
    ),  
    {[a = 1]},  
    "a"  
)
```

Output

```
Table.FromRecords({[a = 3, b = 4]})
```

Related content

[Equation criteria](#)

Table.RemoveRows

06/19/2025

Syntax

```
Table.RemoveRows(table as table, offset as number, optional count as nullable number) as table
```

About

Removes **count** of rows from the beginning of the **table**, starting at the **offset** specified. A default count of 1 is used if the **count** parameter isn't provided.

Example 1

Remove the first row from the table.

Usage

Power Query M

```
Table.RemoveRows(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    0  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
})
```

Example 2

Remove the row at position 1 from the table.

Usage

Power Query M

```
Table.RemoveRows(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    1  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
})
```

Example 3

Remove two rows starting at position 1 from the table.

Usage

Power Query M

```
Table.RemoveRows(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    1,  
    2  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
})
```

Table.RemoveRowsWithErrors

06/19/2025

Syntax

```
Table.RemoveRowsWithErrors(table as table, optional columns as nullable list) as table
```

About

Returns a table with the rows removed from the input table that contain an error in at least one of the cells. If a columns list is specified, then only the cells in the specified columns are inspected for errors.

Example 1

Remove error value from first row.

Usage

Power Query M

```
Table.RemoveRowsWithErrors(  
    Table.FromRecords({  
        [Column1 = ...],  
        [Column1 = 2],  
        [Column1 = 3]  
    })  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [Column1 = 2],  
    [Column1 = 3]  
})
```

Table.RenameColumns

06/19/2025

Syntax

```
Table.RenameColumns(table as table, renames as list, optional missingField as nullable number) as table
```

About

Performs the given renames to the columns in table `table`. A replacement operation `renames` consists of a list of two values, the old column name and new column name, provided in a list. If the column doesn't exist, an exception is thrown unless the optional parameter `missingField` specifies an alternative (eg. [MissingField.UseNull](#) or [MissingField.Ignore](#)).

Example 1

Replace the column name "CustomerNum" with "CustomerID" in the table.

Usage

Power Query M

```
Table.RenameColumns(  
    Table.FromRecords({[CustomerNum = 1, Name = "Bob", Phone = "123-4567"]}),  
    {"CustomerNum", "CustomerID"}  
)
```

Output

```
Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]})
```

Example 2

Replace the column name "CustomerNum" with "CustomerID" and "PhoneNum" with "Phone" in the table.

Usage

Power Query M

```
Table.RenameColumns(  
    Table.FromRecords({[CustomerNum = 1, Name = "Bob", PhoneNum = "123-4567"]}),  
    {  
        {"CustomerNum", "CustomerID"},  
        {"PhoneNum", "Phone"}  
    }  
)
```

Output

```
Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]})
```

Example 3

Replace the column name "NewCol" with "NewColumn" in the table, and ignore if the column doesn't exist.

Usage

Power Query M

```
Table.RenameColumns(  
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),  
    {"NewCol", "NewColumn"},  
    MissingField.Ignore  
)
```

Output

```
Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]})
```

Table.ReorderColumns

06/19/2025

Syntax

```
Table.ReorderColumns(table as table, columnOrder as list, optional missingField as nullable number) as table
```

About

Returns a table from the input `table`, with the columns in the order specified by `columnOrder`. Columns that are not specified in the list will not be reordered. If the column doesn't exist, an exception is thrown unless the optional parameter `missingField` specifies an alternative (eg. [MissingField.UseNull](#) or [MissingField.Ignore](#)).

Example 1

Switch the order of the columns [Phone] and [Name] in the table.

Usage

Power Query M

```
Table.ReorderColumns(  
    Table.FromRecords({[CustomerID = 1, Phone = "123-4567", Name = "Bob"]}),  
    {"Name", "Phone"}  
)
```

Output

```
Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]})
```

Example 2

Switch the order of the columns [Phone] and [Address] or use "MissingField.Ignore" in the table. It doesn't change the table because column [Address] doesn't exist.

Usage

Power Query M

```
Table.ReorderColumns(  
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),  
    {"Phone", "Address"},  
    MissingField.Ignore  
)
```

Output

```
Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]})
```


Table.Repeat

06/19/2025

Syntax

```
Table.Repeat(table as table, count as number) as table
```

About

Returns a table with the rows from the input `table` repeated the specified `count` times.

Example 1

Repeat the rows in the table two times.

Usage

Power Query M

```
Table.Repeat(  
    Table.FromRecords({  
        [a = 1, b = "hello"],  
        [a = 3, b = "world"]  
    }),  
    2  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [a = 1, b = "hello"],  
    [a = 3, b = "world"],  
    [a = 1, b = "hello"],  
    [a = 3, b = "world"]  
})
```

Table.ReplaceErrorValues

06/19/2025

Syntax

```
Table.ReplaceErrorValues(table as table, errorReplacement as list) as table
```

About

Replaces the error values in the specified columns of the `table` with the new values in the `errorReplacement` list. The format of the list is {{column1, value1}, ...}. There may only be one replacement value per column, specifying the column more than once will result in an error.

Example 1

Replace the error value with the text "world" in the table.

Usage

Power Query M

```
Table.ReplaceErrorValues(  
    Table.FromRows({{1, "hello"}, {3, ...}}, {"A", "B"}),  
    {"B", "world"}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [A = 1, B = "hello"],  
    [A = 3, B = "world"]  
})
```

Example 2

Replace the error value in column A with the text "hello" and in column B with the text "world" in the table.

Usage

Power Query M

```
Table.ReplaceErrorValues(  
    Table.FromRows({{..., ...}, {1, 2}}, {"A", "B"}),  
    {"A", "hello"}, {"B", "world"}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [A = "hello", B = "world"],  
    [A = 1, B = 2]  
})
```

Table.ReplaceKeys

06/19/2025

Syntax

```
Table.ReplaceKeys(table as table, keys as list) as table
```

About

Replaces the keys of the specified table.

Example 1

Replace the existing keys of a table.

Usage

Power Query M

```
let
    table = Table.FromRecords({
        [Id = 1, Name = "Hello There"],
        [Id = 2, Name = "Good Bye"]
    }),
    tableWithKeys = Table.AddKey(table, {"Id"}, true),
    resultTable = Table.ReplaceKeys(tableWithKeys, {[Columns = {"Id"}, Primary = false]})
in
    resultTable
```

Output

Power Query M

```
Table.FromRecords({
    [Id = 1, Name = "Hello There"],
    [Id = 2, Name = "Good Bye"]
})
```

Table.ReplaceMatchingRows

06/19/2025

Syntax

```
Table.ReplaceMatchingRows(table as table, replacements as list, optional  
equationCriteria as any) as table
```

About

Replaces all the specified rows in the `table` with the provided ones. The rows to replace and the replacements are specified in `replacements`, using {old, new} formatting. An optional `equationCriteria` parameter may be specified to control comparison between the rows of the table.

Example 1

Replace the rows [a = 1, b = 2] and [a = 2, b = 3] with [a = -1, b = -2],[a = -2, b = -3] in the table.

Usage

Power Query M

```
Table.ReplaceMatchingRows(  
    Table.FromRecords({  
        [a = 1, b = 2],  
        [a = 2, b = 3],  
        [a = 3, b = 4],  
        [a = 1, b = 2]  
    }),  
    {  
        {[a = 1, b = 2], [a = -1, b = -2]},  
        {[a = 2, b = 3], [a = -2, b = -3]}  
    }  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [a = -1, b = -2],  
    [a = -2, b = -3],  
    [a = 3, b = 4],  
    [a = -1, b = -2]  
})
```

Related content

[Equation criteria](#)

Table.ReplaceRelationshipIdentity

06/19/2025

Syntax

```
Table.ReplaceRelationshipIdentity(value as any, identity as text) as any
```

About

This function is intended for internal use only.

Table.ReplaceRows

06/19/2025

Syntax

```
Table.ReplaceRows(table as table, offset as number, count as number, rows as list)  
as table
```

About

Replaces a specified number of rows, **count**, in the input **table** with the specified **rows**, beginning after the **offset**. The **rows** parameter is a list of records.

- **table**: The table where the replacement is performed.
- **offset**: The number of rows to skip before making the replacement.
- **count**: The number of rows to replace.
- **rows**: The list of row records to insert into the **table** at the location specified by the **offset**.

Example 1

Starting at position 1, replace 3 rows.

Usage

Power Query M

```
Table.ReplaceRows(  
    Table.FromRecords({  
        [Column1 = 1],  
        [Column1 = 2],  
        [Column1 = 3],  
        [Column1 = 4],  
        [Column1 = 5]  
    }),  
    1,  
    3,  
    {[Column1 = 6], [Column1 = 7]}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [Column1 = 1],  
    [Column1 = 6],  
    [Column1 = 7],  
    [Column1 = 5]  
})
```

Table.ReplaceValue

06/19/2025

Syntax

```
Table.ReplaceValue(table as table, oldValue as any, newValue as any, replacer as function, columnsToSearch as list) as table
```

About

Replaces `oldValue` with `newValue` in the specified columns of the `table`.

Example 1

Replace the text "goodbye" with "world" in column B, matching only the entire value.

Usage

Power Query M

```
Table.ReplaceValue(  
    Table.FromRecords({  
        [A = 1, B = "hello"],  
        [A = 2, B = "goodbye"],  
        [A = 3, B = "goodbyes"]  
    }),  
    "goodbye",  
    "world",  
    Replacer.ReplaceValue,  
    {"B"}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [A = 1, B = "hello"],  
    [A = 2, B = "world"],  
    [A = 3, B = "goodbyes"]  
})
```

Example 2

Replace the text "ur" with "or" in column B, matching any part of the value.

Usage

Power Query M

```
Table.ReplaceValue(  
    Table.FromRecords({  
        [A = 1, B = "hello"],  
        [A = 2, B = "wurld"]  
    }),  
    "ur",  
    "or",  
    Replacer.ReplaceText,  
    {"B"}  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [A = 1, B = "hello"],  
    [A = 2, B = "world"]  
})
```

Example 3

Anonymize the names of US employees.

Usage

Power Query M

```
Table.ReplaceValue(  
    Table.FromRecords({  
        [Name = "Cindy", Country = "US"],  
        [Name = "Bob", Country = "CA"]  
    }),  
    each if [Country] = "US" then [Name] else false,  
    each Text.Repeat("*", Text.Length([Name])),  
    Replacer.ReplaceValue,  
    {"Name"}  
)
```

Output

Power Query M

```
Table.FromRecords({
    [Name = "*****", Country = "US"],
    [Name = "Bob", Country = "CA"]
})
```

Example 4

Anonymize all columns of US employees.

Usage

Power Query M

```
Table.ReplaceValue(
    Table.FromRecords({
        [Name = "Cindy", Country = "US"],
        [Name = "Bob", Country = "CA"]
    }),
    each [Country] = "US",
    "?",
    (currentValue, isUS, replacementValue) =>
        if isUS then
            Text.Repeat(replacementValue, Text.Length(currentValue))
        else
            currentValue,
    {"Name", "Country"}
)
```

Output

Power Query M

```
Table.FromRecords({
    [Name = "?????", Country = "??"],
    [Name = "Bob", Country = "CA"]
})
```

Related content

[Replacer functions](#)

Table.ReverseRows

06/19/2025

Syntax

```
Table.ReverseRows(table as table) as table
```

About

Returns a table with the rows from the input `table` in reverse order.

Example 1

Reverse the rows in the table.

Usage

Power Query M

```
Table.ReverseRows(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    })  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"]  
})
```

Table.RowCount

06/19/2025

Syntax

```
Table.RowCount(table as table) as number
```

About

Returns the number of rows in the `table`.

Example 1

Find the number of rows in the table.

Usage

Power Query M

```
Table.RowCount(  
    Table.FromRecords(  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
    )  
)
```

Output

3

Table.Schema

06/19/2025

Syntax

```
Table.Schema(table as table) as table
```

About

Returns a table describing the columns of `table`.

Each row in the table describes the properties of a column of `table`:

 Expand table

Column Name	Description
<code>Name</code>	The name of the column.
<code>Position</code>	The 0-based position of the column in <code>table</code> .
<code>TypeName</code>	The name of the type of the column.
<code>Kind</code>	The kind of the type of the column.
<code>IsNullable</code>	Whether the column can contain <code>null</code> values.
<code>NumericPrecisionBase</code>	The numeric base (for example, base-2 or base-10) of the <code>NumericPrecision</code> and <code>NumericScale</code> fields.
<code>NumericPrecision</code>	The precision of a numeric column in the base specified by <code>NumericPrecisionBase</code> . This is the maximum number of digits that can be represented by a value of this type (including fractional digits).
<code>NumericScale</code>	The scale of a numeric column in the base specified by <code>NumericPrecisionBase</code> . This is the number of digits in the fractional part of a value of this type. A value of <code>0</code> indicates a fixed scale with no fractional digits. A value of <code>null</code> indicates the scale is not known (either because it is floating or not defined).
<code>DateTimePrecision</code>	The maximum number of fractional digits supported in the seconds portion of a date or time value.
<code>MaxLength</code>	The maximum number of characters permitted in a <code>text</code> column, or the maximum number of bytes permitted in a <code>binary</code> column.

Column Name	Description
IsVariableLength	Indicates whether this column can vary in length (up to MaxLength) or if it is of fixed size.
NativeTypeName	The name of the type of the column in the native type system of the source (for example, nvarchar for SQL Server).
NativeDefaultExpression	The default expression for a value of this column in the native expression language of the source (for example, 42 or newid() for SQL Server).
Description	The description of the column.

Table.SelectColumns

06/19/2025

Syntax

```
Table.SelectColumns(table as table, columns as any, optional missingField as nullable number) as table
```

About

Returns the `table` with only the specified `columns`.

- `table`: The provided table.
- `columns`: The list of columns from the table `table` to return. Columns in the returned table are in the order listed in `columns`.
- `missingField`: (*Optional*) What to do if the column does not exist. Example: [MissingField.UseNull](#) or [MissingField.Ignore](#).

Example 1

Only include column [Name].

Usage

Power Query M

```
Table.SelectColumns(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    "Name"  
)
```

Output

Power Query M

```
Table.FromRecords({
    [Name = "Bob"],
    [Name = "Jim"],
    [Name = "Paul"],
    [Name = "Ringo"]
})
```

Example 2

Only include columns [CustomerID] and [Name].

Usage

Power Query M

```
Table.SelectColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    {"CustomerID", "Name"}
)
```

Output

```
Table.FromRecords({[CustomerID = 1, Name = "Bob"]})
```

Example 3

If the included column does not exist, the default result is an error.

Usage

Power Query M

```
Table.SelectColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    "NewColumn"
)
```

Output

```
[Expression.Error] The field 'NewColumn' of the record wasn't found.
```

Example 4

If the included column does not exist, option `MissingField.UseNull` creates a column of null values.

Usage

Power Query M

```
Table.SelectColumns(  
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),  
    {"CustomerID", "NewColumn"},  
    MissingField.UseNull  
)
```

Output

```
Table.FromRecords({[CustomerID = 1, NewColumn = null]})
```

Table.SelectRows

06/19/2025

Syntax

```
Table.SelectRows(table as table, condition as function) as table
```

About

Returns a table of rows from the `table`, that matches the selection `condition`.

Example 1

Select the rows in the table where the values in [CustomerID] column are greater than 2.

Usage

Power Query M

```
Table.SelectRows(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    each [CustomerID] > 2  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
})
```

Example 2

Select the rows in the table where the names do not contain a "B".

Usage

Power Query M

```
Table.SelectRows(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    each not Text.Contains([Name], "B")  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
})
```

Table.SelectRowsWithErrors

06/19/2025

Syntax

```
Table.SelectRowsWithErrors(table as table, optional columns as nullable list) as table
```

About

Returns a table with only those rows of the input table that contain an error in at least one of the cells. If a columns list is specified, then only the cells in the specified columns are inspected for errors.

Example 1

Select names of customers with errors in their rows.

Usage

Power Query M

```
Table.SelectRowsWithErrors(  
    Table.FromRecords({  
        [CustomerID = ..., Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    })  
)[Name]
```

Output

```
{"Bob"}
```

Table.SingleRow

06/19/2025

Syntax

```
Table.SingleRow(table as table) as record
```

About

Returns the single row in the one row `table`. If the `table` does not contain exactly one row, an error is raised.

Example 1

Return the single row in the table.

Usage

Power Query M

```
Table.SingleRow(Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}))
```

Output

```
[CustomerID = 1, Name = "Bob", Phone = "123-4567"]
```

Table.Skip

06/19/2025

Syntax

```
Table.Skip(table as table, optional countOrCondition as any) as table
```

About

Returns a table that does not contain the first specified number of rows, **countOrCondition**, of the table **table**. The number of rows skipped depends on the optional parameter **countOrCondition**.

- If **countOrCondition** is omitted only the first row is skipped.
- If **countOrCondition** is a number, that many rows (starting at the top) will be skipped.
- If **countOrCondition** is a condition, the rows that meet the condition will be skipped until a row does not meet the condition.

Example 1

Skip the first row of the table.

Usage

Power Query M

```
Table.Skip(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]  
    }),  
    1  
)
```

Output

Power Query M


```
Table.FromRecords({
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
})
```

Example 2

Skip the first two rows of the table.

Usage

Power Query M

```
Table.Skip(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    2
)
```

Output

Power Query M

```
Table.FromRecords({
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
    [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
})
```

Example 3

Skip the first rows where [Price] > 25 of the table.

Usage

Power Query M

```
Table.Skip(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
```

```

        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Band-aids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
    }),
    each [Price] > 25
)

```

Output

Power Query M

```

Table.FromRecords({
    [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5],
    [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25],
    [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200],
    [OrderID = 5, CustomerID = 3, Item = "Band-aids", Price = 2],
    [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20],
    [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
    [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100],
    [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
})

```

Table.Sort

06/19/2025

Syntax

```
Table.Sort(table as table, comparisonCriteria as any) as table
```

About

Sorts the `table` using the list of one or more column names and optional `comparisonCriteria` in the form `{ { col1, comparisonCriteria }, {col2} }`.

Example 1

Sort the table on column "OrderID".

Usage

Power Query M

```
Table.Sort(  
    Table.FromRecords(  
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],  
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],  
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],  
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],  
        [OrderID = 5, CustomerID = 3, Item = "Band-aids", Price = 2.0],  
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],  
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],  
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],  
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]  
    ),  
    {"OrderID"}  
)
```

Output

Power Query M

```
Table.FromRecords(  
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100],  
    [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5],  
    [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25],  
    [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200],  
    [OrderID = 5, CustomerID = 3, Item = "Band-aids", Price = 2],  
    [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20],  
    [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],  
    [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100],  
    [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]  
)
```

```

[OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200],
[OrderID = 5, CustomerID = 3, Item = "Band-aids", Price = 2],
[OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20],
[OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
[OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100],
[OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
})

```

Example 2

Sort the table on column "OrderID" in descending order.

Usage

Power Query M

```

Table.Sort(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Band-aids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
    }),
    {"OrderID", Order.Descending}
)

```

Output

Power Query M

```

Table.FromRecords({
    [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25],
    [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100],
    [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
    [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20],
    [OrderID = 5, CustomerID = 3, Item = "Band-aids", Price = 2],
    [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200],
    [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25],
    [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5],
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100]
})

```

Example 3

Sort the table on column "CustomerID" then "OrderID", with "CustomerID" being in ascending order.

Usage

Power Query M

```
Table.Sort(  
    Table.FromRecords({  
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],  
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],  
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],  
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],  
        [OrderID = 5, CustomerID = 3, Item = "Band-aids", Price = 2.0],  
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],  
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],  
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],  
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]  
    }  
),  
    {  
        {"CustomerID", Order.Ascending},  
        "OrderID"  
    }  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100],  
    [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5],  
    [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20],  
    [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25],  
    [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200],  
    [OrderID = 5, CustomerID = 3, Item = "Band-aids", Price = 2],  
    [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],  
    [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100],  
    [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]  
})
```

Related content

[Comparison criteria](#)

Table.Split

06/19/2025

Syntax

```
Table.Split(table as table, pageSize as number) as list
```

About

Splits `table` into a list of tables where the first element of the list is a table containing the first `pageSize` rows from the source table, the next element of the list is a table containing the next `pageSize` rows from the source table, and so on.

Example 1

Split a table of five records into tables with two records each.

Usage

Power Query M

```
let
    Customers = Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Cristina", Phone = "232-1550"],
        [CustomerID = 5, Name = "Anita", Phone = "530-1459"]
    })
in
    Table.Split(Customers, 2)
```

Output

Power Query M

```
{
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"]
    }),
    Table.FromRecords({
```

```
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],  
        [CustomerID = 4, Name = "Cristina", Phone = "232-1550"]  
    }),  
    Table.FromRecords({  
        [CustomerID = 5, Name = "Anita", Phone = "530-1459"]  
    })  
}
```

Table.SplitAt

06/19/2025

```
Table.SplitAt(table as table, count as number) as list
```

About

Returns a list containing two tables: a table with the first N rows of `table` (as specified by `count`) and a table containing the remaining rows of `table`. If the tables of the resulting list are enumerated exactly once and in order, the function will enumerate `table` only once.

Example 1

Return the first two rows of the table and the remaining rows of the table.

Usage

Power Query M

```
Table.SplitAt(#table({"a", "b", "c"}, {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}), 2)
```

Output

Power Query M

```
{
  #table({"a", "b", "c"}, {{1, 2, 3}, {4, 5, 6}}),
  #table({"a", "b", "c"}, {{7, 8, 9}})
}
```


Table.SplitColumn

06/19/2025

Syntax

```
Table.SplitColumn(table as table, sourceColumn as text, splitter as function,  
optional columnNamesOrNumber as any, optional default as any, optional  
extraColumns as any) as table
```

About

Splits the specified column into a set of additional columns using the specified splitter function.

- **table**: The table containing the column to split.
- **sourceColumn**: The name of the column to split.
- **splitter**: The [splitter function](#) used to split the column (for example, [Splitter.SplitTextByDelimiter](#) or [Splitter.SplitTextByPositions](#)).
- **columnNamesOrNumber**: Either a list of new column names to create, or the number of new columns.
- **default**: Overrides the value used when there aren't enough split values to fill all of the new columns. The default for this parameter is `null`.
- **extraColumns**: Specifies what to do if there might be more split values than the number of new columns. You can pass an [ExtraValues.Type](#) enumeration value to this parameter. The default is `ExtraValues.Ignore`.

Example 1

Split the name column into first name and last name.

Usage

Power Query M

```
let  
    Source = #table(type table[CustomerID = number, Name = text, Phone = text],  
        {  
            {1, "Bob White", "123-4567"},  
            {2, "Jim Smith", "987-6543"},  
            {3, "Paul", "543-7890"},  
            {4, "Cristina Best", "232-1550"}  
        })
```

```

    }),
    SplitColumns = Table.SplitColumn(
        Source,
        "Name",
        Splitter.SplitTextByDelimiter(" ")
    )
in
    SplitColumns

```

Output

Power Query M

```

#table(type table[CustomerID = number, Name.1 = text, Name.2 = text, Phone = text],
{
    {1, "Bob", "White", "123-4567"},
    {2, "Jim", "Smith", "987-6543"},
    {3, "Paul", null, "543-7890"},
    {4, "Cristina", "Best", "232-1550"}
})

```

Example 2

Split the name column into first name and last name, then rename the new columns.

Usage

Power Query M

```

let
    Source = #table(type table[CustomerID = number, Name = text, Phone = text],
        {
            {1, "Bob White", "123-4567"},
            {2, "Jim Smith", "987-6543"},
            {3, "Paul", "543-7890"},
            {4, "Cristina Best", "232-1550"}
        }
    ),
    SplitColumns = Table.SplitColumn(
        Source,
        "Name",
        Splitter.SplitTextByDelimiter(" "),
        {"First Name", "Last Name"}
    )
in
    SplitColumns

```

Output

Power Query M

```
#table(type table[CustomerID = number, First Name = text, Last Name = text, Phone
= text],
{
  {1, "Bob", "White", "123-4567"},
  {2, "Jim", "Smith", "987-6543"},
  {3, "Paul", null, "543-7890"},
  {4, "Cristina", "Best", "232-1550"}
})
```

Example 3

Split the name column into first name and last name, rename the new columns, and fill in any blanks with "-No Entry-".

Usage

Power Query M

```
let
    Source = #table(type table[CustomerID = number, Name = text, Phone = text],
        {
            {1, "Bob White", "123-4567"},
            {2, "Jim Smith", "987-6543"},
            {3, "Paul", "543-7890"},
            {4, "Cristina Best", "232-1550"}
        }
    ),
    SplitColumns = Table.SplitColumn(
        Source,
        "Name",
        Splitter.SplitTextByDelimiter(" "),
        {"First Name", "Last Name"},
        "-No Entry-"
    )
in
    SplitColumns
```

Output

Power Query M

```
#table(type table[CustomerID = number, First Name = text, Last Name = text, Phone
= text],
{
  {1, "Bob", "White", "123-4567"},
  {2, "Jim", "Smith", "987-6543"},
  {3, "Paul", "-No Entry-", "543-7890"},
  {4, "Cristina", "Best", "232-1550"}
})
```

Example 4

Split the name column into first name and last name, then rename the new columns. Because there might be more values than the number of available columns, make the last name column a list that includes all values after the first name.

Usage

Power Query M

```
let
    Source = #table(type table[CustomerID = number, Name = text, Phone = text],
        {
            {1, "Bob White", "123-4567"},
            {2, "Jim Smith", "987-6543"},
            {3, "Paul Green", "543-7890"},
            {4, "Cristina J. Best", "232-1550"}
        }
    ),
    SplitColumns = Table.SplitColumn(
        Source,
        "Name",
        Splitter.SplitTextByDelimiter(" "),
        {"First Name", "Last Name"},
        null,
        ExtraValues.List)
in
    SplitColumns
```

Output

Power Query M

```
#table(type table[CustomerID = number, First Name = text, Last Name = text, Phone = text],
{
    {1, "Bob", {"White"}, "123-4567"},
    {2, "Jim", {"Smith"}, "987-6543"},
    {3, "Paul", {"Green"}, "543-7890"},
    {4, "Cristina", {"J.", "Best"}, "232-1550"}
})
```

Table.StopFolding

06/19/2025

Syntax

```
Table.StopFolding(table as table) as table
```

About

Prevents any downstream operations from being run against the original source of the data in `table`.

Example 1

Fetches data from a SQL table in a way that prevents any downstream operations from running as a query on the SQL server.

Usage

Power Query M

```
let
    Source = Sql.Database("SomeSQLServer", "MyDb"),
    MyTable = Source[[Item="MyTable"]][Data],
    MyLocalTable = Table.StopFolding(MyTable)
in
    MyLocalTable
```

Output

`table`

Table.ToColumns

06/19/2025

Syntax

```
Table.ToColumns(table as table) as list
```

About

Creates a list of nested lists from the table, `table`. Each list item is an inner list that contains the column values.

Example 1

Create a list of the column values from the table.

Usage

Power Query M

```
Table.ToColumns(  
    Table.FromRecords(  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"]  
    )  
)
```

Output

```
{{1, 2}, {"Bob", "Jim"}, {"123-4567", "987-6543"}}
```

Table.ToList

06/19/2025

Syntax

```
Table.ToList(table as table, optional combiner as nullable function) as list
```

About

Converts a table into a list by applying the specified combining function to each row of values in the table.

Example 1

Combine the text of each row with a comma.

Usage

Power Query M

```
Table.ToList(  
    Table.FromRows({  
        {Number.ToText(1), "Bob", "123-4567"},  
        {Number.ToText(2), "Jim", "987-6543"},  
        {Number.ToText(3), "Paul", "543-7890"}  
    }),  
    Combiner.CombineTextByDelimiter(",")  
)
```

Output

```
{"1,Bob,123-4567", "2,Jim,987-6543", "3,Paul,543-7890"}
```

Related content

[Combiner functions](#)

Table.ToRecords

06/19/2025

Syntax

```
Table.ToRecords(table as table) as list
```

About

Converts a table, `table`, to a list of records.

Example 1

Convert the table to a list of records.

Usage

Power Query M

```
Table.ToRecords(  
    Table.FromRows(  
        {  
            {1, "Bob", "123-4567"},  
            {2, "Jim", "987-6543"},  
            {3, "Paul", "543-7890"}  
        },  
        {"CustomerID", "Name", "Phone"}  
    )  
)
```

Output

Power Query M

```
{  
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
}
```


Table.ToRows

06/19/2025

Syntax

```
Table.ToRows(table as table) as list
```

About

Creates a list of nested lists from the table, `table`. Each list item is an inner list that contains the row values.

Example 1

Create a list of the row values from the table.

Usage

Power Query M

```
Table.ToRows(  
    Table.FromRecords({  
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],  
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],  
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]  
    })  
)
```

Output

Power Query M

```
{  
    {1, "Bob", "123-4567"},  
    {2, "Jim", "987-6543"},  
    {3, "Paul", "543-7890"}  
}
```

Table.TransformColumnNames

06/19/2025

Syntax

```
Table.TransformColumnNames(table as table, nameGenerator as function, optional  
options as nullable record) as table
```

About

Transforms column names by using the given `nameGenerator` function. Valid options:

`MaxLength` specifies the maximum length of new column names. If the given function results with a longer column name, the long name will be trimmed.

`Comparer` is used to control the comparison while generating new column names. Comparers can be used to provide case-insensitive or culture and locale-aware comparisons.

The following built-in comparers are available in the formula language:

- `Comparer.Ordinal`: Used to perform an exact ordinal comparison
- `Comparer.OrdinalIgnoreCase`: Used to perform an exact ordinal case-insensitive comparison
- `Comparer.FromCulture`: Used to perform a culture-aware comparison

Example 1

Remove the `#(tab)` character from column names

Usage

Power Query M

```
Table.TransformColumnNames(Table.FromRecords({["Col#(tab)umn" = 1]}), Text.Clean)
```

Output

```
Table.FromRecords({[Column = 1]})
```

Example 2

Transform column names to generate case-insensitive names of length 6.

Usage

Power Query M

```
Table.TransformColumnNames(  
    Table.FromRecords({[ColumnNum = 1, cOlumnum = 2, coLumnNUM = 3]}),  
    Text.Clean,  
    [MaxLength = 6, Comparer = Comparer.OrdinalIgnoreCase]  
)
```

Output

```
Table.FromRecords({[Column = 1, cOlum1 = 2, coLum2 = 3]})
```

Related content

- [How culture affects text formatting](#)

Table.TransformColumns

06/19/2025

Syntax

```
Table.TransformColumns(table as table, transformOperations as list, optional  
defaultTransformation as nullable function, optional missingField as nullable  
number) as table
```

About

Transforms `table` by applying each column operation listed in `transformOperations` (where the format is { column name, transformation } or { column name, transformation, new column type }). If a `defaultTransformation` is specified, it will be applied to all columns not listed in `transformOperations`. If a column listed in `transformOperations` doesn't exist, an exception is thrown unless the optional parameter `missingField` specifies an alternative (for example, [MissingField.UseNull](#) or [MissingField.Ignore](#)).

Example 1

Convert the text values in column [A] to number values, and the number values in column [B] to text values.

Usage

Power Query M

```
Table.TransformColumns(  
    Table.FromRecords({  
        [A = "1", B = 2],  
        [A = "5", B = 10]  
    }),  
    {  
        {"A", Number.FromText},  
        {"B", Text.From}  
    }  
)
```

Output

Power Query M

```
Table.FromRecords({
    [A = 1, B = "2"],
    [A = 5, B = "10"]
})
```

Example 2

Convert the number values in missing column [X] to text values, ignoring columns which don't exist.

Usage

Power Query M

```
Table.TransformColumns(
    Table.FromRecords({
        [A = "1", B = 2],
        [A = "5", B = 10]
    }),
    {"X", Number.FromText},
    null,
    MissingField.Ignore
)
```

Output

Power Query M

```
Table.FromRecords({
    [A = "1", B = 2],
    [A = "5", B = 10]
})
```

Example 3

Convert the number values in missing column [X] to text values, defaulting to null on columns which don't exist.

Usage

Power Query M

```
Table.TransformColumns(
    Table.FromRecords({
        [A = "1", B = 2],
```

```

        [A = "5", B = 10]
    }},
    {"X", Number.FromText},
    null,
    MissingField.UseNull
)

```

Output

Power Query M

```

Table.FromRecords({
    [A = "1", B = 2, X = null],
    [A = "5", B = 10, X = null]
})

```

Example 4

Increment the number values in column [B] and convert them to text values, and convert all other columns to numbers.

Usage

Power Query M

```

Table.TransformColumns(
    Table.FromRecords({
        [A = "1", B = 2],
        [A = "5", B = 10]
    }),
    {"B", each Text.From(_ + 1), type text},
    Number.FromText
)

```

Output

Power Query M

```

Table.FromRecords({
    [A = 1, B = "3"],
    [A = 5, B = "11"]
})

```

Table.TransformColumnTypes

06/19/2025

Syntax

```
Table.TransformColumnTypes(table as table, typeTransformations as list, optional  
culture as nullable text) as table
```

About

Returns a table by applying the transform operations to the specified columns using an optional culture.

- **table**: The input table to transform.
- **typeTransformations**: The type transformations to apply. The format for a single transformation is { column name, type value }. A list of transformations can be used to change the types of more than one column at a time. If a column doesn't exist, an error is raised.
- **culture**: (Optional) The culture to use when transforming the column types (for example, "en-US"). If a record is specified for **culture**, it can contain the following fields:
 - **Culture**: The culture to use when transforming the column types (for example, "en-US").
 - **MissingField**: If a column doesn't exist, an error is raised unless this field provides an alternative behavior (for example, [MissingField.UseNull](#) or [MissingField.Ignore](#)).

The type value in the **typeTransformations** parameter can be **any**, all of the **number** types, **text**, all of the **date**, **time**, **datetime**, **datetimezone**, and **duration** types, **logical**, or **binary**. The **list**, **record**, **table**, or **function** types aren't valid for this parameter.

For each column listed in **typeTransformations**, the ".From" method corresponding to the specified type value is normally used to perform the transformation. For example, if a [Currency.Type](#) type value is given for a column, the transformation function [Currency.From](#) is applied to each value in that column.

Example 1

Transform the number values in the first column to text values.

Usage

Power Query M

```
let
    Source = #table(type table [a = number, b = number],
        {
            {1, 2},
            {3, 4}
        }
    ),
    #"Transform Column" = Table.TransformColumnTypes(
        Source,
        {"a", type text}
    )
in
    #"Transform Column"
```

Output

Power Query M

```
#table(type table [a = text, b = number],
{
    {"1", 2},
    {"3", 4}
})
```

Example 2

Transform the dates in the table to their French text equivalents.

Usage

Power Query M

```
let
    Source = #table(type table [Company ID = text, Country = text, Date = date],
        {
            {"JS-464", "USA", #date(2024, 3, 24)},
            {"LT-331", "France", #date(2024, 10, 5)},
            {"XE-100", "USA", #date(2024, 5, 21)},
            {"RT-430", "Germany", #date(2024, 1, 18)},
            {"LS-005", "France", #date(2023, 12, 31)},
            {"UW-220", "Germany", #date(2024, 2, 25)}
        }
    ),
    #"Transform Column" = Table.TransformColumnTypes(
        Source,
        {"Date", type text},
        "fr-FR"
    )
in
    #"Transform Column"
```



```
)  
in  
#"Transform Column"
```

Output

Power Query M

```
#table(type table [Company ID = text, Country = text, Date = text],  
{  
    {"JS-464", "USA", "24/03/2024"},  
    {"LT-331", "France", "05/10/2024"},  
    {"XE-100", "USA", "21/05/2024"},  
    {"RT-430", "Germany", "18/01/2024"},  
    {"LS-005", "France", "31/12/2023"},  
    {"UW-220", "Germany", "25/02/2024"}  
})
```

Example 3

Transform the dates in the table to their German text equivalents, and the values in the table to percentages.

Usage

Power Query M

```
let  
    Source = #table(type table [Date = date, Customer ID = text, Value = number],  
    {  
        {#date(2024, 3, 12), "134282", .24368},  
        {#date(2024, 5, 30), "44343", .03556},  
        {#date(2023, 12, 14), "22", .3834}  
    }  
),  
    #"Transform Columns" = Table.TransformColumnTypes(  
        Source,  
        {{"Date", type text}, {"Value", Percentage.Type}},  
        "de-DE")  
in  
    #"Transform Columns"
```

Output

Power Query M

```
#table(type table [Date = text, Customer ID = text, Value = Percentage.Type],  
{  
    {"12.03.2024", "134282", .24368},
```

```
{ "30.05.2024", "44343", .03556 },  
{ "14.12.2023", "22", .3834 }  
})
```

Related content

- [Types and type conversion](#)
- [How culture affects text formatting](#)

Table.TransformRows

06/19/2025

Syntax

```
Table.TransformRows(table as table, transform as function) as list
```

About

Creates a `list` by applying the `transform` operation to each row in `table`.

Example 1

Transform the rows of a table into a list of numbers.

Usage

Power Query M

```
Table.TransformRows(  
    Table.FromRecords({  
        [a = 1],  
        [a = 2],  
        [a = 3],  
        [a = 4],  
        [a = 5]  
    }  
),  
each [a]  
)
```

Output

```
{1, 2, 3, 4, 5}
```

Example 2

Transform the rows of a numeric table into textual records.

Usage

Power Query M

```
Table.TransformRows(  
    Table.FromRecords({  
        [a = 1],  
        [a = 2],  
        [a = 3],  
        [a = 4],  
        [a = 5]  
    }  
),  
(row) as record => [B = Number.ToText(row[a])]  
)
```

Output

Power Query M

```
{  
    [B = "1"],  
    [B = "2"],  
    [B = "3"],  
    [B = "4"],  
    [B = "5"]  
}
```

Table.Transpose

06/19/2025

Syntax

```
Table.Transpose(table as table, optional columns as any) as table
```

About

Makes columns into rows and rows into columns.

Example 1

Make the rows of the table of name-value pairs into columns.

Usage

Power Query M

```
Table.Transpose(  
    Table.FromRecords({  
        [Name = "Full Name", Value = "Fred"],  
        [Name = "Age", Value = 42],  
        [Name = "Country", Value = "UK"]  
    })  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [Column1 = "Full Name", Column2 = "Age", Column3 = "Country"],  
    [Column1 = "Fred", Column2 = 42, Column3 = "UK"]  
})
```

Table.Unpivot

06/19/2025

Syntax

```
Table.Unpivot(table as table, pivotColumns as list, attributeColumn as text,  
valueColumn as text) as table
```

About

Translates a set of columns in a table into attribute-value pairs, combined with the rest of the values in each row.

Example 1

Take the columns "a", "b", and "c" in the table `([ [ key = "x", a = 1, b = null, c = 3 ], [ key = "y", a = 2, b = 4, c = null ] ])` and unpivot them into attribute-value pairs.

Usage

Power Query M

```
Table.Unpivot(  
    Table.FromRecords({  
        [key = "x", a = 1, b = null, c = 3],  
        [key = "y", a = 2, b = 4, c = null]  
    }),  
    {"a", "b", "c"},  
    "attribute",  
    "value"  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [key = "x", attribute = "a", value = 1],  
    [key = "x", attribute = "c", value = 3],  
    [key = "y", attribute = "a", value = 2],  
    [key = "y", attribute = "b", value = 4]  
})
```


Table.UnpivotOtherColumns

06/19/2025

Syntax

```
Table.UnpivotOtherColumns(table as table, pivotColumns as list, attributeColumn as text, valueColumn as text) as table
```

About

Translates all columns other than a specified set into attribute-value pairs, combined with the rest of the values in each row.

Example 1

Translates all columns other than a specified set into attribute-value pairs, combined with the rest of the values in each row.

Usage

Power Query M

```
Table.UnpivotOtherColumns(  
    Table.FromRecords({  
        [key = "key1", attribute1 = 1, attribute2 = 2, attribute3 = 3],  
        [key = "key2", attribute1 = 4, attribute2 = 5, attribute3 = 6]  
    }),  
    {"key"},  
    "column1",  
    "column2"  
)
```

Output

Power Query M

```
Table.FromRecords({  
    [key = "key1", column1 = "attribute1", column2 = 1],  
    [key = "key1", column1 = "attribute2", column2 = 2],  
    [key = "key1", column1 = "attribute3", column2 = 3],  
    [key = "key2", column1 = "attribute1", column2 = 4],  
    [key = "key2", column1 = "attribute2", column2 = 5],  
    [key = "key2", column1 = "attribute3", column2 = 6]
```



```
[key = "key2", column1 = "attribute3", column2 = 6]  
})
```

Table.View

06/19/2025

Syntax

```
Table.View(table as nullable table, handlers as record) as table
```

About

Returns a view of `table` where the functions specified in `handlers` are used in lieu of the default behavior of an operation when the operation is applied to the view.

If `table` is provided, all handler functions are optional. If `table` isn't provided, the `GetType` and `GetRows` handler functions are required. If a handler function isn't specified for an operation, the default behavior of the operation is applied to `table` instead (except in the case of `GetExpression`).

Handler functions must return a value that is semantically equivalent to the result of applying the operation against `table` (or the resulting view in the case of `GetExpression`).

If a handler function raises an error, the default behavior of the operation is applied to the view.

Table.View can be used to implement folding to a data source—the translation of M queries into source-specific queries (for example, to create T-SQL statements from M queries).

Refer to the published [Power Query custom connector documentation](#) for a more complete description of **Table.View**.

Example 1

Create a basic view that doesn't require accessing the rows in order to determine the type or the row count.

Usage

Power Query M

```
Table.View(  
    null,
```

```
[
    GetType = () => type table [CustomerID = number, Name = text, Phone =
nullable text],
    GetRows = () => Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone =
"123-4567"]}),
    GetRowCount = () => 1
]
```

Output

```
Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]})
```

Table.ViewError

06/19/2025

Syntax

```
Table.ViewError(errorRecord as record) as record
```

About

Creates a modified error record from `errorRecord` which won't trigger a fallback when thrown by a handler defined on a view (via [Table.View](#)).

Table.ViewFunction

06/19/2025

Syntax

```
Table.ViewFunction(function as function) as function
```

About

Creates a view function based on **function** that can be handled in a view created by [Table.View](#).

The **OnInvoke** handler of [Table.View](#) can be used to define a handler for the view function.

As with the handlers for built-in operations, if no **OnInvoke** handler is specified, or if it does not handle the view function, or if an error is raised by the handler, **function** is applied on top of the view.

Refer to the published [Power Query custom connector documentation](#) for a more complete description of **Table.View** and custom view functions.

Table.WithErrorContext

06/19/2025

Syntax

```
Table.WithErrorContext(value as any, context as text) as any
```

About

This function is intended for internal use only.

Tables.GetRelationships

06/19/2025

Syntax

```
Tables.GetRelationships(tables as table, optional dataColumn as nullable text) as  
table
```

About

Gets the relationships among a set of tables. The set `tables` is assumed to have a structure similar to that of a navigation table. The column defined by `dataColumn` contains the actual data tables.

#table

06/19/2025

Syntax

```
#table(columns as any, rows as any) as any
```

About

Creates a table value from `columns` and `rows`. The `columns` value can be a list of column names, a table type, a number of columns, or null. The `rows` value is a list of lists, where each element contains the column values for a single row.

Example 1

Create an empty table.

Usage

Power Query M

```
#table({}, {})
```

Output

Power Query M

```
#table({}, {})
```

Example 2

Create a table by inferring the number of columns from the first row.

Usage

Power Query M

```
#table(null, {"Betty", 90.3}, {"Carl", 89.5})
```


Output

Power Query M

```
#table({"Column1", "Column2"}, {"Betty", 90.3}, {"Carl", 89.5}))
```

Example 3

Create a table by specifying the number of columns.

Usage

Power Query M

```
#table(2, {"Betty", 90.3}, {"Carl", 89.5}))
```

Output

Power Query M

```
#table({"Column1", "Column2"}, {"Betty", 90.3}, {"Carl", 89.5}))
```

Example 4

Create a table by providing a list of column names.

Usage

Power Query M

```
#table({"Name", "Score"}, {"Betty", 90.3}, {"Carl", 89.5}))
```

Output

Power Query M

```
#table({"Name", "Score"}, {"Betty", 90.3}, {"Carl", 89.5}))
```

Example 5

Create a table with an explicit type.

Usage

Power Query M

```
#table(type table [Name = text, Score = number], {"Betty", 90.3}, {"Carl", 89.5}))
```

Output

Power Query M

```
#table(type table [Name = text, Score = number], {"Betty", 90.3}, {"Carl", 89.5}))
```

Related content

- [Types and type conversion](#)

Text functions

06/09/2025

These functions create and manipulate text values.

Information

 Expand table

Name	Description
Text.InferNumberType	Infers the granular number type (<code>Int64.Type</code> , <code>Double.Type</code> , and so on) of a number encoded in text.
Text.Length	Returns the number of characters in a text value.

Text Comparisons

 Expand table

Name	Description
Character.FromNumber	Converts a number to a text character.
Character.ToNumber	Converts a character to a number value.
Guid.From	Returns a GUID value from the given value.
Json.FromValue	Produces a JSON representation of a given value.
Text.From	Creates a text value from the given value.
Text.FromBinary	Decodes data from a binary value in to a text value using an encoding.
Text.NewGuid	Returns a GUID value as a text value.
Text.ToBinary	Encodes a text value into binary value using an encoding.
Text.ToList	Returns a list of characters from a text value.
Value.FromText	Creates a strongly-typed value from a textual representation.

Extraction

[Expand table](#)

Name	Description
Text.At	Returns a character starting at a zero-based offset.
Text.Middle	Returns the substring up to a specific length.
Text.Range	Returns a number of characters from a text value starting at a zero-based offset and for count number of characters.
Text.Start	Returns the count of characters from the start of a text value.
Text.End	Returns the number of characters from the end of a text value.

Modification

[Expand table](#)

Name	Description
Text.Insert	Inserts one text value into another at a given position.
Text.Remove	Removes all occurrences of the given character or list of characters from the input text value.
Text.RemoveRange	Removes count characters at a zero-based offset from a text value.
Text.Replace	Replaces all occurrences of a substring with a new text value.
Text.ReplaceRange	Replaces length characters in a text value starting at a zero-based offset with the new text value.
Text.Select	Selects all occurrences of the given character or list of characters from the input text value.

Membership

[Expand table](#)

Name	Description
Text.Contains	Returns <code>true</code> if a text value substring was found within a text value string; otherwise, <code>false</code> .
Text.EndsWith	Returns a logical value indicating whether a text value substring was found at the end of a string.

Name	Description
Text.PositionOf	Returns the first position of the value (-1 if not found).
Text.PositionOfAny	Returns the first position in the text value of any listed character (-1 if not found).
Text.StartsWith	Returns a logical value indicating whether a text value substring was found at the beginning of a string.

Transformations

 Expand table

Name	Description
Text.AfterDelimiter	Returns the portion of text after the specified delimiter.
Text.BeforeDelimiter	Returns the portion of text before the specified delimiter.
Text.BetweenDelimiters	Returns the portion of text between the specified <code>startDelimiter</code> and <code>endDelimiter</code> .
Text.Clean	Returns the original text value with non-printable characters removed.
Text.Combine	Returns a text value that is the result of joining all text values with each value separated by a separator.
Text.Lower	Returns the lowercase of a text value.
Text.PadEnd	Returns text of a specified length by padding the end of the given text.
Text.PadStart	Returns text of a specified length by padding the start of the given text.
Text.Proper	Returns a text value with first letters of all words converted to uppercase.
Text.Repeat	Returns a text value composed of the input text value repeated a number of times.
Text.Reverse	Reverses the provided text.
Text.Split	Returns a list containing parts of a text value that are delimited by a separator text value.
Text.SplitAny	Returns a list containing parts of a text value that are delimited by any separator text values.
Text.Trim	Removes all the specified leading and trailing characters.
Text.TrimEnd	Removes all specified trailing characters.

Name	Description
Text.TrimStart	Removes all specified leading characters.
Text.Upper	Returns the uppercase of a text value.

Character.FromNumber

06/19/2025

Syntax

```
Character.FromNumber(number as nullable number) as nullable text
```

About

Returns the character equivalent of the number.

The provided `number` should be a 21-bit Unicode code point.

Example 1

Convert a number to its equivalent character value.

Usage

```
Power Query M
```

```
Character.FromNumber(9)
```

Output

```
"#(tab)"
```

Example 2

Convert a character to a number and back again.

Usage

```
Power Query M
```

```
Character.FromNumber(Character.ToNumber("A"))
```

Output

"A"

Example 3

Convert the hexadecimal code point for the "grinning face" emoticon to its equivalent UTF-16 surrogate pair.

Usage

Power Query M

```
Character.FromNumber(0x1F600)
```

Output

"#(0001F600)"

Character.ToNumber

06/19/2025

Syntax

```
Character.ToNumber(character as nullable text) as nullable number
```

About

Returns the number equivalent of the character, `character`.

The result will be the 21-bit Unicode code point represented by the provided character or surrogate pair.

Example 1

Convert a character to its equivalent number value.

Usage

Power Query M

```
Character.ToNumber("#(tab)")
```

Output

9

Example 2

Convert the UTF-16 surrogate pair for the "grinning face" emoticon to its equivalent hexadecimal code point.

Usage

Power Query M

```
Number.ToText(Character.ToNumber("#(0001F600)"), "X")
```

Output

```
"1F600"
```

Guid.From

06/19/2025

Syntax

```
Guid.From(value as nullable text) as nullable text
```

About

Returns a `Guid.Type` value from the given `value`. If the given `value` is `null`, **Guid.From** returns `null`. A check will be performed to determine if the given `value` is in an acceptable format. Acceptable formats provided in the examples.

Example 1

The Guid can be provided as 32 contiguous hexadecimal digits.

Usage

Power Query M

```
Guid.From("05FE1DADC8C24F3BA4C2D194116B4967")
```

Output

```
"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"
```

Example 2

The Guid can be provided as 32 hexadecimal digits separated by hyphens into blocks of 8-4-4-4-12.

Usage

Power Query M

```
Guid.From("05FE1DAD-C8C2-4F3B-A4C2-D194116B4967")
```

Output

```
"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"
```

Example 3

The Guid can be provided as 32 hexadecimal digits separated by hyphens and enclosed in braces.

Usage

```
Power Query M
```

```
Guid.From("{05FE1DAD-C8C2-4F3B-A4C2-D194116B4967}")
```

Output

```
"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"
```

Example 4

The Guid can be provided as 32 hexadecimal digits separated by hyphens and enclosed by parentheses.

Usage

```
Power Query M
```

```
Guid.From("(05FE1DAD-C8C2-4F3B-A4C2-D194116B4967)")
```

Output

```
"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"
```

Json.FromValue

06/19/2025

Syntax

```
Json.FromValue(value as any, optional encoding as nullable number) as binary
```

About

Produces a JSON representation of a given value `value` with a text encoding specified by `encoding`. If `encoding` is omitted, UTF8 is used. Values are represented as follows:

- Null, text and logical values are represented as the corresponding JSON types
- Numbers are represented as numbers in JSON, except that `#infinity`, `-#infinity` and `#nan` are converted to null
- Lists are represented as JSON arrays
- Records are represented as JSON objects
- Tables are represented as an array of objects
- Dates, times, datetimes, datetimezones and durations are represented as ISO-8601 text
- Binary values are represented as base-64 encoded text
- Types and functions produce an error

Example 1

Convert a complex value to JSON.

Usage

Power Query M

```
Text.FromBinary(Json.FromValue([A = {1, true, "3"}, B = #date(2012, 3, 25)]))
```

Output

```
"{"A":[1,true,"3"],"B":"2012-03-25"}"
```

Text.AfterDelimiter

06/19/2025

Syntax

```
Text.AfterDelimiter(text as nullable text, delimiter as text, optional index as any) as any
```

About

Returns the portion of `text` after the specified `delimiter`. An optional numeric `index` indicates which occurrence of the `delimiter` should be considered. An optional list `index` indicates which occurrence of the `delimiter` should be considered, as well as whether indexing should be done from the start or end of the input.

Example 1

Get the portion of "111-222-333" after the (first) hyphen.

Usage

Power Query M

```
Text.AfterDelimiter("111-222-333", "-")
```

Output

```
"222-333"
```

Example 2

Get the portion of "111-222-333" after the second hyphen.

Usage

Power Query M

```
Text.AfterDelimiter("111-222-333", "-", 1)
```

Output

"333"

Example 3

Get the portion of "111-222-333" after the second hyphen from the end.

Usage

Power Query M

```
Text.AfterDelimiter("111-222-333", "-", {1, RelativePosition.FromEnd})
```

Output

"222-333"

Text.At

06/19/2025

Syntax

```
Text.At(text as nullable text, index as number) as nullable text
```

About

Returns the character in the text value, `text` at position `index`. The first character in the text is at position 0.

Example 1

Find the character at position 4 in string "Hello, World".

Usage

Power Query M

```
Text.At("Hello, World", 4)
```

Output

```
"o"
```


Text.BeforeDelimiter

06/19/2025

Syntax

```
Text.BeforeDelimiter(text as nullable text, delimiter as text, optional index as any) as any
```

About

Returns the portion of `text` before the specified `delimiter`. An optional numeric `index` indicates which occurrence of the `delimiter` should be considered. An optional list `index` indicates which occurrence of the `delimiter` should be considered, as well as whether indexing should be done from the start or end of the input.

Example 1

Get the portion of "111-222-333" before the (first) hyphen.

Usage

Power Query M

```
Text.BeforeDelimiter("111-222-333", "-")
```

Output

```
"111"
```

Example 2

Get the portion of "111-222-333" before the second hyphen.

Usage

Power Query M

```
Text.BeforeDelimiter("111-222-333", "-", 1)
```

Output

```
"111-222"
```

Example 3

Get the portion of "111-222-333" before the second hyphen from the end.

Usage

```
Power Query M
```

```
Text.BeforeDelimiter("111-222-333", "-", {1, RelativePosition.FromEnd})
```

Output

```
"111"
```

Text.BetweenDelimiters

06/19/2025

Syntax

```
Text.BetweenDelimiters(text as nullable text, startDelimiter as text, endDelimiter as text, optional startIndex as any, optional endIndex as any) as any
```

About

Returns the portion of `text` between the specified `startDelimiter` and `endDelimiter`. An optional numeric `startIndex` indicates which occurrence of the `startDelimiter` should be considered. An optional list `startIndex` indicates which occurrence of the `startDelimiter` should be considered, as well as whether indexing should be done from the start or end of the input. The `endIndex` is similar, except that indexing is done relative to the `startIndex`.

Example 1

Get the portion of "111 (222) 333 (444)" between the (first) open parenthesis and the (first) closed parenthesis that follows it.

Usage

Power Query M

```
Text.BetweenDelimiters("111 (222) 333 (444)", "(", ")")
```

Output

```
"222"
```

Example 2

Get the portion of "111 (222) 333 (444)" between the second open parenthesis and the first closed parenthesis that follows it.

Usage

Power Query M

```
Text.BetweenDelimiters("111 (222) 333 (444)", "(", ")", 1, 0)
```

Output

```
"444"
```

Example 3

Get the portion of "111 (222) 333 (444)" between the second open parenthesis from the end and the second closed parenthesis that follows it.

Usage

Power Query M

```
Text.BetweenDelimiters("111 (222) 333 (444)", "(", ")", {1,  
RelativePosition.FromEnd}, {1, RelativePosition.FromStart})
```

Output

```
"222) 333 (444"
```

Text.Clean

06/19/2025

Syntax

```
Text.Clean(text as nullable text) as nullable text
```

About

Returns a text value with all control characters of `text` removed.

Example 1

Remove line feeds and other control characters from a text value.

Usage

Power Query M

```
Text.Clean("ABC#(lf)D")
```

Output

```
"ABCD"
```

Text.Combine

06/19/2025

Syntax

```
Text.Combine(texts as list, optional separator as nullable text) as text
```

About

Returns the result of combining the list of text values, `texts`, into a single text value. Any `null` values present in `texts` are ignored. An optional `separator` used in the final combined text can be specified.

Example 1

Combine text values "Seattle" and "WA".

Usage

Power Query M

```
Text.Combine({"Seattle", "WA"})
```

Output

```
"SeattleWA"
```

Example 2

Combine text values "Seattle" and "WA", separated by a comma and a space.

Usage

Power Query M

```
Text.Combine({"Seattle", "WA"}, ", ")
```

Output

```
"Seattle, WA"
```

Example 3

Combine the values "Seattle", `null`, and "WA", separated by a comma and a space. (Note that the `null` is ignored.)

Usage

Power Query M

```
Text.Combine({"Seattle", null, "WA"}, ", ")
```

Output

```
"Seattle, WA"
```

Example 4

Usage

Combine the first name, middle initial (if present), and last name into the individual's full name.

Power Query M

```
let
    Source = Table.FromRecords({
        [First Name = "Doug", Middle Initial = "J", Last Name = "Elis"],
        [First Name = "Anna", Middle Initial = "M", Last Name = "Jorayew"],
        [First Name = "Rada", Middle Initial = null, Last Name = "Mihaylova"]
    }),
    FullName = Table.AddColumn(Source, "Full Name", each Text.Combine({[First Name], [Middle Initial], [Last Name]}, " "))
in
    FullName
```

Output

Power Query M

```
Table.FromRecords({
    [First Name = "Doug", Middle Initial = "J", Last Name = "Elis", Full Name = "Doug J Elis"],
    [First Name = "Anna", Middle Initial = "M", Last Name = "Jorayew", Full Name = "Anna M Jorayew"],
    [First Name = "Rada", Middle Initial = null, Last Name = "Mihaylova", Full
```

```
Name = "Rada Mihaylova"]  
})
```


Text.Contains

06/19/2025

Syntax

```
Text.Contains(text as nullable text, substring as text, optional comparer as nullable function) as nullable logical
```

About

Detects whether `text` contains the value `substring`. Returns true if the value is found. This function doesn't support wildcards or regular expressions.

The optional argument `comparer` can be used to specify case-insensitive or culture and locale-aware comparisons. The following built-in comparers are available in the formula language:

- [Comparer.Ordinal](#): Used to perform a case-sensitive ordinal comparison
- [Comparer.OrdinalIgnoreCase](#): Used to perform a case-insensitive ordinal comparison
- [Comparer.FromCulture](#): Used to perform a culture-aware comparison

If the first argument is null, this function returns null.

All characters are treated literally. For example, "DR", " DR", "DR ", and " DR " aren't considered equal to each other.

Example 1

Find if the text "Hello World" contains "Hello".

Usage

```
Power Query M
```

```
Text.Contains("Hello World", "Hello")
```

Output

```
true
```

Example 2

Find if the text "Hello World" contains "hello".

Usage

Power Query M

```
Text.Contains("Hello World", "hello")
```

Output

```
false
```

Example 3

Find if the text "Hello World" contains "hello", using a case-insensitive comparer.

Usage

Power Query M

```
Text.Contains("Hello World", "hello", Comparer.OrdinalIgnoreCase)
```

Output

```
true
```

Example 4

Find the rows in a table that contain either "A-" or "7" in the account code.

Usage

Power Query M

```
let
    Source = #table(type table [Account Code = text, Posted Date = date, Sales = number],
        {
            {"US-2004", #date(2023,1,20), 580},
            {"CA-8843", #date(2023,7,18), 280},
            {"PA-1274", #date(2022,1,12), 90},
            {"PA-4323", #date(2023,4,14), 187},
            {"US-1200", #date(2022,12,14), 350},
```

```

        {"PTY-507", #date(2023,6,4), 110}
    }},
    #"Filtered rows" = Table.SelectRows(
        Source,
        each Text.Contains([Account Code], "A-") or
            Text.Contains([Account Code], "7"))
in
    #"Filtered rows"

```

Output

Power Query M

```

#table(type table [Account Code = text, Posted Date = date, Sales = number],
{
    {"CA-8843", #date(2023,7,18), 280},
    {"PA-1274", #date(2022,1,12), 90},
    {"PA-4323", #date(2023,4,14), 187},
    {"PTY-507", #date(2023,6,4), 110}
})

```

Related content

- [How culture affects text formatting](#)

Text.End

06/19/2025

Syntax

```
Text.End(text as nullable text, count as number) as nullable text
```

About

Returns a `text` value that is the last `count` characters of the `text` value `text`.

Example 1

Get the last 5 characters of the text "Hello, World".

Usage

Power Query M

```
Text.End("Hello, World", 5)
```

Output

```
"World"
```

Text.EndsWithWith

06/19/2025

Syntax

```
Text.EndsWith(text as nullable text, substring as text, optional comparer as nullable function) as nullable logical
```

About

Indicates whether the given text, `text`, ends with the specified value, `substring`. The indication is case sensitive.

`comparer` is a `Comparer` which is used to control the comparison. Comparers can be used to provide case-insensitive or culture and locale-aware comparisons.

The following built-in comparers are available in the formula language:

- [Comparer.Ordinal](#): Used to perform an exact ordinal comparison
- [Comparer.OrdinalIgnoreCase](#): Used to perform an exact ordinal case-insensitive comparison
- [Comparer.FromCulture](#): Used to perform a culture-aware comparison

Example 1

Check if "Hello, World" ends with "world".

Usage

Power Query M

```
Text.EndsWith("Hello, World", "world")
```

Output

false

Example 2

Check if "Hello, World" ends with "World".

Usage

Power Query M

```
Text.EndsWith("Hello, World", "World")
```

Output

```
true
```

Related content

- [How culture affects text formatting](#)

Text.Format

06/19/2025

Syntax

```
Text.Format(formatString as text, arguments as any, optional culture as nullable text) as text
```

About

Returns formatted text that is created by applying `arguments` from a list or record to a format string `formatString`. An optional `culture` may also be provided (for example, "en-US").

Example 1

Format a list of numbers.

Usage

Power Query M

```
Text.Format("#{0}, #{1}, and #{2}.", {17, 7, 22})
```

Output

```
"17, 7, and 22."
```

Example 2

Format different data types from a record according to United States English culture.

Usage

Power Query M

```
Text.Format(  
    "The time for the #[distance] km run held in #[city] on #[date] was #  
    [duration].",  
    [  
        city = "Seattle",  
        date = #date(2015, 3, 10),
```

```
        duration = #duration(0, 0, 54, 40),  
        distance = 10  
    ],  
    "en-US"  
)
```

Output

```
"The time for the 10 km run held in Seattle on 3/10/2015 was 00:54:40."
```

Related content

- [How culture affects text formatting](#)

Text.From

06/19/2025

Syntax

```
Text.From(value as any, optional culture as nullable text) as nullable text
```

About

Returns the text representation of a specified value.

- value**: The value to convert to text. The value can be a **number**, **date**, **time**, **datetime**, **datetimezone**, **logical**, **duration**, or **binary** value. If the given value is **null**, this function returns **null**.
- culture**: (Optional) The culture to use when converting the value to text (for example, "en-US").

Example 1

Create a text value from the number 3.

Usage

```
Power Query M
```

```
Text.From(3)
```

Output

```
"3"
```

Example 2

Get the text equivalent of the specified date and time.

Usage

```
Power Query M
```

```
Text.From(#datetime(2024, 6, 24, 14, 32, 22))
```

Output

```
"6/24/2024 2:32:22 PM"
```

Example 3

Get the German text equivalent of the specified date and time.

Usage

Power Query M

```
Text.From(#datetime(2024, 6, 24, 14, 32, 22), "de-DE")
```

Output

```
"24.06.2024 14:32:22"
```

Example 4

Get a binary value from text encoded as hexadecimal and change the value back to text.

Usage

Power Query M

```
Text.From(Binary.FromText("10FF", BinaryEncoding.Hex))
```

Output

```
"EP8="
```

Example 5

Get the rows in the table that contain data for France and convert the dates to text using the French culture.

Usage

Power Query M

```

let
    Source = #table(type table [Company ID = text, Country = text, Date = date],
        {
            {"JS-464", "USA", #date(2024, 3, 24)},
            {"LT-331", "France", #date(2024, 10, 5)},
            {"XE-100", "USA", #date(2024, 5, 21)},
            {"RT-430", "Germany", #date(2024, 1, 18)},
            {"LS-005", "France", #date(2023, 12, 31)},
            {"UW-220", "Germany", #date(2024, 2, 25)}
        }
    ),
    #"Convert Dates" = Table.TransformColumns(
        Table.SelectRows(Source, each [Country] = "France"),
        {"Date", each Text.From(_, "fr-FR")}
    )
in
    #"Convert Dates"

```

Output

Power Query M

```

#table(type table [Company ID = text, Country = text, Date = text],
{
    {"LT-331", "France", "05/10/2024"},
    {"LS-005", "France", "31/12/2023"}
})

```

Related content

- [How culture affects text formatting](#)
- [Standard numeric format strings](#)
- [Custom numeric format strings](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

Text.FromBinary

06/19/2025

Syntax

```
Text.FromBinary(binary as nullable binary, optional encoding as nullable number)  
as nullable text
```

About

Decodes data, `binary`, from a binary value in to a text value, using `encoding` type.

Text.InferNumberType

06/19/2025

Syntax

```
Text.InferNumberType(text as text, optional culture as nullable text) as type
```

About

Infers the granular number type (`Int64.Type`, `Double.Type`, and so on) of `text`. An error is raised if `text` is not a number. An optional `culture` may also be provided (for example, "en-US").

Related content

- [How culture affects text formatting](#)

Text.Insert

06/19/2025

Syntax

```
Text.Insert(text as nullable text, offset as number, newText as text) as nullable text
```

About

Returns the result of inserting text value `newText` into the text value `text` at position `offset`. Positions start at number 0.

Example 1

Insert "C" between "B" and "D" in "ABD".

Usage

Power Query M

```
Text.Insert("ABD", 2, "C")
```

Output

"ABCD"

Text.Length

06/19/2025

Syntax

```
Text.Length(text as nullable text) as nullable number
```

About

Returns the number of characters in the text `text`.

Example 1

Find how many characters are in the text "Hello World".

Usage

Power Query M

```
Text.Length("Hello World")
```

Output

11

Text.Lower

06/19/2025

Syntax

```
Text.Lower(text as nullable text, optional culture as nullable text) as nullable text
```

About

Returns the result of converting all characters in `text` to lowercase. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the lowercase version of "AbCd".

Usage

```
Power Query M
```

```
Text.Lower("AbCd")
```

Output

```
"abcd"
```

Related content

- [How culture affects text formatting](#)

Text.Middle

06/19/2025

Syntax

```
Text.Middle(text as nullable text, start as number, optional count as nullable number) as nullable text
```

About

Returns `count` characters, or through the end of `text`; at the offset `start`.

Example 1

Find the substring from the text "Hello World" starting at index 6 spanning 5 characters.

Usage

Power Query M

```
Text.Middle("Hello World", 6, 5)
```

Output

```
"World"
```

Example 2

Find the substring from the text "Hello World" starting at index 6 through the end.

Usage

Power Query M

```
Text.Middle("Hello World", 6, 20)
```

Output

```
"World"
```

Example 3

Find the substring from the text "Hello World" starting at index 0 spanning 2 characters.

Usage

Power Query M

```
Text.Middle("Hello World", 0, 2)
```

Output

```
"He"
```

Text.NewGuid

06/19/2025

Syntax

```
Text.NewGuid() as text
```

About

Returns a new, random globally unique identifier (GUID).

Text.PadEnd

06/19/2025

Syntax

```
Text.PadEnd(text as nullable text, count as number, optional character as nullable text) as nullable text
```

About

Returns a `text` value padded to length `count` by inserting spaces at the end of the text value `text`. An optional character `character` can be used to specify the character used for padding. The default pad character is a space.

Example 1

Pad the end of a text value so it is 10 characters long.

Usage

Power Query M

```
Text.PadEnd("Name", 10)
```

Output

```
"Name      "
```

Example 2

Pad the end of a text value with "|" so it is 10 characters long.

Usage

Power Query M

```
Text.PadEnd("Name", 10, "|")
```

Output

```
"Name|||||"
```

Text.PadStart

06/19/2025

Syntax

```
Text.PadStart(text as nullable text, count as number, optional character as nullable text) as nullable text
```

About

Returns a `text` value padded to length `count` by inserting spaces at the start of the text value `text`. An optional character `character` can be used to specify the character used for padding. The default pad character is a space.

Example 1

Pad the start of a text value so it is 10 characters long.

Usage

Power Query M

```
Text.PadStart("Name", 10)
```

Output

```
"      Name"
```

Example 2

Pad the start of a text value with "|" so it is 10 characters long.

Usage

Power Query M

```
Text.PadStart("Name", 10, "|")
```

Output

```
"|||||Name"
```

Text.PositionOf

06/19/2025

Syntax

```
Text.PositionOf(text as text, substring as text, optional occurrence as nullable number, optional comparer as nullable function) as any
```

About

Returns the position of the specified occurrence of the text value `substring` found in `text`. An optional parameter `occurrence` may be used to specify which occurrence position to return (first occurrence by default). Returns -1 if `substring` was not found.

`comparer` is a `Comparer` which is used to control the comparison. Comparers can be used to provide case-insensitive or culture and locale-aware comparisons.

The following built-in comparers are available in the formula language:

- `Comparer.Ordinal`: Used to perform an exact ordinal comparison
- `Comparer.OrdinalIgnoreCase`: Used to perform an exact ordinal case-insensitive comparison
- `Comparer.FromCulture`: Used to perform a culture-aware comparison

Example 1

Get the position of the first occurrence of "World" in the text "Hello, World! Hello, World!".

Usage

Power Query M

```
Text.PositionOf("Hello, World! Hello, World!", "World")
```

Output

7

Example 2

Get the position of last occurrence of "World" in "Hello, World! Hello, World!".

Usage

Power Query M

```
Text.PositionOf("Hello, World! Hello, World!", "World", Occurrence.Last)
```

Output

21

Related content

- [How culture affects text formatting](#)

Text.PositionOfAny

06/19/2025

Syntax

```
Text.PositionOfAny(text as text, characters as list, optional occurrence as nullable number) as any
```

About

Returns the first position of any character in the list `characters` that is found in `text`. An optional parameter `occurrence` may be used to specify which occurrence position to return.

Example 1

Find the first position of "W" or "H" in text "Hello, World!".

Usage

Power Query M

```
Text.PositionOfAny("Hello, World!", {"H", "W"})
```

Output

0

Example 2

Find all the positions of "W" or "H" in text "Hello, World!".

Usage

Power Query M

```
Text.PositionOfAny("Hello, World!", {"H", "W"}, Occurrence.All)
```

Output

$\{0, 7\}$

Text.Proper

06/19/2025

Syntax

```
Text.Proper(text as nullable text, optional culture as nullable text) as nullable text
```

About

Returns the result of capitalizing only the first letter of each word in text value `text`. All other letters are returned in lowercase. An optional `culture` may also be provided (for example, "en-US").

Example 1

Use `Text.Proper` on a simple sentence.

Usage

Power Query M

```
Text.Proper("the QUICK BrOWn fOx jUmPs oVER tHe LAzy DoG")
```

Output

```
"The Quick Brown Fox Jumps Over The Lazy Dog"
```

Related content

- [How culture affects text formatting](#)

Text.Range

06/19/2025

Syntax

```
Text.Range(text as nullable text, offset as number, optional count as nullable number) as nullable text
```

About

Returns the substring from the text `text` found at the offset `offset`. An optional parameter, `count`, can be included to specify how many characters to return. Throws an error if there aren't enough characters.

Example 1

Find the substring from the text "Hello World" starting at index 6.

Usage

```
Power Query M
```

```
Text.Range("Hello World", 6)
```

Output

```
"World"
```

Example 2

Find the substring from the text "Hello World Hello" starting at index 6 spanning 5 characters.

Usage

```
Power Query M
```

```
Text.Range("Hello World Hello", 6, 5)
```

Output

"World"

Text.Remove

06/19/2025

Syntax

```
Text.Remove(text as nullable text, removeChars as any) as nullable text
```

About

Returns a copy of the text value `text` with all the characters from `removeChars` removed.

Example 1

Remove characters , and ; from the text value.

Usage

Power Query M

```
Text.Remove("a,b;c", {"",",",";"})
```

Output

```
"abc"
```

Text.RemoveRange

06/19/2025

Syntax

```
Text.RemoveRange(text as nullable text, offset as number, optional count as nullable number) as nullable text
```

About

Returns a copy of the text value `text` with all the characters from position `offset` removed. An optional parameter, `count` can be used to specify the number of characters to remove. The default value of `count` is 1. Position values start at 0.

Example 1

Remove 1 character from the text value "ABEFC" at position 2.

Usage

Power Query M

```
Text.RemoveRange("ABEFC", 2)
```

Output

"ABFC"

Example 2

Remove two characters from the text value "ABEFC" starting at position 2.

Usage

Power Query M

```
Text.RemoveRange("ABEFC", 2, 2)
```

Output

"ABC"

Text.Repeat

06/19/2025

Syntax

```
Text.Repeat(text as nullable text, count as number) as nullable text
```

About

Returns a text value composed of the input text `text` repeated `count` times.

Example 1

Repeat the text "a" five times.

Usage

Power Query M

```
Text.Repeat("a", 5)
```

Output

```
"aaaaa"
```

Example 2

Repeat the text "helloworld" three times.

Usage

Power Query M

```
Text.Repeat("helloworld.", 3)
```

Output

```
"helloworld.helloworld.helloworld."
```

Text.Replace

06/19/2025

Syntax

```
Text.Replace(text as nullable text, old as text, new as text) as nullable text
```

About

Returns the result of replacing all occurrences of text value **old** in text value **text** with text value **new**. This function is case sensitive.

Example 1

Replace every occurrence of "the" in a sentence with "a".

Usage

Power Query M

```
Text.Replace("the quick brown fox jumps over the lazy dog", "the", "a")
```

Output

```
"a quick brown fox jumps over a lazy dog"
```

Text.ReplaceRange

06/19/2025

Syntax

```
Text.ReplaceRange(text as nullable text, offset as number, count as number,  
newText as text) as nullable text
```

About

Returns the result of removing a number of characters, `count`, from text value `text` beginning at position `offset` and then inserting the text value `newText` at the same position in `text`.

Example 1

Replace a single character at position 2 in text value "ABGF" with new text value "CDE".

Usage

Power Query M

```
Text.ReplaceRange("ABGF", 2, 1, "CDE")
```

Output

```
"ABCDEF"
```

Text.Reverse

06/19/2025

Syntax

```
Text.Reverse(text as nullable text) as nullable text
```

About

Reverses the provided `text`.

Example 1

Reverse the text "123".

Usage

Power Query M

```
Text.Reverse("123")
```

Output

```
"321"
```

Text.Select

06/19/2025

Syntax

```
Text.Select(text as nullable text, selectChars as any) as nullable text
```

About

Returns a copy of the text value `text` with all the characters not in `selectChars` removed.

Example 1

Select all characters in the range of 'a' to 'z' from the text value.

Usage

Power Query M

```
Text.Select("a,b;c", {"a".."z"})
```

Output

```
"abc"
```

Text.Split

06/19/2025

Syntax

```
Text.Split(text as text, separator as text) as list
```

About

Returns a list of text values resulting from the splitting of a text value based on the specified delimiter.

- **text**: The text value to split.
- **separator**: The delimiter used to split the text. The delimiter can be either a single character or a sequence of characters. If a sequence of characters is used, the text is split only at instances where the exact sequence occurs.

Example 1

Create a list from the "|" delimited text value "Name|Address|PhoneNumber".

Usage

Power Query M

```
Text.Split("Name|Address|PhoneNumber", "|")
```

Output

Power Query M

```
{  
    "Name",  
    "Address",  
    "PhoneNumber"  
}
```

Example 2

Create a list from the text value using a sequence of characters.

Usage

Power Query M

```
Text.Split("Name, the Customer, the Purchase Date", ", the ")
```

Output

Power Query M

```
{  
    Name,  
    Customer,  
    Purchase Date  
}
```


Text.SplitAny

06/19/2025

Syntax

```
Text.SplitAny(text as text, separators as text) as list
```

About

Returns a list of text values resulting from the splitting of a text value based on any character specified in the delimiter.

- **text**: The text value to split.
- **separator**: The delimiter characters used to split the text.

Example 1

Create a list from the given text using the specified delimiter characters.

Usage

Power Query M

```
Text.SplitAny("Name|Customer ID|Purchase|Month-Day-Year", "|-")
```

Output

Power Query M

```
{
    "Name",
    "Customer ID",
    "Purchase",
    "Month",
    "Day",
    "Year"
}
```

Text.Start

06/19/2025

Syntax

```
Text.Start(text as nullable text, count as number) as nullable text
```

About

Returns the first `count` characters of `text` as a text value.

Example 1

Get the first 5 characters of "Hello, World".

Usage

Power Query M

```
Text.Start("Hello, World", 5)
```

Output

```
"Hello"
```

Example 2

Use the first four characters of the first name and the first three characters of the last name to create an individual's email address.

Usage

Power Query M

```
let
    Source = #table(type table [First Name = text, Last Name = text],
        {
            {"Douglas", "Elis"},
            {"Ana", "Jorayew"},
            {"Rada", "Mihaylova"}
        })
```

```

        EmailAddress = Table.AddColumn(
            Source,
            "Email Address",
            each Text.Combine({
                Text.Start([First Name], 4),
                Text.Start([Last Name], 3),
                "@contoso.com"
            })
        )
    in
        EmailAddress

```

Output

Power Query M

```

#table(type table [First Name = text, Last Name = text, Email Address = text],
{
    {"Douglas", "Elis", "DougEli@contoso.com"},
    {"Ana", "Jorayew", "AnaJor@contoso.com"},
    {"Rada", "Mihaylova", "RadaMih@contoso.com"}
})

```

Text.StartsWith

06/19/2025

Syntax

```
Text.StartsWith(text as nullable text, substring as text, optional comparer as nullable function) as nullable logical
```

About

Returns true if text value `text` starts with text value `substring`.

- `text`: A `text` value which is to be searched.
- `substring`: A `text` value which is the substring to be searched for in `text`.
- `comparer`: *[Optional]* A `Comparer` used for controlling the comparison. For example, [Comparer.OrdinalIgnoreCase](#) may be used to perform case-insensitive searches.

`comparer` is a `Comparer` which is used to control the comparison. Comparers can be used to provide case-insensitive or culture and locale-aware comparisons.

The following built-in comparers are available in the formula language:

- [Comparer.Ordinal](#): Used to perform an exact ordinal comparison.
- `Comparer.OrdinalIgnoreCase`: Used to perform an exact ordinal case-insensitive comparison.
- [Comparer.FromCulture](#): Used to perform a culture-aware comparison.

Example 1

Check if the text "Hello, World" starts with the text "hello".

Usage

Power Query M

```
Text.StartsWith("Hello, World", "hello")
```

Output

false

Example 2

Check if the text "Hello, World" starts with the text "Hello".

Usage

Power Query M

```
Text.StartsWith("Hello, World", "Hello")
```

Output

```
true
```

Example 3

Ignoring case, check if the text "Hello, World" starts with the text "hello".

Usage

Power Query M

```
Text.StartsWith("Hello, World", "hello", Comparer.OrdinalIgnoreCase)
```

Output

```
true
```

Related content

- [How culture affects text formatting](#)

Text.ToBinary

06/19/2025

Syntax

```
Text.ToBinary(text as nullable text, optional encoding as nullable number,  
optional includeByteOrderMark as nullable logical) as nullable binary
```

About

Encodes the given text value, `text`, into a binary value using the specified `encoding`.

Text.ToList

06/19/2025

Syntax

```
Text.ToList(text as text) as list
```

About

Returns a list of character values from the given text value `text`.

Example 1

Create a list of character values from the text "Hello World".

Usage

Power Query M

```
Text.ToList("Hello World")
```

Output

powerquery-m{

```
"H",  
"e",  
"l",  
"l",  
"o",  
" ",  
"W",  
"o",  
"r",  
"l",  
"d"
```

}

Text.Trim

06/19/2025

Syntax

```
Text.Trim(text as nullable text, optional trim as any) as nullable text
```

About

Returns the result of removing all leading and trailing characters from the specified `text`. By default, all the leading and trailing whitespace characters are removed.

- `text`: The text from which the leading and trailing characters are to be removed.
- `trim`: Overrides the whitespace characters that are trimmed by default. This parameter can either be a single character or a list of single characters. Each leading and trailing trim operation stops when a non-trimmed character is encountered.

Example 1

Remove leading and trailing whitespace from " a b c d ".

Usage

Power Query M

```
Text.Trim("    a b c d    ")
```

Output

```
"a b c d"
```

Example 2

Remove leading and trailing zeroes from the text representation of a number.

Usage

Power Query M


```
Text.Trim("0000056.4200", "0")
```

Output

```
"56.42"
```

Example 3

Remove the leading and trailing brackets from an HTML tag.

Usage

Power Query M

```
Text.Trim("<div/>", {"<", ">", "/"})
```

Output

```
"div"
```

Example 4

Remove the special characters used around the pending sales status.

Usage

Power Query M

```
let
    Source = #table(type table [Home Sale = text, Sales Date = date, Sales Status = text],
        {
            {"1620 Ferris Way", #date(2024, 8, 22), "###@@Pending@@###"},
            {"757 1st Ave. S.", #date(2024, 3, 15), "Sold"},
            {"22303 Fillmore", #date(2024, 10, 2), "###@@Pending@@###"}
        }
    ),
    #"Trimmed Status" = Table.TransformColumns(Source, {"Sales Status", each Text.Trim(_, {"#", "@"})})
in
    #"Trimmed Status"
```

Output

Power Query M

```
#table(type table [Home Sale = text, Sales Date = date, Sales Status = text],
{
  {"1620 Ferris Way", #date(2024, 8, 22), "Pending"},
  {"757 1st Ave. S.", #date(2024, 3, 15), "Sold"},
  {"22303 Fillmore", #date(2024, 10, 2), "Pending"}
})
```

Text.TrimEnd

06/19/2025

Syntax

```
Text.TrimEnd(text as nullable text, optional trim as any) as nullable text
```

About

Returns the result of removing all trailing characters from the specified `text`. By default, all the trailing whitespace characters are removed.

- `text`: The text from which the trailing characters are to be removed.
- `trim`: Overrides the whitespace characters that are trimmed by default. This parameter can either be a single character or a list of single characters. Each trailing trim operation stops when a non-trimmed character is encountered.

Example 1

Remove trailing whitespace from " a b c d ".

Usage

Power Query M

```
Text.TrimEnd(" a b c d ")
```

Output

```
" a b c d"
```

Example 2

Remove trailing zeroes from a text representation of a padded floating point number.

Usage

Power Query M

```
Text.TrimEnd("03.487700000", "0")
```

Output

```
"03.4877"
```

Example 3

Remove the trailing padding characters from a fixed-width account name.

Usage

Power Query M

```
let
    Source = #table(type table [Name = text, Account Name= text, Interest =
number],
        {
            {"Bob", "US-847263****@", 2.8410},
            {"Leslie", "FR-4648****@**", 3.8392},
            {"Ringo", "DE-2046790@****", 12.6600}
        }
    ),
    #"Trimmed Account" = Table.TransformColumns(Source, {"Account Name", each
Text.TrimEnd(_, {"*", "@"})})
in
    #"Trimmed Account"
```

Output

Power Query M

```
#table(type table [Name = text, Account Name = text, Interest = number],
    {
        {"Bob", "US-847263", 2.841},
        {"Leslie", "FR-4648", 3.8392},
        {"Ringo", "DE-2046790", 12.66}
    }
),
```

Text.TrimStart

06/19/2025

Syntax

```
Text.TrimStart(text as nullable text, optional trim as any) as nullable text
```

About

Returns the result of removing all leading characters from the specified `text`. By default, all the leading whitespace characters are removed.

- `text`: The text from which the leading characters are to be removed.
- `trim`: Overrides the whitespace characters that are trimmed by default. This parameter can either be a single character or a list of single characters. Each leading trim operation stops when a non-trimmed character is encountered.

Example 1

Remove leading whitespace from " a b c d ".

Usage**

Power Query M

```
Text.TrimStart("  a b c d  ")
```

Output

```
"a b c d  "
```

Example 2

Remove leading zeroes from the text representation of a number.

Usage

Power Query M

```
Text.TrimStart("0000056.420", "0")
```

Output

```
"56.420"
```

Example 3

Remove the leading padding characters from a fixed width account name.

Usage

Power Query M

```
let
    Source = #table(type table [Name = text, Account Name= text, Interest =
number],
        {
            {"Bob", "@****847263-US", 2.8410},
            {"Leslie", "@*****4648-FR", 3.8392},
            {"Ringo", "@*****24679-DE", 12.6600}
        }
    ),
    #"Trimmed Account" = Table.TransformColumns(Source, {"Account Name", each
Text.TrimStart(_, {"*", "@"})})
in
    #"Trimmed Account"
```

Output

Power Query M

```
#table(type table [Name = text, Account Name = text, Interest = number],
    {
        {"Bob", "847263-US", 2.841},
        {"Leslie", "4648-FR", 3.8392},
        {"Ringo", "2046790-DE", 12.66}
    }
),
```

Text.Upper

06/19/2025

Syntax

```
Text.Upper(text as nullable text, optional culture as nullable text) as nullable text
```

About

Returns the result of converting all characters in `text` to uppercase. An optional `culture` may also be provided (for example, "en-US").

Example 1

Get the uppercase version of "aBcD".

Usage

Power Query M

```
Text.Upper("aBcD")
```

Output

"ABCD"

Related content

- [How culture affects text formatting](#)

Time functions

Article • 11/15/2022

These functions create and manipulate time values.

Name	Description
Time.EndOfHour	Returns the end of the hour.
Time.From	Returns a time value from a value.
Time.FromText	Creates a Time from local, universal, and custom Time formats.
Time.Hour	Returns an hour value from a DateTime value.
Time.Minute	Returns a minute value from a DateTime value.
Time.Second	Returns a second value from a DateTime value
Time.StartOfHour	Returns the start of the hour.
Time.ToRecord	Returns a record containing parts of a Date value.
Time.ToText	Returns a text value from a Time value.
#time	Creates a time value from hour, minute, and second.

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

Time.EndOfHour

06/19/2025

Syntax

```
Time.EndOfHour(dateTime as any) as any
```

About

Returns the end of the hour represented by `dateTime`, including fractional seconds. Time zone information is preserved.

- `dateTime`: A `time`, `datetime`, or `datetimezone` value from which the end of the hour is calculated.

Example 1

Get the end of the hour for 5/14/2011 05:00:00 PM.

Usage

Power Query M

```
Time.EndOfHour(#datetime(2011, 5, 14, 17, 0, 0))
```

Output

```
#datetime(2011, 5, 14, 17, 59, 59.9999999)
```

Example 2

Get the end of the hour for 5/17/2011 05:00:00 PM -7:00.

Usage

Power Query M

```
Time.EndOfHour(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

Output

```
#datetimezone(2011, 5, 17, 5, 59, 59.9999999, -7, 0)
```

Time.From

06/19/2025

Syntax

```
Time.From(value as any, optional culture as nullable text) as nullable time
```

About

Returns a `time` value from the given `value`. An optional `culture` may also be provided (for example, "en-US"). If the given `value` is `null`, **Time.From** returns `null`. If the given `value` is `time`, `value` is returned. Values of the following types can be converted to a `time` value:

- `text`: A `time` value from textual representation. Refer to [Time.FromText](#) for details.
- `datetime`: The time component of the `value`.
- `datetimezone`: The time component of the local datetime equivalent of `value`.
- `number`: A `time` equivalent to the number of fractional days expressed by `value`. If `value` is negative or greater or equal to 1, an error is returned.

If `value` is of any other type, an error is returned.

Example 1

Convert `0.7575` to a `time` value.

Usage

```
Power Query M
```

```
Time.From(0.7575)
```

Output

```
#time(18, 10, 48)
```

Example 2

Convert `#datetime(1899, 12, 30, 06, 45, 12)` to a `time` value.

Usage

Power Query M

```
Time.From(#datetime(1899, 12, 30, 06, 45, 12))
```

Output

```
#time(06, 45, 12)
```

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

Time.FromText

06/19/2025

Syntax

```
Time.FromText(text as nullable text, optional options as any) as nullable time
```

About

Creates a `time` value from a textual representation, `text`. An optional `record` parameter, `options`, may be provided to specify additional properties. The `record` can contain the following fields:

- **Format**: A `text` value indicating the format to use. For more details, go to [Standard date and time format strings](#) and [Custom date and time format strings](#). Omitting this field or providing `null` will result in parsing the time using a best effort.
- **Culture**: When **Format** is not null, **Culture** controls some format specifiers. For example, in "en-US" `"tt"` is "AM" or "PM", while in "ar-EG" `"tt"` is "ص" or "م". When **Format** is null, **Culture** controls the default format to use. When **Culture** is null or omitted, [Culture.Current](#) is used.

To support legacy workflows, `options` may also be a text value. This has the same behavior as if `options = [Format = null, Culture = options]`.

Example 1

Convert `"10:12:31am"` into a Time value.

Usage

Power Query M

```
Time.FromText("10:12:31am")
```

Output

```
#time(10, 12, 31)
```

Example 2

Convert "1012" into a Time value.

Usage

Power Query M

```
Time.FromText("1012")
```

Output

```
#time(10, 12, 00)
```

Example 3

Convert "10" into a Time value.

Usage

Power Query M

```
Time.FromText("10")
```

Output

```
#time(10, 00, 00)
```

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

Time.Hour

06/19/2025

Syntax

```
Time.Hour(dateTime as any) as nullable number
```

About

Returns the hour component of the provided `time`, `datetime`, or `datetimezone` value, `dateTime`.

Example 1

Find the hour in `#datetime(2011, 12, 31, 9, 15, 36)`.

Usage

Power Query M

```
Time.Hour(#datetime(2011, 12, 31, 9, 15, 36))
```

Output

9

Time.Minute

06/19/2025

Syntax

```
Time.Minute(dateTime as any) as nullable number
```

About

Returns the minute component of the provided `time`, `datetime`, or `datetimezone` value, `dateTime`.

Example 1

Find the minute in `#datetime(2011, 12, 31, 9, 15, 36)`.

Usage

Power Query M

```
Time.Minute(#datetime(2011, 12, 31, 9, 15, 36))
```

Output

15

Time.Second

06/19/2025

Syntax

```
Time.Second(dateTime as any) as nullable number
```

About

Returns the second component of the provided `time`, `datetime`, or `datetimezone` value, `dateTime`.

Example 1

Find the second value from a datetime value.

Usage

```
Power Query M
```

```
Time.Second(#datetime(2011, 12, 31, 9, 15, 36.5))
```

Output

```
36.5
```

Time.StartOfHour

06/19/2025

Syntax

```
Time.StartOfHour(dateTime as any) as any
```

About

Returns the start of the hour represented by `dateTime`. `dateTime` must be a `time`, `datetime` or `datetimezone` value.

Example 1

Find the start of the hour for October 10th, 2011, 8:10:32AM.

Usage

Power Query M

```
Time.StartOfHour(#datetime(2011, 10, 10, 8, 10, 32))
```

Output

```
#datetime(2011, 10, 10, 8, 0, 0)
```

Time.ToRecord

06/19/2025

Syntax

```
Time.ToRecord(time as time) as record
```

About

Returns a record containing the parts of the given Time value, **time**.

- **time**: A **time** value for from which the record of its parts is to be calculated.

Example 1

Convert the **#time(11, 56, 2)** value into a record containing Time values.

Usage

Power Query M

```
Time.ToRecord(#time(11, 56, 2))
```

Output

Power Query M

```
[
    Hour = 11,
    Minute = 56,
    Second = 2
]
```

Time.ToText

06/19/2025

Syntax

```
Time.ToText(time as nullable time, optional options as any, optional culture as nullable text) as nullable text
```

About

Returns a textual representation of `time`. An optional `record` parameter, `options`, may be provided to specify additional properties. `culture` is only used for legacy workflows. The `record` can contain the following fields:

- Format**: A `text` value indicating the format to use. For more details, go to [Standard date and time format strings](#) and [Custom date and time format strings](#). Omitting this field or providing `null` will result in formatting the date using the default defined by `Culture`.
- Culture**: When `Format` is not null, `Culture` controls some format specifiers. For example, in "en-US" `"tt"` is "AM" or "PM", while in "ar-EG" `"tt"` is "ص" or "م". When `Format` is `null`, `Culture` controls the default format to use. When `Culture` is `null` or omitted, `Culture.Current` is used.

To support legacy workflows, `options` and `culture` may also be text values. This has the same behavior as if `options = [Format = options, Culture = culture]`.

Example 1

Convert `#time(01, 30, 25)` into a `text` value. *Result output may vary depending on current culture.*

Usage

Power Query M

```
Time.ToText(#time(11, 56, 2))
```

Output

"11:56 AM"

Example 2

Convert using a custom format and the German culture.

Usage

Power Query M

```
Time.ToText(#time(11, 56, 2), [Format="hh:mm", Culture="de-DE"])
```

Output

"11:56"

Example 3

Convert using standard time format.

Usage

Power Query M

```
Time.ToText(#time(11, 56, 2), [Format="T", Culture="de-DE"])
```

Output

"11:56:02"

Related content

- [How culture affects text formatting](#)
- [Standard date and time format strings](#)
- [Custom date and time format strings](#)

#time

06/19/2025

Syntax

```
#time(hour as number, minute as number, second as number) as time
```

About

Creates a time value from numbers representing the hour, minute, and (fractional) second.

Raises an error if these conditions are not true:

- $0 \leq \text{hour} \leq 24$
- $0 \leq \text{minute} \leq 59$
- $0 \leq \text{second} < 60$
- if hour is 24, then minute and second must be 0

Type functions

Article • 01/10/2025

These functions create and manipulate type values.

 Expand table

Name	Description
Type.AddTableKey	Add a key to a table type.
Type.ClosedRecord	The given type must be a record type returns a closed version of the given record type (or the same type, if it is already closed)
Type.Facets	Returns the facets of a type.
Type.ForFunction	Returns a type that represents functions with specific parameter and return type constraints.
Type.ForRecord	Returns a Record type from a fields record.
Type.FunctionParameters	Returns a record with field values set to the name of the parameters of a function type, and their values set to their corresponding types.
Type.FunctionRequiredParameters	Returns a number indicating the minimum number of parameters required to invoke the a type of function.
Type.FunctionReturn	Returns a type returned by a function type.
Type.Is	Determines if a value of the first type is always compatible with the second type.
Type.IsNullable	Returns true if a type is a nullable type; otherwise, false.
Type.IsOpenRecord	Returns whether a record type is open.
Type.ListItem	Returns an item type from a list type.
Type.NonNullable	Returns the non nullable type from a type.
Type.OpenRecord	Returns an opened version of a record type, or the same type, if it is already open.
Type.RecordFields	Returns a record describing the fields of a record type with each field of the returned record type having a corresponding name and a value that is a record of the form <code>[Type = type, Optional = logical]</code> .

Name	Description
Type.ReplaceFacets	Replaces the facets of a type.
Type.ReplaceTableKeys	Replaces the keys in a table type.
Type.TableColumn	Returns the type of a column in a table.
Type.TableKeys	Returns keys from a table type.
Type.TableRow	Returns a row type from a table type.
Type.TableSchema	Returns a table containing a description of the columns (i.e. the schema) of the specified table type.
Type.Union	Returns the union of a list of types.

Feedback

Was this page helpful?



[Provide product feedback](#) | [Ask the community](#) [↗](#)

Type.AddTableKey

06/19/2025

Syntax

```
Type.AddTableKey(table as type, columns as list, isPrimary as logical) as type
```

About

Adds a key to the given table type.

Type.ClosedRecord

06/19/2025

Syntax

```
Type.ClosedRecord(type as type) as type
```

About

Returns a closed version of the given `record type` (or the same type, if it is already closed).

Example 1

Create a closed version of `type [ A = number,... ]`.

Usage

Power Query M

```
Type.ClosedRecord(type [A = number, ...])
```

Output

```
type [A = number]
```

Related content

- [Types and type conversion](#)

Type.Facets

06/19/2025

Syntax

```
Type.Facets(type as type) as record
```

About

Returns a record containing the facets of `type`.

Type.ForFunction

06/19/2025

Syntax

```
Type.ForFunction(signature as record, min as number) as type
```

About

Creates a `function type` from `signature`, a record of `ReturnType` and `Parameters`, and `min`, the minimum number of arguments required to invoke the function.

Example 1

Creates the type for a function that takes a number parameter named X and returns a number.

Usage

Power Query M

```
Type.ForFunction([ReturnType = type number, Parameters = [X = type number]], 1)
```

Output

```
type function (X as number) as number
```

Type.ForRecord

06/19/2025

Syntax

```
Type.ForRecord(fields as record, open as logical) as type
```

About

Returns a type that represents records with specific type constraints on fields.

Example 1

Dynamically generate a table type.

Usage

Power Query M

```
let
    columnNames = {"Name", "Score"},
    columnTypes = {type text, type number},
    rowColumnTypes = List.Transform(columnTypes, (t) => [Type = t, Optional =
false]),
    rowType = Type.ForRecord(Record.FromList(rowColumnTypes, columnNames), false)
in
    #table(type table rowType, {"Betty", 90.3}, {"Carl", 89.5}))
```

Output

Power Query M

```
#table(
    type table [Name = text, Score = number],
    {"Betty", 90.3}, {"Carl", 89.5}
)
```

Type.FunctionParameters

06/19/2025

Syntax

```
Type.FunctionParameters(type as type) as record
```

About

Returns a record with field values set to the name of the parameters of `type`, and their values set to their corresponding types.

Example 1

Find the types of the parameters to the function `(x as number, y as text)`.

Usage

Power Query M

```
Type.FunctionParameters(type function (x as number, y as text) as any)
```

Output

```
[x = type number, y = type text]
```

Type.FunctionRequiredParameters

06/19/2025

Syntax

```
Type.FunctionRequiredParameters(type as type) as number
```

About

Returns a number indicating the minimum number of parameters required to invoke the input `type` of function.

Example 1

Find the number of required parameters to the function `(x as number, optional y as text)`.

Usage

Power Query M

```
Type.FunctionRequiredParameters(type function (x as number, optional y as text) as any)
```

Output

1

Type.FunctionReturn

06/19/2025

Syntax

```
Type.FunctionReturn(type as type) as type
```

About

Returns a type returned by a function `type`.

Example 1

Find the return type of `() as any`.

Usage

Power Query M

```
Type.FunctionReturn(type function () as any)
```

Output

```
type any
```


Type.Is

06/19/2025

Syntax

```
Type.Is(type1 as type, type2 as type) as logical
```

About

Determines if a value of `type1` is always compatible with `type2`. Parameter `type2` should be a primitive (or nullable primitive) type value. Otherwise, this function's behavior is undefined and shouldn't be relied on.

Example 1

Determine if a value of type number can always also be treated as type any.

Usage

```
Power Query M
```

```
Type.Is(type number, type any)
```

Output

```
true
```

Example 2

Determine if a value of type any can always also be treated as type number.

Usage

```
Power Query M
```

```
Type.Is(type any, type number)
```

Output

false

Related content

- [Types and type conversion](#)

Type.IsNullable

06/19/2025

Syntax

```
Type.IsNullable(type as type) as logical
```

About

Returns `true` if a type is a `nullable` type; otherwise, `false`.

Example 1

Determine if `number` is nullable.

Usage

Power Query M

```
Type.IsNullable(type number)
```

Output

```
false
```

Example 2

Determine if `type nullable number` is nullable.

Usage

Power Query M

```
Type.IsNullable(type nullable number)
```

Output

```
true
```

Related content

- [Types and type conversion](#)

Type.IsOpenRecord

06/19/2025

Syntax

```
Type.IsOpenRecord(type as type) as logical
```

About

Returns a `logical` indicating whether a record `type` is open.

Example 1

Determine if the record `type [ A = number, ... ]` is open.

Usage

Power Query M

```
Type.IsOpenRecord(type [A = number, ...])
```

Output

```
true
```

Type.ListItem

06/19/2025

Syntax

```
Type.ListItem(type as type) as type
```

About

Returns an item type from a list `type`.

Example 1

Find item type from the list `{number}`.

Usage

Power Query M

```
Type.ListItem(type {number})
```

Output

```
type number
```

Related content

- [Types and type conversion](#)

Type.NonNullable

06/19/2025

Syntax

```
Type.NonNullable(type as type) as type
```

About

Returns the non `nullable` type from the `type`.

Example 1

Return the non nullable type of `type nullable number`.

Usage

Power Query M

```
Type.NonNullable(type nullable number)
```

Output

```
type number
```

Related content

- [Types and type conversion](#)

Type.OpenRecord

06/19/2025

Syntax

```
Type.OpenRecord(type as type) as type
```

About

Returns an opened version of the given `record` `type` (or the same type, if it is already opened).

Example 1

Create an opened version of `type [ A = number ]`.

Usage

Power Query M

```
Type.OpenRecord(type [A = number])
```

Output

```
type [A = number, ...]
```

Related content

- [Types and type conversion](#)

Type.RecordFields

06/19/2025

Syntax

```
Type.RecordFields(type as type) as record
```

About

Returns a record describing the fields of a record `type`. Each field of the returned record type has a corresponding name and a value, in the form of a record `[ Type = type, Optional = logical ]`.

Example 1

Find the name and value of the record `[ A = number, optional B = any ]`.

Usage

Power Query M

```
Type.RecordFields(type [A = number, optional B = any])
```

Output

Power Query M

```
[  
  A = [Type = type number, Optional = false],  
  B = [Type = type any, Optional = true]  
]
```

Related content

- [Types and type conversion](#)

Type.ReplaceFacets

06/19/2025

Syntax

```
Type.ReplaceFacets(type as type, facets as record) as type
```

About

Replaces the facets of `type` with the facets contained in the record `facets`.

Type.ReplaceTableKeys

06/19/2025

Syntax

```
Type.ReplaceTableKeys(tableType as type, keys as list) as type
```

About

Returns a new table type with all keys replaced by the specified list of keys.

Each key is defined using a record in the following form:

- **Columns**: a list of the column names that define the key
- **Primary**: `true` if the key is the table's primary key; otherwise, `false`

The specified list of keys is validated to ensure that no more than one primary key is defined and that all key column names exist on the table type.

Example 1

Replace the key information on a table type.

Usage

Power Query M

```
let
    BaseType = type table [ID = number, FirstName = text, LastName = text],
    KeysAdded = Type.ReplaceTableKeys(
        BaseType,
        {
            [Columns = {"ID"}, Primary = true],
            [Columns = {"FirstName", "LastName"}, Primary = false]
        }
    ),
    DetailsOfKeys = Type.TableKeys(KeysAdded)
in
    DetailsOfKeys
```

Output

Power Query M

```
{  
    [Columns = {"ID"}, Primary = true],  
    [Columns = {"FirstName", "LastName"}, Primary = false]  
}
```

Example 2

Clear the key information previously defined on a table type.

Usage

Power Query M

```
let  
    TypeWithKey = Type.AddTableKey(type table [ID = number, Name = text], {"ID"},  
true),  
    KeyRemoved = Type.ReplaceTableKeys(TypeWithKey, {}),  
    DetailsOfKeys = Type.TableKeys(KeyRemoved)  
in  
    DetailsOfKeys
```

Output

Power Query M

```
{}
```

Type.TableColumn

06/19/2025

Syntax

```
Type.TableColumn(tableType as type, column as text) as type
```

About

Returns the type of the column `column` in the table type `tableType`.

Type.TableKeys

06/19/2025

Syntax

```
Type.TableKeys(tableType as type) as list
```

About

Returns the possibly empty list of keys for the given table type.

Each key is described by a record in the following form:

- **Columns**: a list of the column names that define the key
- **Primary**: **true** if the key is the table's primary key; otherwise, **false**

Example 1

Return the key information for a table type.

Usage

Power Query M

```
let
    BaseType = type table [ID = number, Name = text],
    AddKey = Type.AddTableKey(BaseType, {"ID"}, true),
    DetailsOfKeys = Type.TableKeys(AddKey)
in
    DetailsOfKeys
```

Output

Power Query M

```
{[Columns = {"ID"}, Primary = true]}
```

Type.TableRow

06/19/2025

Syntax

```
Type.TableRow(table as type) as type
```

About

Returns the row type of the specified table type. The result will always be a record type.

Example 1

Return the row type information for a simple table.

Usage

Power Query M

```
let  
    tableRowType = Type.TableRow(Value.Type(#table({"Column1"}, {})))  
in  
    Type.RecordFields(tableRowType)
```

Output

```
[Column1 = [Type = type any, Optional = false]]
```

Related content

- [Types and type conversion](#)

Type.TableSchema

06/19/2025

Syntax

```
Type.TableSchema(tableType as type) as table
```

About

Returns a table describing the columns of `tableType`.

Refer to the documentation for [Table.Schema](#) for a description of the resulting table.

Related content

- [Types and type conversion](#)

Type.Union

06/19/2025

Syntax

```
Type.Union(types as list) as type
```

About

Returns the union of the types in `types`.

Uri functions

Article • 08/04/2022

These functions create and manipulate URI query strings.

Name	Description
Uri.BuildQueryString	Assemble a record into a URI query string.
Uri.Combine	Returns a Uri based on the combination of the base and relative parts.
Uri.EscapeDataString	Encodes special characters in accordance with RFC 3986.
Uri.Parts	Returns a record value with the fields set to the parts of a Uri text value.

Feedback

Was this page helpful?

 Yes

 No

Get help at [Microsoft Q&A](#)

Uri.BuildQueryString

06/19/2025

Syntax

```
Uri.BuildQueryString(query as record) as text
```

About

Assemble the record `query` into a URI query string, escaping characters as necessary.

Example 1

Encode a query string which contains some special characters.

Usage

Power Query M

```
Uri.BuildQueryString([a = "1", b = "+$"])
```

Output

```
"a=1&b=%2B%24"
```

Uri.Combine

06/19/2025

Syntax

```
Uri.Combine(baseUri as text, relativeUri as text) as text
```

About

Returns an absolute URI that is the combination of the input `baseUri` and `relativeUri`.

Uri.EscapeDataString

06/19/2025

Syntax

```
Uri.EscapeDataString(data as text) as text
```

About

Encodes special characters in the input `data` according to the rules of RFC 3986.

Example 1

Encode the special characters in "+money\$".

Usage

Power Query M

```
Uri.EscapeDataString("+money$")
```

Output

```
"%2Bmoney%24"
```

Uri.Parts

06/19/2025

Syntax

```
Uri.Parts(absoluteUri as text) as record
```

About

Returns the parts of the input `absoluteUri` as a record, containing values such as Scheme, Host, Port, Path, Query, Fragment, UserName and Password.

Example 1

Find the parts of the absolute URI "www.adventure-works.com".

Usage

Power Query M

```
Uri.Parts("www.adventure-works.com")
```

Output

Power Query M

```
[
    Scheme = "http",
    Host = "www.adventure-works.com",
    Port = 80,
    Path = "/",
    Query = [],
    Fragment = "",
    UserName = "",
    Password = ""
]
```

Example 2

Decode a percent-encoded string.

Usage

Power Query M

```
let
    UriUnescapeDataString = (data as text) as text => Uri.Parts("http://contoso?
a=" & data)[Query][a]
in
    UriUnescapeDataString("%2Bmoney%24")
```

Output

```
" +money$"
```

Value functions

Article • 04/24/2024

These functions evaluate and perform operations on values.

 Expand table

Name	Description
Value.Alternates	Expresses alternate query plans.
Value.Compare	Returns -1, 0, or 1 based on whether the first value is less than, equal to, or greater than the second.
Value.Equals	Returns whether two values are equal.
Value.Expression	Returns an abstract syntax tree (AST) that represents the value's expression.
Value.VersionIdentity	Returns the version identity of a value.
Value.Versions	Returns a navigation table containing the available versions of a value.
Value.NativeQuery	Evaluates a query against a target.
Value.NullableEquals	Returns a logical value or null based on two values.
Value.Optimize	If value represents a query that can be optimized, returns the optimized query. Otherwise returns value.
Value.Type	Returns the type of the given value.

Arithmetic operations

 Expand table

Name	Description
Value.Add	Returns the sum of the two values.
Value.Divide	Returns the result of dividing the first value by the second.
Value.Multiply	Returns the product of the two values.
Value.Subtract	Returns the difference of the two values.

Parameter types

[Expand table](#)

Name	Description
Value.As	Returns the value if it is compatible with the specified type.
Value.Is	Determines whether a value is compatible with the specified type.
Value.ReplaceType	Replaces the value's type.

[Expand table](#)

Implementation	Description
Action.WithErrorContext	This function is intended for internal use only.
DirectQueryCapabilities.From	This function is intended for internal use only.
Embedded.Value	Accesses a value by name in an embedded mashup.
Excel.ShapeTable	This function is intended for internal use only.
Module.Versions	Returns a record of module versions for the current module and its dependencies.
Progress.DataSourceProgress	This function is intended for internal use only.
SqlExpression.SchemaFrom	This function is intended for internal use only.
SqlExpression.ToExpression	This function is intended for internal use only.
Value.Firewall	This function is intended for internal use only.
Value.ViewError	This function is intended for internal use only.
Value.ViewFunction	This function is intended for internal use only.
Variable.Value	This function is intended for internal use only.

Metadata

[Expand table](#)

Name	Description
Value.Metadata	Returns a record containing the input's metadata.

Name	Description
Value.RemoveMetadata	Removes the metadata on the value and returns the original value.
Value.ReplaceMetadata	Replaces the metadata on a value with the new metadata record provided and returns the original value with the new metadata attached.

Lineage

 Expand table

Name	Description
Graph.Nodes	This function is intended for internal use only.
Value.Lineage	This function is intended for internal use only.
Value.Traits	This function is intended for internal use only.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#) 

Action.WithErrorContext

06/19/2025

Syntax

```
Action.WithErrorContext(action as action, context as text) as action
```

About

This function is intended for internal use only.

DirectQueryCapabilities.From

06/19/2025

Syntax

```
DirectQueryCapabilities.From(value as any) as table
```

About

This function is intended for internal use only.

Embedded.Value

06/19/2025

Syntax

```
Embedded.Value(value as any, path as text) as any
```

About

Accesses a value by name in an embedded mashup.

Excel.ShapeTable

06/19/2025

Syntax

```
Excel.ShapeTable(table as table, optional options as nullable record) as any
```

About

This function is intended for internal use only.

Graph.Nodes

06/19/2025

Syntax

```
Graph.Nodes(graph as record) as list
```

About

This function is intended for internal use only.

Module.Versions

06/19/2025

Syntax

```
Module.Versions() as record
```

About

Returns a record of module versions for the current module and its dependencies.

Progress.DataSourceProgress

06/19/2025

Syntax

```
Progress.DataSourceProgress() as any
```

About

This function is intended for internal use only.

SqlExpression.SchemaFrom

06/19/2025

Syntax

```
SqlExpression.SchemaFrom(schema as any) as any
```

About

This function is intended for internal use only.

SqlExpression.ToExpression

06/19/2025

Syntax

```
SqlExpression.ToExpression(sql as text, environment as record) as text
```

About

Converts the provided `sql` query to M code, with the available identifiers defined by `environment`. This function is intended for internal use only.

Value.Add

06/19/2025

Syntax

```
Value.Add(value1 as any, value2 as any, optional precision as nullable number) as  
any
```

About

Returns the sum of `value1` and `value2`. An optional `precision` parameter may be specified, by default `Precision.Double` is used.

Value.Alternates

06/19/2025

Syntax

```
Value.Alternates(alternates as list) as any
```

About

Expresses alternate query plans within a query plan expression obtained through `Value.Expression(Value.Optimize(...))`. Not intended for other uses.

Value.As

06/19/2025

Syntax

```
Value.As(value as any, type as type) as any
```

About

Returns the value if it's compatible with the specified type. This is equivalent to the "as" operator in M, with the exception that it can accept identifier type references such as Number.Type.

Example 1

Cast a number to a number.

Usage

```
Power Query M
```

```
Value.As(123, Number.Type)
```

Output

```
123
```

Example 2

Attempt to cast a text value to a number.

Usage

```
Power Query M
```

```
Value.As("abc", type number)
```

Output

[Expression.Error] We cannot convert the value "abc" to type Number.

Value.Compare

06/19/2025

Syntax

```
Value.Compare(value1 as any, value2 as any, optional precision as nullable number)  
as number
```

About

Returns -1, 0, or 1 based on whether the first value is less than, equal to, or greater than the second.

Value.Divide

06/19/2025

Syntax

```
Value.Divide(value1 as any, value2 as any, optional precision as nullable number)  
as any
```

About

Returns the result of dividing `value1` by `value2`. An optional `precision` parameter may be specified, by default `Precision.Double` is used.

Value.Equals

06/19/2025

Syntax

```
Value.Equals(value1 as any, value2 as any, optional precision as nullable number)  
as logical
```

About

Returns true if value `value1` is equal to value `value2`, false otherwise.

Value.Expression

06/19/2025

Syntax

```
Value.Expression(value as any) as nullable record
```

About

Returns an abstract syntax tree (AST) that represents the value's expression.

Value.Firewall

06/19/2025

Syntax

```
Value.Firewall(key as text) as any
```

About

This function is intended for internal use only.

Value.FromText

06/19/2025

Syntax

```
Value.FromText(text as any, optional culture as nullable text) as any
```

About

Decodes a value from a textual representation and interprets it as a value with an appropriate type.

- **text**: The text to interpret.
- **culture** (Optional) A specific culture used to interpret the text (for example, "en-US").

This function takes a text value and returns a value of type **number**, **logical**, **null**, **datetime**, **duration**, or **text**. An empty text value is interpreted as a **null** value.

Example 1

Convert text representing a number to its corresponding number value.

Usage

```
Power Query M
```

```
Value.FromText("12345.6789")
```

Output

```
12345.6789
```

Example 2

Convert text representing a percentage to its corresponding number value.

Usage

```
Power Query M
```

```
Value.FromText("25.4%")
```

Output

```
0.254
```

Example 3

Convert text representing a French Euro value to its corresponding number value.

Usage

```
Power Query M
```

```
Value.FromText("€1,190", "fr-FR")
```

Output

```
1.19
```

Example 4

Convert text representing a German date and time to its corresponding date and time value.

Usage

```
Power Query M
```

```
Value.FromText("24 Dez 2024 14:33:20", "de-DE")
```

Output

```
#datetime(2024, 12, 24, 14, 33, 20)
```

Related content

- [How culture affects text formatting](#)

Value.Is

06/19/2025

Syntax

```
Value.Is(value as any, type as type) as logical
```

About

Determines whether a value is compatible with the specified type. This is equivalent to the "is" operator in M, with the exception that it can accept identifier type references such as Number.Type.

Example 1

Compare two ways of determining if a number is compatible with type number.

Usage

```
Power Query M
```

```
Value.Is(123, Number.Type) = (123 is number)
```

Output

```
true
```

Value.Lineage

06/19/2025

Syntax

```
Value.Lineage(value as any) as any
```

About

This function is intended for internal use only.

Value.Metadata

06/19/2025

Syntax

```
Value.Metadata(value as any) as any
```

About

Returns a record containing the input's metadata.

Value.Multiply

06/19/2025

Syntax

```
Value.Multiply(value1 as any, value2 as any, optional precision as nullable  
number) as any
```

About

Returns the product of multiplying `value1` by `value2`. An optional `precision` parameter may be specified, by default [Precision.Double](#) is used.

Value.NativeQuery

06/19/2025

Syntax

```
Value.NativeQuery(target as any, query as text, optional parameters as any,  
optional options as nullable record) as any
```

About

Evaluates `query` against `target` using the parameters specified in `parameters` and the options specified in `options`.

The output of the query is defined by `target`.

`target` provides the context for the operation described by `query`.

`query` describes the query to be executed against `target`. `query` is expressed in a manner specific to `target` (for example, a T-SQL statement).

The optional `parameters` value may contain either a list or record as appropriate to supply the parameter values expected by `query`.

The optional `options` record may contain options that affect the evaluation behavior of `query` against `target`. These options are specific to `target`.

Value.NullableEquals

06/19/2025

Syntax

```
Value.NullableEquals(value1 as any, value2 as any, optional precision as nullable  
number) as nullable logical
```

About

Returns null if either argument `value1`, `value2` is null, otherwise equivalent to [Value.Equals](#).

Value.Optimize

06/19/2025

Syntax

```
Value.Optimize(value as any) as any
```

About

When used within Value.Expression, if `value` represents a query that can be optimized, this function indicates that the optimized expression should be returned. Otherwise, `value` will be passed through with no effect.

Value.RemoveMetadata

06/19/2025

Syntax

```
Value.RemoveMetadata(value as any, optional metaValue as any) as any
```

About

Strips the input of metadata.

Example 1

Remove all metadata from a text value.

Usage

Power Query M

```
Value.Metadata(  
    Value.RemoveMetadata("abc" meta [a = 1, b = 2])  
)
```

Output

```
[[]]
```

Example 2

Remove only one field of metadata from a text value.

Usage

Power Query M

```
Value.Metadata(  
    Value.RemoveMetadata("abc" meta [a = 1, b = 2], {"a"})  
)
```

Output

[b = 2]

Value.ReplaceMetadata

06/19/2025

Syntax

```
Value.ReplaceMetadata(value as any, metaValue as any) as any
```

About

Replaces the input's metadata information.

Value.ReplaceType

06/19/2025

Syntax

```
Value.ReplaceType(value as any, type as type) as any
```

About

Replaces the `value`'s type with the provided `type`.

Example 1

Replace the default type of a record with a more specific type.

Usage

Power Query M

```
Type.RecordFields(  
    Value.Type(  
        Value.ReplaceType(  
            [Column1 = 123],  
            type [Column1 = number]  
        )  
    )  
)[Column1][Type]
```

Output

type number

Value.Subtract

06/19/2025

Syntax

```
Value.Subtract(value1 as any, value2 as any, optional precision as nullable  
number) as any
```

About

Returns the difference of `value1` and `value2`. An optional `precision` parameter may be specified, by default `Precision.Double` is used.

Value.Traits

06/19/2025

Syntax

```
Value.Traits(value as any) as table
```

About

This function is intended for internal use only.

Value.Type

06/19/2025

Syntax

```
Value.Type(value as any) as type
```

About

Returns the type of the given value.

- **value**: The value whose type is returned.

Example 1

Return the type of the specified number.

Usage

```
Power Query M
```

```
Value.Type(243.448)
```

Output

```
type number
```

Example 2

Return the type of the specified date.

Usage

```
Power Query M
```

```
Value.Type(#datetime(2010, 12, 31))
```

Output

type date

Example 3

Return the type of the specified record.

Usage

Power Query M

```
Value.Type([a = 1, b = 2])
```

Output

type record

Related content

- [Types and type conversion](#)

Value.VersionIdentity

06/19/2025

Syntax

```
Value.VersionIdentity(value as any) as any
```

About

Returns the version identity of the `value`, or `null` if it doesn't have a version.

Value.Versions

06/19/2025

Syntax

```
Value.Versions(value as any) as table
```

About

Returns a navigation table containing the available versions of `value`.

Value.ViewError

06/19/2025

Syntax

```
Value.ViewError(errorRecord as record) as record
```

About

This function is intended for internal use only.

Value.ViewFunction

06/19/2025

Syntax

```
Value.ViewFunction(function as function) as function
```

About

This function is intended for internal use only.

Variable.Value

06/19/2025

Syntax

```
Variable.Value(identifier as text) as any
```

About

This function is intended for internal use only.

Enumerations

06/19/2025

The Power Query M formula language includes these enumerations.

List of enumerations

 Expand table

Name	Description
AccessControlKind.Type	Specifies the kind of access control.
BinaryEncoding.Type	Specifies the type of binary encoding.
BinaryOccurrence.Type	Specifies how many times the item is expected to appear in the group.
BufferMode.Type	Describes the type of buffering to be performed.
ByteOrder.Type	Specifies the byte order.
Compression.Type	Specifies the type of compression.
CsvStyle.Type	Specifies the significance of quotes in a CSV document.
Day.Type	Represents the day of the week.
ExtraValues.Type	Specifies the expected action for extra values in a row that contains columns less than expected.
GroupKind.Type	Specifies the kind of grouping.
JoinAlgorithm.Type	Specifies the join algorithm to be used in the join operation.
JoinKind.Type	Specifies the kind of join operation.
JoinSide.Type	Specifies the left or right table of a join.
LimitClauseKind.Type	Indicates the features that the specific SQL dialect supports.
MissingField.Type	Specifies the expected action for missing values in a row that contains columns less than expected.
Occurrence.Type	Specifies the occurrence of an element in a sequence.
ODataOmitValues.Type	Specifies the kinds of values an OData service can omit.

Name	Description
Order.Type	Specifies the direction of sorting.
PercentileMode.Type	Specifies the percentile mode type.
Precision.Type	Specifies the precision of comparison.
QuoteStyle.Type	Specifies the quote style.
RankKind.Type	Specifies the precise ranking method.
RelativePosition.Type	Indicates whether indexing should be done from the start or end of the input.
RoundingMode.Type	Specifies rounding direction when there is a tie between the possible numbers to round to.
SapBusinessWarehouseExecutionMode.Type	Specifies valid options for SAP Business Warehouse execution mode option.
SapHanaDistribution.Type	Specifies valid options for SAP HANA distribution option.
SapHanaRangeOperator.Type	Specifies a range operator for SAP HANA range input parameters.
TextEncoding.Type	Specifies the text encoding type.
TraceLevel.Type	Specifies the trace level.
WebMethod.Type	Specifies an HTTP method.

AccessControlKind.Type

06/19/2025

Definition

Specifies the kind of access control.

Allowed values

 Expand table

Name	Value	Description
AccessControlKind.Deny	0	Access is denied.
AccessControlKind.Allow	1	Access is allowed.

Applies to

- [Accessing data functions](#)

BinaryEncoding.Type

06/19/2025

Definition

Specifies the type of binary encoding.

Allowed values

 Expand table

Name	Value	Description
BinaryEncoding.Base64	0	Constant to use as the encoding type when base-64 encoding is required.
BinaryEncoding.Hex	1	Constant to use as the encoding type when hexadecimal encoding is required.

Applies to

- [Binary functions](#)

BinaryOccurrence.Type

06/19/2025

Definition

Specifies how many times the item is expected to appear in the group.

Allowed values

 Expand table

Name	Value	Description
BinaryOccurrence.Optional	0	The item is expected to appear zero or one time in the input.
BinaryOccurrence.Required	1	The item is expected to appear once in the input.
BinaryOccurrence.Repeating	2	The item is expected to appear zero or more times in the input.

Applies to

- [Binary functions](#)

BufferMode.Type

06/19/2025

Definition

Describes the type of buffering to be performed.

Allowed values

 Expand table

Name	Value	Description
BufferMode.Eager	1	The entire value is immediately buffered in memory before continuing.
BufferMode.Delayed	2	The type of the value is computed immediately but its contents aren't buffered until data is needed, at which point the entire value is immediately buffered.

Applies to

- [Accessing data functions](#)

ByteOrder.Type

06/19/2025

Definition

Specifies the byte order.

Allowed values

 Expand table

Name	Value	Description
ByteOrder.LittleEndian	0	The least significant byte appears first in Little Endian byte order.
ByteOrder.BigEndian	1	The most significant byte appears first in Big Endian byte order.

Remarks

The allowed values for this enumeration are possible values for the `byteOrder` parameter in [BinaryFormat.ByteOrder](#).

Applies to

- [Binary functions](#)

Compression.Type

06/19/2025

Definition

Specifies the type of compression.

Allowed values

 Expand table

Name	Value	Description
Compression.None	-1	The data is uncompressed.
Compression.GZip	0	The compressed data is in the 'GZip' format.
Compression.Deflate	1	The compressed data is in the 'Deflate' format.
Compression.Snappy	2	The compressed data is in the 'Snappy' format.
Compression.Brotli	3	The compressed data is in the 'Brotli' format.
Compression.LZ4	4	The compressed data is in the 'LZ4' format.
Compression.Zstandard	5	The compressed data is in the 'Zstandard' format.

Applies to

- [Binary functions](#)

CsvStyle.Type

06/19/2025

Definition

Specifies the significance of quotes in CSV documents.

Allowed values

 Expand table

Name	Value	Description
CsvStyle.QuoteAfterDelimiter	0	Quotes in a field are only significant immediately following the delimiter.
CsvStyle.QuoteAlways	1	Quotes in a field are always significant regardless of where they appear.

Applies to

- [Accessing data functions](#)

Day.Type

06/19/2025

Definition

Represents the day of the week.

Allowed values

 Expand table

Name	Value	Description
Day.Sunday	0	Represents Sunday.
Day.Monday	1	Represents Monday.
Day.Tuesday	2	Represents Tuesday.
Day.Wednesday	3	Represents Wednesday.
Day.Thursday	4	Represents Thursday.
Day.Friday	5	Represents Friday.
Day.Saturday	6	Represents Saturday.

Applies to

- [Date functions](#)

ExtraValues.Type

06/19/2025

Definition

Specifies the expected action for extra values in a row that contains columns less than expected.

Allowed values

 Expand table

Name	Value	Description
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.
ExtraValues.Ignore	2	If the splitter function returns more columns than the table expects, they should be ignored.

Applies to

- [Accessing data functions](#)
- [Table functions](#)

GroupKind.Type

06/19/2025

Definition

Specifies the kind of grouping.

Allowed values

 Expand table

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.

Applies to

- [Table functions](#)

JoinAlgorithm.Type

06/19/2025

Definition

Specifies the join algorithm to be used in the join operation.

Allowed values

 Expand table

Name	Value	Description
JoinAlgorithm.Dynamic	0	Automatically chooses a join algorithm based on inspecting the initial rows and metadata of both tables.
JoinAlgorithm.PairwiseHash	1	Buffers the rows of both the left and right tables until one of the tables is completely buffered, and then performs a LeftHash or RightHash, depending on which table was buffered completely. This algorithm is recommended only for small tables.
JoinAlgorithm.SortMerge	2	Performs a streaming merge based on the assumption that both tables are sorted by their join keys. While efficient, it will return incorrect results if the tables aren't sorted as expected.
JoinAlgorithm.LeftHash	3	Buffers the left rows into a lookup table and streams the right rows. For each right row, the matching left rows are found via the buffered lookup table. This algorithm is recommended when the left table is small and most of the rows from the right table are expected to match a left row.
JoinAlgorithm.RightHash	4	Buffers the right rows into a lookup table and streams the left rows. For each left row, the matching right rows are found via the buffered lookup table. This algorithm is recommended when the right table is small and most of the rows from the left table are expected to match a right row.
JoinAlgorithm.LeftIndex	5	In batches, uses the keys from the left table to do predicate-based queries against the right table. This algorithm is recommended when the right table is large, supports folding of Table.SelectRows , and contains few rows that are expected to match a left row.
JoinAlgorithm.RightIndex	6	In batches, uses the keys from the right table to do predicate-based queries against the left table. This algorithm is recommended when the left table is large, supports folding of

Name	Value	Description
		<code>Table.SelectRows</code> , and contains few rows that are expected to match a right row.

Applies to

- [Table functions](#)

JoinKind.Type

06/19/2025

Definition

Specifies the kind of join operation.

Allowed values

 Expand table

Name	Value	Description
JoinKind.Inner	0	The table resulting from an inner join contains a row for each pair of rows from the specified tables that were determined to match based on the specified key columns.
JoinKind.LeftOuter	1	A left outer join ensures that all rows of the first table appear in the result.
JoinKind.RightOuter	2	A right outer join ensures that all rows of the second table appear in the result.
JoinKind.FullOuter	3	A full outer join ensures that all rows of both tables appear in the result. Rows that did not have a match in the other table are joined with a default row containing null values for all of its columns.
JoinKind.LeftAnti	4	A left anti join returns all rows from the first table that do not have a match in the second table.
JoinKind.RightAnti	5	A right anti join returns all rows from the second table that do not have a match in the first table.
JoinKind.LeftSemi	6	A left semi join returns all rows from the first table that have a match in the second table.
JoinKind.RightSemi	7	A right semi join returns all rows from the second table that have a match in the first table.

Remarks

The fields of this enumeration are possible values for the optional `JoinKind` parameter in [Table.Join](#).

Applies to

- [Table functions](#)

JoinSide.Type

06/19/2025

Definition

Specifies the left or right table of a join.

Allowed values

 Expand table

Name	Value	Description
JoinSide.Left	0	Specifies the left table of a join.
JoinSide.Right	1	Specifies the right table of a join.

Applies to

- [Table functions](#)

LimitClauseKind.Type

06/19/2025

Definition

Describes the type of limit clause supported by the SQL dialect used by this data source.

Allowed values

 Expand table

Name	Value	Description
LimitClauseKind.None	0	This SQL dialect does not support a limit clause.
LimitClauseKind.Top	1	This SQL dialect supports a TOP specifier to limit the number of rows returned.
LimitClauseKind.LimitOffset	2	This SQL dialect supports LIMIT and OFFSET specifiers to limit the number of rows returned.
LimitClauseKind.Limit	3	This SQL dialect supports a LIMIT specifier to limit the number of rows returned.
LimitClauseKind.AnsiSql2008	4	This SQL dialect supports an ANSI SQL-compatible LIMIT N ROWS specifier to limit the number of rows returned.

Applies to

- [Accessing data functions](#)

MissingField.Type

06/19/2025

Definition

Specifies the expected action for missing values in a row that contains columns less than expected.

Allowed values

 Expand table

Name	Value	Description
MissingField.Error	0	Indicates that missing fields should result in an error. (This is the default value.)
MissingField.Ignore	1	Indicates that missing fields should be ignored.
MissingField.UseNull	2	Indicates that missing fields should be included as null values.

Applies to

- Record functions
- Table functions

Occurrence.Type

06/19/2025

Definition

Specifies the occurrence of an element in a sequence.

Allowed values

 Expand table

Name	Value	Description
Occurrence.First	0	The position of the first occurrence of the found value is returned.
Occurrence.Last	1	The position of the last occurrence of the found value is returned.
Occurrence.All	2	A list of positions of all occurrences of the found values is returned.
Occurrence.Optional	0	The item is expected to appear zero or one time in the input. Provided for backward compatibility in binary functions. In this case, use BinaryOccurrence.Optional instead.
Occurrence.Required	1	The item is expected to appear once in the input. Provided for backward compatibility in binary functions. In this case, use BinaryOccurrence.Required instead.
Occurrence.Repeating	2	The item is expected to appear zero or more times in the input. Provided for backward compatibility in binary functions. In this case, use BinaryOccurrence.Repeating instead.

Applies to

- [Table functions](#)
- [Text functions](#)

ODataOmitValues.Type

06/19/2025

Definition

Specifies the kinds of values an OData service can omit.

Allowed values

 Expand table

Name	Value	Description
ODataOmitValues.Nulls	"nulls"	Allows the OData service to omit null values.

Applies to

- [Accessing data functions](#)

Order.Type

06/19/2025

Definition

Specifies the direction of sorting.

Allowed values

 Expand table

Name	Value	Description
Order.Ascending	0	Sorts the values in ascending order.
Order.Descending	1	Sorts the values in descending order.

Applies to

- [Table functions](#)

PercentileMode.Type

06/19/2025

Definition

Specifies the percentile mode type.

Allowed values

 Expand table

Name	Value	Description
PercentileMode.ExcelInc	1	When interpolating values for List.Percentile , use a method compatible with Excel's PERCENTILE.INC  .
PercentileMode.ExcelExc	2	When interpolating values for List.Percentile , use a method compatible with Excel's PERCENTILE.EXC  .
PercentileMode.SqlDisc	3	When interpolating values for List.Percentile , use a method compatible with SQL Server's PERCENTILE_DISC .
PercentileMode.SqlCont	4	When interpolating values for List.Percentile , use a method compatible with SQL Server's PERCENTILE_CONT .

Applies to

- [List functions](#)

Precision.Type

06/19/2025

Definition

Specifies the precision of comparison.

Allowed values

 Expand table

Name	Value	Description
Precision.Double	0	An optional parameter for the built-in arithmetic operators to specify double precision.
Precision.Decimal	1	An optional parameter for the built-in arithmetic operators to specify decimal precision.

Applies to

- [Value functions](#)

QuoteStyle.Type

06/19/2025

Definition

Specifies the quote style.

Allowed values

 Expand table

Name	Value	Description
QuoteStyle.None	0	Quote characters have no significance.
QuoteStyle.Csv	1	Quote characters indicate the start of a quoted string. Nested quotes are indicated by two quote characters.

Applies to

- [Splitter functions](#)

RankKind.Type

06/19/2025

Definition

Specifies the precise ranking method.

Allowed values

 Expand table

Name	Value	Description
RankKind.Competition	0	Items which compare as equal receive the same ranking number and then a gap is left before the next ranking.
RankKind.Dense	1	Items which compare as equal receive the same ranking number and the next item is numbered consecutively with no gap.
RankKind.Ordinal	2	All items are given a unique ranking number even if they compare as equal.

Applies to

- [Table functions](#)

RelativePosition.Type

06/19/2025

Definition

Indicates whether indexing should be done from the start or end of the input.

Allowed values

 Expand table

Name	Value	Description
RelativePosition.FromStart	0	Indicates indexing should be done from the start of the input.
RelativePosition.FromEnd	1	Indicates indexing should be done from the end of the input.

Applies to

- [Text functions](#)

RoundingMode.Type

06/19/2025

Definition

Specifies rounding direction when there is a tie between the possible numbers to round to.

Allowed values

 Expand table

Name	Value	Description
RoundingMode.Up	0	Round up when there is a tie between the possible numbers to round to.
RoundingMode.Down	1	Round down when there is a tie between the possible numbers to round to.
RoundingMode.AwayFromZero	2	Round away from zero when there is a tie between the possible numbers to round to.
RoundingMode.TowardZero	3	Round toward zero when there is a tie between the possible numbers to round to.
RoundingMode.ToEven	4	Round to the nearest even number when there is a tie between the possible numbers to round to.

Applies to

- [List functions](#)

SapBusinessWarehouseExecutionMode.Type

06/19/2025

Definition

Specifies valid options for SAP Business Warehouse execution mode option.

Allowed values

 Expand table

Name	Value	Description
SapBusinessWarehouseExecutionMode.BasXml	64	'bXML flattening mode' option for MDX execution in SAP Business Warehouse.
SapBusinessWarehouseExecutionMode.BasXmlGzip	65	'Gzip compressed bXML flattening mode' option for MDX execution in SAP Business Warehouse. Recommended for low latency or high volume queries.
SapBusinessWarehouseExecutionMode.DataStream	66	'DataStream flattening mode' option for MDX execution in SAP Business Warehouse.

Applies to

- [Accessing data functions](#)

SapHanaDistribution.Type

06/19/2025

Definition

Specifies valid options for SAP HANA distribution option.

Allowed values

 Expand table

Name	Value	Description
SapHanaDistribution.Off	0	'Off' distribution option for SAP HANA.
SapHanaDistribution.Connection	1	'Connection' distribution option for SAP HANA.
SapHanaDistribution.Statement	2	'Statement' distribution option for SAP HANA.
SapHanaDistribution.All	3	Returns the packages in an SAP HANA database.

Applies to

- [Accessing data functions](#)

SapHanaRangeOperator.Type

06/19/2025

Definition

Specifies a range operator for SAP HANA range input parameters.

Allowed values

 Expand table

Name	Value	Description
SapHanaRangeOperator.GreaterThan	0	'Greater than' range operator for SAP HANA input parameters.
SapHanaRangeOperator.LessThan	1	'Less than' range operator for SAP HANA input parameters.
SapHanaRangeOperator.GreaterThanOrEquals	2	'Greater than or equals' range operator for SAP HANA input parameters.
SapHanaRangeOperator.LessThanOrEquals	3	'Less than or equals' range operator for SAP HANA input parameters.
SapHanaRangeOperator.Equals	4	'Equals' range operator for SAP HANA input parameters.
SapHanaRangeOperator.NotEquals	5	'Not equals' range operator for SAP HANA input parameters.

Applies to

- [Accessing data functions](#)

TextEncoding.Type

06/19/2025

Definition

Specifies the text encoding type.

Allowed values

 Expand table

Name	Value	Description
TextEncoding.Utf16	1200	Use to choose the UTF16 little endian binary form.
TextEncoding.Unicode	1200	Use to choose the UTF16 little endian binary form.
TextEncoding.BigEndianUnicode	1201	Use to choose the UTF16 big endian binary form.
TextEncoding.Windows	1252	Use to choose the Windows binary form.
TextEncoding.Ascii	20127	Use to choose the ASCII binary form.
TextEncoding.Utf8	65001	Use to choose the UTF8 binary form.

Applies to

- [Text functions](#)

TraceLevel.Type

06/19/2025

Definition

Specifies the trace level.

Allowed values

 Expand table

Name	Value	Description
TraceLevel.Critical	1	Specifies the Critical trace level.
TraceLevel.Error	2	Specifies the Error trace level.
TraceLevel.Warning	4	Specifies the Warning trace level.
TraceLevel.Information	8	Specifies the Information trace level.
TraceLevel.Verbose	16	Specifies the Verbose trace level.

Applies to

- [Error handling](#)

WebMethod.Type

06/19/2025

Definition

Specifies an HTTP method.

Allowed values

 Expand table

Name	Value	Description
WebMethod.Delete	"DELETE"	Specifies the DELETE method for HTTP.
WebMethod.Get	"GET"	Specifies the GET method for HTTP.
WebMethod.Head	"HEAD"	Specifies the HEAD method for HTTP.
WebMethod.Patch	"PATCH"	Specifies the PATCH method for HTTP.
WebMethod.Post	"POST"	Specifies the POST method for HTTP.
WebMethod.Put	"PUT"	Specifies the PUT method for HTTP.

Remarks

These fields only work in the context of custom connectors.

Applies to

- [Accessing data functions](#)

Constants

06/19/2025

The Power Query M formula language includes these constant values.

List of constants

 Expand table

Name	Description
Number.E	Returns 2.7182818284590451, the value of e up to 16 decimal digits.
Number.Epsilon	Returns the smallest possible number.
Number.NaN	Represents 0/0.
Number.NegativeInfinity	Represents -1/0.
Number.Pi	Returns 3.1415926535897931, the value for Pi up to 16 decimal digits.
Number.PositiveInfinity	Represents 1/0.

See also

- [Number functions](#)

Number.E

06/19/2025

About

A constant that represents 2.7182818284590451, the value for e up to 16 decimal digits.

Number.Epsilon

06/19/2025

About

A constant value that represents the smallest positive number a floating-point number can hold.

Number.NaN

06/19/2025

About

A constant value that represents 0 divided by 0.

Number.NegativeInfinity

06/19/2025

About

A constant value that represents -1 divided by 0 .

Number.PI

06/19/2025

About

A constant that represents 3.1415926535897932, the value for pi up to 16 decimal digits.

Number.PositiveInfinity

06/19/2025

About

A constant value that represents 1 divided by 0.

Dynamic values

06/19/2025

The Power Query M formula language includes these dynamic values.

List of dynamic values

 Expand table

Name	Description
Culture.Current	Returns the name of the current culture for the application.
TimeZone.Current	Returns the name of the current time zone for the application.

See also

- [Comparer functions](#)
- [DateTimeZone functions](#)

Culture.Current

06/19/2025

About

Returns the name of the current culture for the application.

Related content

- [How culture affects text formatting](#)

TimeZone.Current

06/19/2025

About

Returns the name of the current time zone for the application.