#binary

In this article, we will explore the M code behind the #binary function and how to use it to work with binary data in Power Query.

Understanding Binary Data

Before we dive into the #binary function, let's first understand what binary data is. Binary data refers to any data that is stored in a binary format, consisting of sets of 0s and 1s. This includes images, audio, and video files, as well as other data types such as PDFs, ZIP files, and more.

When working with binary data, it can be useful to encode it in a specific format to make it easier to work with. Common encoding formats include Base64, Hexadecimal, and ASCII.

The #binary Function

The #binary function in Power Query is used to create a binary value from a given input. It takes a single argument, which can be any data type that can be converted to binary, such as a text string or a number.

For example, if we wanted to create a binary value from the text string "Hello World", we could use the following M code:

```
#binary([Text.ToBinary("Hello World")])
```

This would create a binary value that represents the text string "Hello World". We could then use this binary value in our Power Query transformations.

Reading Binary Data

Once we have binary data in Power Query, we may need to read or decode it to work with it. The #binary function also provides a way to read binary data using the following syntax:

```
#binary([Binary.FromText("base64-encoded-data")])
```

This would read the base64-encoded data and return it as a binary value that we could use in our transformations.

#date

What is the #date function?
The #date function is used in Power Query to create a date value from year, month, and day. The syntax for the function is as follows:

#date(year, month, day)

For example, the following formula creates a date value for January 1st, 2022:

#date(2022, 1, 1)

Understanding the M code behind the #date function
Behind the scenes, the #date function is actually a simple M expression that combines the year, month, and day values into a date value. The M expression for the #date function is as follows:

Date.From([Year] & "-" & [Month] & "-" & [Day])

This expression uses the Date.From function to convert a text value into a date value. The text value is created by combining the year, month, and day values using the “&” operator.
Let’s break down this expression further:
– The [Year], [Month], and [Day] values are parameters that are passed to the function.
– The “&” operator is used to concatenate the year, month, and day values into a text value. The resulting text value has the format “YYYY-MM-DD”.

#datetime

Understanding the #datetime function

The #datetime function is a part of the M language, which is used to create custom functions in Power Query. The function takes three arguments – year, month, and day – and returns a datetime value. Here is the syntax of the #datetime function:

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

Here, year, month, and day are integer values, representing the year, month, and day of the date you want to create. For example, if you want to create a datetime value for 1st January 2022, you would use the following code:

```
#datetime(2022, 1, 1)
```

The M code behind the #datetime function

The M code behind the #datetime function is relatively simple. Here is the code:

```
DateTime.Local(year, month, day, 0, 0, 0)
```

As you can see, the #datetime function uses the DateTime.Local function, which is a part of the M language. The DateTime.Local function takes six arguments – year, month, day, hour, minute, and second – and returns a datetime value. In the #datetime function, we only use the first three arguments, as we are not concerned with the time component of the datetime value.

Using the DateTime.Local function

The DateTime.Local function is a powerful function that can be used to create datetime values in a variety of ways. Here are some

#duration

Introduction to the #duration function

The #duration function is used to create a duration value in Power Query. It is commonly used in scenarios where we need to calculate the duration between two dates or times. The function takes two arguments, the first argument is the start time, and the second argument is the end time. The format for both arguments is a date/time type.

The syntax for the #duration function is as follows:

```
#duration(duration_hours, duration_minutes, duration_seconds)
```

For example, to create a duration of one hour and thirty minutes, we can write the following M code:

```
#duration(1, 30, 0)
```

This will return a duration value of 1 hour and 30 minutes.

Understanding the M code behind the #duration function

To understand the M code behind the #duration function, we need to understand the data types used in Power Query M language.

Power Query M language uses the following data types:

– Text
– Number
– Date/Time
– Logical
– List
– Record

The #duration function is built using the Date/Time data type. When we pass the start and end time to the function, Power Query

#time

What is the #time function?

The #time function is a part of the DateTime.Local function family in Power Query. It allows you to create a new datetime value by specifying the hour, minute, and second components. The syntax for the #time function is as follows:

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

For example, to create a datetime value for 10:30:00, you would use the following M code:

```
#time(10, 30, 0)
```

Understanding the M code behind #time

The M code behind the #time function is quite simple. It takes the hour, minute, and second components and returns a new datetime value. Let's take a closer look at the M code for the #time function:

```
(DateTime.LocalTime(hour, minute, second))
```

As you can see, the M code for the #time function uses the DateTime.LocalTime function to create a new datetime value. The DateTime.LocalTime function takes three parameters – hour, minute, and second – and returns a new datetime value with the specified time components.

Using the #time function in Power Query

Now that we understand the M code behind the #time function, let's take a look at how we can use it in Power Query. One common use

Access.Database

What is Access.Database?

Access.Database is a function in Power Query that is used to connect to and access data from Microsoft Access databases. It is part of the Microsoft Access Database Engine Redistributable, which is a set of components that enable communication with Microsoft Access databases. Access.Database can be used to connect to any Access database, whether it is stored locally or on a network server.

Understanding the M Code Behind Access.Database

The M code behind Access.Database is what enables Power Query to connect to and access data from Access databases. The code is written in the M language, which is the query language used by Power Query. The M code for Access.Database is as follows:

```
Access.Database(file as text, optional options as nullable record) as table
```

Let's break down what this code means. The first parameter, "file as text", specifies the path and file name of the Access database that you want to connect to. This can be either a local file path or a network path. The second parameter, "optional options as nullable record", is an optional parameter that allows you to specify additional options for the connection. These options can include things like user credentials and query timeouts.

The output of the Access.Database function is a table that contains all of the data in the specified Access database. This table can then be used for further data transformation and analysis within Power Query.

Using Access.Database in Power Query

To use Access.Database in Power Query, you must first have the Microsoft Access Database Engine Redistributable installed on your computer. This can be downloaded from the Microsoft website. Once you have the engine installed, you can follow these steps to connect to an Access database using Access.Database:

1. Open Microsoft Excel and create a new workbook.
2. Click on the "Data" tab in the ribbon.
3. Click on the "Get Data" button and select "From Database" > "From Microsoft Access Database" from the dropdown menu.
4. In the "Access database" dialog box that appears, enter the path and file name of the Access database that you want to connect to. Click "OK".

AccessControlEntry.ConditionToIdentities

Understanding the AccessControlEntry.ConditionToIdentities Function

The AccessControlEntry.ConditionToIdentities function is a part of the Active Directory module in Power Query M. The function takes a security group condition as an input and returns the corresponding identities. The security group condition is a combination of the AD security group name, domain name, and group scope. The function uses this information to retrieve the corresponding Active Directory object, and then returns the object's distinguished name and object class.

The M Code Behind the AccessControlEntry.ConditionToIdentities Function

The M code behind the AccessControlEntry.ConditionToIdentities function is quite complex. It includes several nested functions and variables that work together to retrieve the security group object and return its distinguished name and object class. Here is a breakdown of the M code:

```
let
  ConditionToIdentities = (condition) =>

    let
      domainName = SplitString(condition, ":"){0},
      groupName = SplitString(condition, ":"){1},
      scope = SplitString(condition, ":"){2},
      group = GetGroupObject(domainName, groupName, scope),
      dn = group[attributes][distinguishedName][0],
      objectClass = GetObjectClass(group)
    in
      [dn = dn, objectClass = objectClass],

  GetGroupObject = (domainName, groupName, scope) =>

    let
ActiveDirectory.Domains
```

What is Active Directory?

Active Directory (AD) is a Microsoft directory service that allows businesses to manage user and computer accounts, security groups, and other resources from a central location. It is a hierarchical database that stores information about objects on a network, such as users, groups, computers, and printers. Active Directory is commonly used in enterprise environments to manage permissions, authentication, and network resources.

The ActiveDirectory.Domains M Function

The ActiveDirectory.Domains function is a Power Query M function that allows users to retrieve a list of all domains in an Active Directory forest. This function is incredibly useful for administrators who need to manage multiple domains and want to automate the process of retrieving information about them.

Syntax

The syntax for the ActiveDirectory.Domains function is as follows:

```
ActiveDirectory.Domains(server as text, optional options as nullable record) as table
```

The function takes two parameters: the server name and an options record. The server parameter is a text value that specifies the name of the domain controller to connect to. The options record is an optional parameter that allows users to specify additional options for the connection, such as the authentication method to use.

Example

Here is an example of how to use the ActiveDirectory.Domains function:

```
let
  Source = ActiveDirectory.Domains("dc01.mydomain.local"),
  #"Expanded domains" = Table.ExpandTableColumn(Source, "domains", {"distinguishedName", "name", "netBiosName", "dnsRoot"},
{"distinguishedName", "name", "netBiosName", "dnsRoot"})
```

AdobeAnalytics.Cubes

Introduction to Adobe Analytics

Adobe Analytics is a digital analytics tool used by marketers, analysts, and businesses to track and analyze website traffic, user behavior, and campaigns. Adobe Analytics provides various features to help users to analyze their website data. Adobe Analytics.Cubes is one such feature that enables users to create custom cubes and perform complex data analysis. The AdobeAnalytics.Cubes M function in Power Query helps to connect, transform, and load data from Adobe Analytics.Cubes into Excel or Power BI.

The M Code behind the AdobeAnalytics.Cubes Function

The AdobeAnalytics.Cubes M function is a part of the Power Query M function library. It helps users to connect to Adobe Analytics.Cubes, fetch data, and transform it according to their requirements. Here is the M code behind the AdobeAnalytics.Cubes function:

```
let
  Source = AdobeAnalytics.Cubes(
    cubeID,
    startDate,
    endDate,
    dimensions,
    metrics,
    segments,
    dateGranularity,
    timePeriods,
    customGranularity,
    customGranularityName,
    maxRows
  ),
  #"Expanded Data" = Table.ExpandTableColumn(
    Source,
```

AdoDotNet.DataSource

Overview of AdoDotNet.DataSource

AdoDotNet.DataSource is a Power Query M function that allows users to connect to various data sources using .NET data providers. The function takes two arguments, the first being a connection string that specifies the data source, and the second being a query that retrieves the data from the source.

The function uses the .NET Framework's System.Data.OleDb.OleDbConnection class to connect to the data source. This class provides a way to connect to a variety of data sources, including Microsoft Access, Excel, SQL Server, and Oracle.

Syntax of AdoDotNet.DataSource

The syntax of AdoDotNet.DataSource is as follows:

```
AdoDotNet.DataSource(connectionString as text, query as text) as table
```

The first argument, connectionString, is a text value that specifies the connection string to the data source. The second argument, query, is a text value that specifies the query to retrieve data from the data source. The function returns a table that contains the retrieved data.

Examples of AdoDotNet.DataSource

Here are some examples of how AdoDotNet.DataSource can be used:

Example 1: Connect to a SQL Server database

```
let
  connectionString = "Provider=SQLNCLI11;Server=myServerAddress;Database=myDataBase;Uid=myUsername;Pwd=myPassword;",
  query = "SELECT  FROM Customers",
  Source = AdoDotNet.DataSource(connectionString, query)
in
  Source
```

AdoDotNet.Query

The AdoDotNet.Query function is one of the many M functions available in Power Query. This function allows users to execute SQL queries against a variety of database systems, including Microsoft SQL Server, Oracle, and MySQL. In this article, we will explore the M code behind the AdoDotNet.Query function and how it can be used to access and manipulate data in a database.

Understanding AdoDotNet.Query

Before delving into the M code behind AdoDotNet.Query, it is helpful to understand the function itself. AdoDotNet.Query is a function that takes three arguments: connectionString, sqlStatement, and options. The connectionString argument is a string that defines the connection information for the database. The sqlStatement argument is a string that contains the SQL query to be executed. The options argument is an optional record that contains additional parameters for the query, such as the timeout value or the type of command to be executed.

When executed, the AdoDotNet.Query function returns a table that contains the results of the SQL query. This table can then be transformed and loaded into a variety of destinations using other Power Query functions.

The M Code Behind AdoDotNet.Query

The M code behind the AdoDotNet.Query function is relatively straightforward. At its core, the function is simply a wrapper around the .NET System.Data.SqlClient.SqlCommand class, which is used to execute SQL queries against a Microsoft SQL Server database.

Here is the M code for the AdoDotNet.Query function:

```
let AdoDotNet.Query = (connectionString as text, sqlStatement as text, optional options as record) =>
  let
    conn = AdoDotNetProvider.GetConnection(connectionString),
    command = conn.CreateCommand(),
    commandTimeout = if options <> null and options[CommandTimeout] <> null then options[CommandTimeout] else null,
    commandType = if options <> null and options[CommandType] <> null then options[CommandType] else Sql.CommandType.Text,
    resultTable = Table.FromRows(Sql.DataReader(command.ExecuteReader(CommandBehavior.CloseConnection),
commandTimeout), command.ExecuteReader().GetSchemaTable())
  in
    resultTable
```

AnalysisServices.Database

At the heart of Power Query lies the M language, a functional programming language used to write queries. The M language is used to perform a variety of tasks within Power Query, from importing data to transforming and shaping it. In this article, we'll delve into the M code behind the Power Query M function AnalysisServices.Database and explore how it can be used to analyze data.

What is AnalysisServices.Database?

AnalysisServices.Database is a function within Power Query that allows you to connect to and query data from SQL Server Analysis Services. Analysis Services is a tool provided by Microsoft that lets you create and manage multidimensional data models and OLAP cubes. It can also be used to create tabular data models, which can be queried with Power Query.

How to use AnalysisServices.Database

To use AnalysisServices.Database in Power Query, you first need to connect to your Analysis Services server. This can be done by clicking on the "Get Data" button in the Home tab of the Power Query ribbon, selecting "SQL Server Analysis Services" from the list of available data sources, and entering the server name and any necessary credentials.

Once you've connected to your server, you can use the AnalysisServices.Database function to query your data. The function takes three arguments:

– Server: The name of the Analysis Services server that you want to query.

– Database: The name of the database that you want to query.

– Options: A record containing any additional options that you want to pass to the function.

Here's an example of how to use the function to query data from an Analysis Services database:

```
let
  Source = AnalysisServices.Database("localhost", "AdventureWorksDW2017", [Query="SELECT  FROM FactInternetSales"])
in
  Source
```

This code connects to the local Analysis Services server, selects the AdventureWorksDW2017 database, and queries all the data from the FactInternetSales table.

AnalysisServices.Databases

Introduction to AnalysisServices.Databases

AnalysisServices.Databases is a Power Query M function that is used to retrieve a list of databases from a Microsoft SQL Server Analysis Services server. This function requires the following parameters:

– server: The name of the Analysis Services server.

– optional options: A record containing optional parameters for the function.

The AnalysisServices.Databases function returns a table that contains the following columns:

– Name: The name of the database.

– ID: The ID of the database.

– Description: A description of the database.

– Created: The date and time when the database was created.

– LastProcessed: The date and time when the database was last processed.

The M Code Behind AnalysisServices.Databases

The M code behind the AnalysisServices.Databases function is relatively complex, but it can be broken down into several parts. Let's take a closer look at each part:

Part 1: Connection to Analysis Services Server

The first part of the M code establishes a connection to the Analysis Services server using the server parameter. This is done using the following code:

```
let
  Source = AnalysisServices.Database(server),
  ...
```

This code creates a new data source using the AnalysisServices.Database function and passes the server parameter. The resulting data source is stored in the Source variable.

Part 2: Retrieve List of Databases

AzureStorage.BlobContents

In this article, we'll take a closer look at the M code behind the AzureStorage.BlobContents function, and explore some of the key features and benefits of this powerful tool.

Understanding the AzureStorage.BlobContents Function

At its core, the AzureStorage.BlobContents function is designed to enable users to read data stored in Azure Blob Storage directly into Power Query. This function takes three main arguments:

By specifying these arguments, users can extract data from virtually any type of blob, regardless of its format or structure.

Key Features of the AzureStorage.BlobContents Function

One of the key benefits of the AzureStorage.BlobContents function is its ease of use. With just a few lines of code, users can extract data from Azure Blob Storage and load it directly into Power Query.

Another important feature of this function is its flexibility. Users can specify a wide range of options, such as headers, timeouts, and encoding formats, to ensure that they retrieve the exact data they need.

In addition to these features, the AzureStorage.BlobContents function also offers robust error handling capabilities, allowing users to easily identify and address any potential issues that may arise during the data extraction process.

Best Practices for Using the AzureStorage.BlobContents Function

To get the most out of the AzureStorage.BlobContents function, there are a few key best practices that users should keep in mind:

1. Always specify the encoding format of the data in the blob. This will ensure that the data is properly formatted and can be loaded into Power Query without any issues.

3. Test your code thoroughly before using it in production, to ensure that you're able to extract the data you need without any errors or issues.

By following these best practices, users can leverage the full power of the AzureStorage.BlobContents function to extract and analyze

AzureStorage.Blobs

In this article, we will take a closer look at the M code behind the Power Query M function AzureStorage.Blobs and explore how it can be used to connect to and manipulate data stored in Azure Blob Storage.

Understanding Azure Blob Storage

Before we dive into the M code behind the AzureStorage.Blobs function, let's first understand what Azure Blob Storage is and how it works.

Azure Blob Storage is a cloud-based object storage solution that allows users to store and access large amounts of unstructured data, such as text or binary data. It offers three types of blobs: block blobs, append blobs, and page blobs, each designed for a specific use case.

Block blobs are optimized for streaming and storing large files, such as videos or images. Append blobs are ideal for scenarios where data needs to be appended to an existing file, such as log files. Page blobs are used for virtual machine disks and offer random read/write access to data.

Azure Blob Storage allows users to create containers to organize their data and offers various access tiers, including hot, cool, and archive, each designed for a specific use case. It also provides a range of security features, such as encryption at rest and in transit, role-based access control, and firewall and virtual network integration.

Using the AzureStorage.Blobs Function in Power Query

Now that we have a basic understanding of Azure Blob Storage, let's explore how we can use the AzureStorage.Blobs function in Power Query to connect to and manipulate data stored in Azure Blob Storage.

The AzureStorage.Blobs function is part of the Power Query M language and allows users to connect to and retrieve data from Azure Blob Storage. To use the function, we need to provide the function with the following parameters:

– Account name: the name of the Azure Storage account

– Container name: the name of the container where the data is stored

– Folder path: the path to the folder where the data is stored

– Recursive: a boolean value that indicates whether to retrieve data from subfolders as well

Once we have provided the function with these parameters, we can use it to retrieve data from Azure Blob Storage in Power Query.

Examples of Using AzureStorage.Blobs

Let's take a look at some examples of how we can use the AzureStorage.Blobs function in Power Query.

Example 1: Retrieving Data from a Single Blob

AzureStorage.DataLake

What is Azure Data Lake Storage?

Azure Data Lake Storage is a cloud-based storage solution from Microsoft that is optimized for big data workloads. It is built on top of the Azure Blob Storage and provides a distributed file system that can store and process petabytes of data. With Azure Data Lake Storage, you can store any type of data, including structured, semi-structured, and unstructured data.

Understanding the AzureStorage.DataLake Function

The AzureStorage.DataLake function is a part of the Power Query M language that allows users to read and write data to and from Azure Data Lake Storage. It is a powerful function that enables users to perform a wide range of data manipulation tasks using M language.

The syntax for the AzureStorage.DataLake function is as follows:

```
AzureStorage.DataLake(accountName as text, optional options as nullable record) as function
```

The function takes two parameters:

– `accountName`: The name of the Azure Data Lake Storage account.

– `options`: An optional record that contains additional parameters for the function.

The M Code Behind the AzureStorage.DataLake Function

The AzureStorage.DataLake function is implemented using M language code that interacts with the Azure Data Lake Storage REST API. The M code behind the function is responsible for connecting to the Azure Data Lake Storage account, authenticating the user, and performing the requested data operations.

Here is a high-level overview of the M code behind the AzureStorage.DataLake function:

1. The M code starts by creating a URL string that points to the Azure Data Lake Storage account.
2. The code then creates a web request using the URL string and sets the necessary headers for authentication.
3. The web request is then sent to the Azure Data Lake Storage REST API, which returns a response.
4. The M code then parses the response and returns the requested data.

Here is the detailed breakdown of the M code behind the AzureStorage.DataLake function:

```
AzureStorage.DataLakeContents
```

What is Azure Data Lake Storage?

Azure Data Lake Storage is a cloud-based storage solution that allows users to store and analyze big data in a cost-effective and scalable way. It provides two types of storage accounts: Gen1 and Gen2. Both storage accounts offer similar functionalities but have different features. Gen1 storage accounts provide a hierarchical namespace, while Gen2 storage accounts provide a flat namespace. This article will focus on the M code for accessing data from both storage account types.

AzureStorage.DataLakeContents Function

The AzureStorage.DataLakeContents function is a custom M function that allows users to access and transform data from Azure Data Lake Storage accounts. The function takes two parameters: the path to the data and the optional options parameter. The path parameter is a string that specifies the path to the data in the Azure Data Lake Storage account. The options parameter is an optional record that specifies additional options for the function. The function returns a table with the contents of the data.

The M code for the AzureStorage.DataLakeContents function is as follows:

```
let
  AzureStorage.DataLakeContents = (path as text, optional options as nullable record) as table =>
  let
    options_defaults = [
      HierarchicalNavigation = true,
      IncludeSubfolders = true,
      InferDataTypes = true,
      QuoteCharacter = """""""""""""""",
      FirstRowAsHeader = true,
      Encoding = null,
      Culture = null,
      Delimiter = null,
      ExtraColumns = null,
      SkipLines = 0,
AzureStorage.Tables
```

What is Azure Storage Tables?

Azure Storage Tables is a NoSQL key-value storage service that allows users to store and query large amounts of structured data in the cloud. It provides a scalable and cost-effective solution for storing data, and it is widely used in various applications, including IoT, e-commerce, and finance.

With Azure Storage Tables, you can store data in a table format consisting of rows and columns. Each table can have up to billions of rows, and each row can have up to 252 properties. The data in each property can be of different data types, including string, binary, integer, Boolean, and datetime.

What is Power Query M?

Power Query M is a functional programming language that enables users to perform data transformations and manipulations in Power BI, Excel, and other applications. It uses a combination of functions, operators, and expressions to transform data into a desired format. One of the key benefits of Power Query M is that it is highly scalable and can handle large amounts of data.

The AzureStorage.Tables Function

Here is an example of how to use the AzureStorage.Tables function to retrieve data from an Azure Storage Table:

```
let
  accountName = “myaccount”,
  tableName = “mytable”,
  source = AzureStorage.Tables(accountName, tableName)
in
  source
```

Basic Querying

Binary data

What is Binary Data?

Binary data refers to data that is stored in a binary format, rather than a text format. In other words, binary data is represented as a series of 1's and 0's, rather than as characters. Some common examples of binary data include images, videos, and audio files. Binary data is often used in computing because it can be processed much more quickly than text data.

The M Code for Binary Data

The M function for binary data is called Binary.FromText. This function takes a text value as input, and returns a binary value. The syntax for Binary.FromText is as follows:

```
Binary.FromText(text as text, optional encoding as nullable number) as binary
```

The text parameter is the text value that you want to convert to binary data. The encoding parameter is an optional parameter that specifies the character encoding that should be used when converting the text to binary data. If you don't specify an encoding, Power Query will use the default encoding for your system.

Here's an example of how you might use Binary.FromText to convert a text value to binary data:

```
let
  textValue = "Hello, world!",
  binaryValue = Binary.FromText(textValue)
in
  binaryValue
```

In this example, we're converting the text value "Hello, world!" to binary data using the Binary.FromText function. The result is a binary value that represents the text value.

Binary.ApproximateLength

Understanding Binary Data

Before diving into the M code behind Binary.ApproximateLength, it's important to understand what binary data is and how it's used. Binary data is a sequence of 1s and 0s that is used to represent information in a computer system. This information could be anything from text to images to executable code.

In Power Query, binary data is often used when working with files, such as PDFs, images, and videos. These files are typically stored in binary format, which means that the data is represented as a series of bytes (groups of 8 bits). Each byte can have a value between 0 and 255, which corresponds to a unique combination of 1s and 0s.

The Binary.ApproximateLength Function

The Binary.ApproximateLength function in Power Query is used to calculate the approximate length of a binary value. This function takes a binary value as its input and returns the approximate length of that value in bytes.

Here's the M code for the Binary.ApproximateLength function:

```
(Binary as binary) as number =>
let
  Length = List.Count(Binary),
  ApproximateLength = Number.RoundUp(Length/1024)1024
in
  ApproximateLength
```

Let's break down this code and see how it works.

Breaking Down the M Code

The Binary.ApproximateLength function takes a single parameter, Binary, which is the binary value that you want to calculate the approximate length of. This parameter is defined as a binary data type.

The first step in the function is to calculate the length of the binary value. This is done using the List.Count function, which returns the number of elements in a list. In this case, the binary value is treated as a list of bytes, so the List.Count function returns the number of

Binary.Buffer

What is Binary.Buffer?

Binary.Buffer is a Power Query M function that allows you to cache the results of a query or subquery. The function takes a single argument, the query or subquery to cache, and returns a binary value that can be used to retrieve the cached results.

Here's an example of how Binary.Buffer can be used in a query:

```
let
  Source = Sql.Database("localhost", "MyDatabase"),
  CachedQuery = Binary.Buffer(
    Source{[Schema="dbo",Table="MyTable"]}[Data]
  ),
  FilteredQuery = Table.SelectRows(
    CachedQuery,
    each [Column1] = "Value"
  )
in
  FilteredQuery
```

In this example, we're using Binary.Buffer to cache the results of a SQL query. The CachedQuery variable contains the cached results, which we then filter using Table.SelectRows.

How Does Binary.Buffer Work?

When you use Binary.Buffer to cache a query or subquery, Power Query stores the results in memory. The function returns a binary value that represents the cached results. This binary value can be used to retrieve the cached results later on.

Here's how the caching process works:

1. The query or subquery is executed and the results are stored in memory.
2. Binary.Buffer generates a binary value that represents the cached results.

Binary.Combine

What is Binary.Combine?

Binary.Combine is a function in Power Query that is used to combine multiple binary values into a single binary value. This function is often used in scenarios where data needs to be compressed or combined into a single value for storage or transmission.

The Binary.Combine function takes two or more binary values as input and returns a single binary value. The resulting binary value is the concatenation of the input binary values in the order they were passed to the function.

The M Code Behind Binary.Combine

The M code behind Binary.Combine is relatively simple, but it can be a bit confusing for those who are not familiar with the language. Here is the M code that powers Binary.Combine:

```
(Binary.Combine) =>
let
  Output = Binary.FromText(List.Combine(List.Transform(Binary.ToList(Binary.Combine), each List.Buffer(_))))
in
  Output
```

Let's break down this code and explore what each part does.

The Function Definition

The first line of the M code defines the Binary.Combine function. This line tells Power Query that we are defining a new function called Binary.Combine. The "=>" symbol is used to separate the function definition from the function body.

The Function Body

The function body is the code that is executed when the Binary.Combine function is called. In this case, the function body consists of two parts: the "let" statement and the "in" statement.

The "let" Statement

The "let" statement is used to define variables that are used in the function body. In this case, we are defining a variable called "Output".

Binary.Compress

In this article, we'll dive into the M code behind Binary.Compress and explore how it works.

What is Binary.Compress?

Binary.Compress is a Power Query M function that compresses binary data using the GZip algorithm. It takes a binary value as its input and returns the compressed binary data as output.

Here's an example of how to use Binary.Compress in Power Query:

```
let
  Source = Binary.FromText("Hello, World!"),
  Compressed = Binary.Compress(Source),
  Decompressed = Binary.Decompress(Compressed)
in
  Decompressed
```

In this example, we first convert the text "Hello, World!" to binary using the Binary.FromText function. We then compress the binary data using Binary.Compress and store the result in the variable Compressed. Finally, we decompress the compressed data using Binary.Decompress and store the result in the variable Decompressed.

The M Code Behind Binary.Compress

Let's take a closer look at the M code behind Binary.Compress. Here's the full code:

```
(Binary as binary, optional compressionLevel as nullable number) =>
let
  compressedBinary = Binary.Buffer(List.Buffer(GZipCompress.Binary(binary, compressionLevel)))
in
  compressedBinary
```

Binary.Decompress

The Binary.Decompress function takes two arguments: the compressed binary data and an optional encoding type. The encoding type argument is optional because the function can automatically detect the encoding type of the compressed data.

How Binary.Decompress Works

The Binary.Decompress function uses the DEFLATE algorithm to decompress the binary data. The DEFLATE algorithm is a lossless compression algorithm that is commonly used to compress data for storage or transmission.

The DEFLATE algorithm works by analyzing the input data and replacing repeated patterns with references to previously-encountered patterns. This allows the algorithm to achieve high compression ratios while retaining the original data without loss.

The Binary.Decompress function takes the compressed binary data as input and applies the DEFLATE algorithm to it. The resulting output is the decompressed data in binary format.

Example Usage

Suppose we have a compressed binary file containing data that we want to analyze using Power Query. We can use the Binary.Decompress function to decompress the file and load the data into Power Query for further analysis.

Here is an example of how to use the Binary.Decompress function in Power Query:

```
let
  compressedData = Binary.Buffer(File.Contents("C:compressedData.bin")),
  decompressedData = Binary.Decompress(compressedData),
  outputTable = Table.FromBinary(decompressedData)
in
  outputTable
```

In this example, we first load the compressed binary data from a file using the File.Contents function. We then use the Binary.Buffer function to store the data in memory as a binary buffer.

Next, we pass the binary buffer to the Binary.Decompress function to decompress the data. Finally, we use the Table.FromBinary function to convert the decompressed binary data into a Power Query table for analysis.

Binary.From

Understanding Binary Data

Before we dive into the M code behind Binary.From, let's first understand what binary data is and why it's important. Binary data is any data stored in binary format, which is a system of representing data using only two digits: 0 and 1. This system is used by computers to store and manipulate data at the lowest level.

Binary data is commonly used for storing images, audio, and video files, as well as other types of data that require a high level of precision. It's also used in encryption algorithms, where it provides a way to securely store and transmit sensitive information.

The Binary.From Function

The Binary.From function in Power Query is used to convert binary data into a text format. The function takes a binary value as input and returns a text value. The syntax of the function is as follows:

```
Binary.From(binary as binary) as text
```

The function takes a single parameter, binary, which is the binary value to be converted. The binary value can be any valid binary data, such as a binary file or a binary string.

The M Code Behind Binary.From

Now that we understand what Binary.From does, let's take a closer look at the M code behind the function. The M code is what Power Query uses to perform the actual conversion from binary to text.

Here's the M code for Binary.From:

```
let
  Source = #binary({0}),
  Value = Source{0},
  Text = Text.From(Value)
in
Binary.FromList
```

What is the Binary.FromList Function?

The Binary.FromList function in Power Query M is used to convert a list of values into a binary value. The function takes a list of values as an input and returns a binary value. The binary value is a representation of the input list in a binary format.

The Binary.FromList function is useful in a variety of applications where binary values are required. For example, it can be used in cryptography to encrypt and decrypt data, in data compression to compress and decompress data, and in data transmission to send and receive data.

The M Code Behind Binary.FromList

The M Code behind the Binary.FromList function is relatively simple. The function takes a list of values as an input and converts each value to a binary format. The binary values are then concatenated to create a binary value that represents the input list.

The following is the M Code for the Binary.FromList function:

```
(Binary.FromList) => (list as list) =>
  let
    binaryList = List.Transform(list, (x) => Number.ToBinary(x)),
    binaryString = Text.Combine(binaryList, ""),
    binaryValue = Binary.FromText(binaryString)
  in
    binaryValue
```

In this code, the list of values is first transformed using the List.Transform function. The List.Transform function applies a transformation function to each value in the list, in this case the Number.ToBinary function. This function converts each value in the list to a binary format.

The resulting binary values are then concatenated using the Text.Combine function. This function combines the binary values into a single binary string.

Finally, the Binary.FromText function is used to convert the binary string into a binary value.

Binary.FromText

What is Binary Data?

Binary data is a type of data that is represented as a sequence of 0s and 1s. It is used to represent many different types of data, such as images, audio files, and computer programs. Binary data is often stored in files as a stream of bytes, where each byte is a sequence of 8 bits (0s and 1s).

How Does Binary.FromText Work?

Binary.FromText is a function in M that takes a text string as input and returns a binary value. The text string must represent binary data in a specific format, which is called a hexadecimal string. A hexadecimal string is a sequence of characters that represents binary data as a series of hexadecimal digits.

For example, the binary data represented by the hexadecimal string “4D 5A” is the first two bytes of a Windows executable file. The Binary.FromText function can convert this string into its corresponding binary representation, which is the byte sequence 4D 5A.

Here is an example of how to use Binary.FromText in Power Query:

```
let
  binaryText = "4D5A",
  binaryValue = Binary.FromText(binaryText)
in
  binaryValue
```

This code creates a variable called binaryText, which contains the hexadecimal string “4D5A”. It then calls the Binary.FromText function with this string as input, and assigns the result to a variable called binaryValue. The result is a binary value that represents the byte sequence 4D 5A.

How to Convert Binary Data to Text

Sometimes you may need to convert binary data into a text string, for example, when you are working with text-based file formats like CSV or XML. You can use the Text.FromBinary function in M to do this.

Here is an example of how to use Text.FromBinary in Power Query:

Binary.InferContentType

What is Binary.InferContentType?

Binary.InferContentType is a function in the M language that is used to infer the content type of a binary value. In other words, it is used to determine the file type of a binary value, such as a document, image, or audio file. This function is particularly useful when working with files that do not have a file extension or when the file extension is incorrect or missing.

The syntax of the Binary.InferContentType function is as follows:

```
Binary.InferContentType(binary as binary) as text
```

The function takes a binary value as input and returns the content type as text. The content type is returned in the form of a MIME type, which is a standard way of identifying the type of a file.

How does Binary.InferContentType work?

The Binary.InferContentType function works by examining the first few bytes of a binary value to determine its content type. Each file type has a unique signature, which is a specific sequence of bytes at the beginning of the file. By examining these bytes, the function can determine the content type of the file.

For example, a JPEG image file always starts with the bytes “FF D8 FF”. By examining the first few bytes of a binary value, the Binary.InferContentType function can determine whether it is a JPEG image file or not.

How to use Binary.InferContentType in Power Query

To use the Binary.InferContentType function in Power Query, you need to create a custom function that calls the Binary.InferContentType function. The custom function takes a binary value as input and returns the content type as text.

Here is an example of a custom function that uses the Binary.InferContentType function to determine the content type of a binary value:

```
let
  GetContentType = (binaryValue as binary) =>
    let
Binary.Length
```

Understanding the Binary.Length Function

The Binary.Length function is a part of the Power Query M language, which is a functional programming language used to manage and transform data. As the name suggests, the Binary.Length function is used to calculate the length of binary values in a data set. It takes a binary value as an input and returns the length of the binary value.

Syntax of Binary.Length Function

The syntax of the Binary.Length function is as follows:

```
Binary.Length(binary as binary) as number
```

The function takes a binary value as an input and returns a number that represents the length of the binary value. Let's take a look at an example to understand the function better.

Suppose we have a data set that contains binary values in the 'BinaryData' column. We can use the Binary.Length function to determine the length of these binary values. The M code for this would be:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WcikqzS9R0lEyNjQwVUjMKy8GzgYjIzC3LzMvJzMxNwA=",
BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [BinaryData =
_t]),
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"BinaryData", type binary}}),
  #"Added Custom" = Table.AddColumn(#"Changed Type", "Binary Length", each Binary.Length([BinaryData]))
in
  #"Added Custom"
```

Binary.Range

What is the Binary.Range Function?

The Binary.Range function is used in Power Query to extract a range of bytes from a binary value. It takes three arguments:

1. The binary value to extract bytes from
2. The starting byte index
3. The number of bytes to extract

The function returns a binary value that consists of the specified range of bytes from the input binary value.

How to Use the Binary.Range Function

To use the Binary.Range function in Power Query, you need to follow these steps:

1. Open Power Query and create a new query.
2. Connect to the data source that contains the binary value you want to extract bytes from.
3. Add a custom column to the data by clicking on the “Add Column” tab and selecting “Custom Column”.
4. In the “Custom Column” dialog box, enter the following formula:

Replace "Binary Column" with the name of the column that contains the binary value, "Start Index" with the index of the first byte you want to extract, and "Number of Bytes" with the number of bytes you want to extract.

5. Click "OK" to create the custom column.

The Binary.Range function can be used in a variety of scenarios. For example, you can use it to extract a specific range of bytes from a binary file, such as an image or a PDF document. You can also use it to extract specific fields from a binary protocol message.

The M Code Behind the Binary.Range Function

To understand how the Binary.Range function works, let's take a look at the M code behind it. In Power Query, M is the formula language used to create queries. The M code for the Binary.Range function is:

```
(BinaryValue as binary, StartIndex as number, NumberOfBytes as number) as binary =>
 let
  EndIndex = StartIndex + NumberOfBytes,
```

Binary.Split

What is Binary.Split?

Binary.Split is a Power Query M function that allows you to split binary data into separate columns. Binary data is data that is represented in binary form, such as a sequence of ones and zeros. Binary data is commonly used in computer systems to represent non-textual data, such as images or sound files.

The Binary.Split function takes two arguments: the binary data to be split, and the delimiter that should be used to split the data. The delimiter is a binary value that is used to separate the data into separate columns.

How to Use Binary.Split

To use the Binary.Split function, you first need to create a query in Power Query. Once you have created a query, you can use the Binary.Split function to split the binary data into separate columns.

Here is an example of how to use the Binary.Split function in Power Query:

```
  let
    binaryData = Binary.FromText("01010100 01100101 01110011 01110100 00100000 01100100 01100001 01110100 01100001"),
    splitData = Binary.Split(binaryData, Binary.FromText(" "))
  in
    splitData
```

In this example, we are using the Binary.FromText function to convert a text string into binary data. We then use the Binary.Split function to split the binary data into separate columns using a space as the delimiter. The resulting output is a table with a single row and eight columns, each containing one byte of binary data.

The M Code Behind Binary.Split

The M code behind the Binary.Split function is relatively simple. Here is the M code that is used to define the Binary.Split function:

```
  (binary as binary, delimiter as binary) as list =>
  let
    bytePositions = List.Positions(
      binary,
      (i) => i = delimiter,
      Occurrence.Last
    ),
```

Binary.ToList

One of the most useful M functions in Power Query is Binary.ToList. This function converts a binary value into a list of integers representing the binary digits. In this article, we will explore the M code behind this function and how it works.

Understanding Binary Values

Before we dive into the M code behind Binary.ToList, we need to understand what binary values are and how they work. Binary values are a sequence of 0s and 1s used to represent numbers, characters, and other data types in computers.

For example, the binary value 1001 represents the decimal number 9. The first 1 represents 2^3 (or 8), and the second 1 represents 2^0 (or 1). Therefore, we add 8 and 1 to get the decimal value 9.

Binary values are often used in computer systems because they are easier to process and store than decimal values. However, they are not as easy to read and understand for humans.

The Binary.ToList Function

The Binary.ToList function in Power Query takes a binary value as input and returns a list of integers representing the binary digits. For example, the binary value 1001 would result in the list [1, 0, 0, 1].

Here is the M code behind Binary.ToList:

```
(BinaryValue as binary) => List.Reverse(List.SkipWhile(List.Reverse(List.Transform(Binary.ToText(BinaryValue), each Number.FromText(_))), (x) => x = 0))
```

Let's break down this M code step by step and see how it works.

Step 1: Converting Binary to Text

The first step in the Binary.ToList function is to convert the binary value to text using the Binary.ToText function. This function takes a binary value as input and returns a text value representing the binary digits.

For example, the binary value 1001 would result in the text value "1001".

Step 2: Converting Text to Numbers

The next step in the Binary.ToList function is to convert each character in the text value to a number using the Number.FromText function. This function takes a text value as input and returns a number value.

Binary.ToText

Understanding the Binary.ToText Function

The Binary.ToText function is used to convert binary data into text format. The function takes an input parameter, which is the binary data to be converted, and an optional parameter, which specifies the encoding to be used for the conversion. The default encoding used is UTF8.

Here is an example of using the Binary.ToText function:

```
let
  binaryData = Binary.FromText(“Hello, world!”),
  textData = Binary.ToText(binaryData)
in
  textData
```

In the above example, we first convert the text "Hello, world!" into binary data using the Binary.FromText function. We then pass this binary data to the Binary.ToText function, which converts it back into text format. The resulting output will be "Hello, world!".

The M Code Behind the Binary.ToText Function

The M code behind the Binary.ToText function is relatively simple. Here is the code:

```
(Binary as binary, optional Encoding as nullable number) as text =>
let
encoding = if Encoding = null then 65001 else Encoding,
text = Text.FromBinary(binary, encoding)
in
text
```

Binary.View

The Binary.View function is used to convert binary data to a text representation. This function is used to view the raw data in a binary format. In this article, we will explore the M code behind the Power Query M function Binary.View.

Understanding Binary Data

Binary data is data that is represented in binary code, which is a series of 1s and 0s. It is a way of representing data using only two digits. Binary data is used in many applications, including computer networking, cryptography, and file formats such as images, audio, and video.

In Power Query, binary data can be represented as a binary type value. This type of value contains binary data that can be converted to text using the Binary.View function.

The M Code Behind Binary.View

The Binary.View function is used to convert binary data to a text representation. It takes a binary type value as its input and returns a text value that represents the binary data in a readable format. The M code behind this function is simple and can be written as follows:

`Binary.View = (binary as binary) as text => Text.FromBinary(binary)`

This code defines the Binary.View function as a lambda function that takes a binary type value as its input and returns a text value that represents the binary data. The Text.FromBinary function is used to convert the binary data to text.

Using Binary.View in Power Query

To use the Binary.View function in Power Query, you need to have binary data in your dataset. You can load binary data into Power Query using the Binary.From function. Once you have loaded your binary data, you can use the Binary.View function to view the raw data in a text format.

Here's an example of how to use the Binary.View function in Power Query:

1. Load your binary data into Power Query using the Binary.From function.

```
let
  Source = Binary.FromText("01010100 01100101 01110011 01110100 00100000 01100111 01100001 01110010 01100010 01100001 01100111 01100101", BinaryEncoding.Base2)
in
  Source
```

Binary.ViewError

Understanding Binary.ViewError

Binary.ViewError is a function in the M language that allows users to view error messages that occur during the execution of M code. The function takes two arguments, the first being the binary data that contains the error message and the second being the optional encoding of the binary data. The function returns a text value that contains the error message.

The primary use case of Binary.ViewError is to debug M code when an error occurs during its execution. When an error occurs, Power Query returns a binary value that contains the error message. The binary data is not human-readable and needs to be converted into a text format to understand the error message.

Binary.ViewError converts the binary data into a human-readable text format, making it easier for users to understand the error message and debug their M code.

The M Code Behind Binary.ViewError

The M code behind Binary.ViewError is relatively simple and involves converting the binary data into a text format. Here is the M code behind Binary.ViewError:

```
(BinaryData as binary, optional Encoding as nullable number) as text =>
  let
    EncodingNumber = if Encoding = null then 1252 else Encoding,
    EncodedText = Text.FromBinary(BinaryData, EncodingNumber),
    DecodedText = try EncodedText otherwise “”,
    ErrorText = if DecodedText = “” then “Binary decoding error” else DecodedText
  in
    ErrorText
```

The function takes two arguments, BinaryData, which is the binary data that contains the error message, and Encoding, which is the optional encoding of the binary data.

The first step in the function is to check if the Encoding argument is null. If it is null, the function sets the EncodingNumber variable to

Binary.ViewFunction

Overview of Binary.ViewFunction

Before diving into the M code behind Binary.ViewFunction, it’s important to understand its purpose and how it works. The Binary.ViewFunction function takes in a binary value as its input and returns the contents of that value as text. This is useful when working with binary files, such as PDFs or images, that cannot be easily interpreted by humans.

The function takes two arguments: value and optional encoding. The value argument is the binary value that you want to view, while the encoding argument specifies the character encoding to use when converting the binary value to text. If the encoding argument is not provided, then the function uses the default encoding of UTF-8.

Here’s an example of how to use the Binary.ViewFunction function:

```
let
  binaryValue = Binary.FromText(“Hello, world!”, TextEncoding.Ascii),
  viewValue = Binary.ViewFunction(binaryValue, TextEncoding.Ascii)
in
  viewValue
```

In this example, we first convert the text "Hello, world!" into a binary value using the Binary.FromText function and the ASCII encoding. We then pass this binary value into the Binary.ViewFunction function along with the ASCII encoding to get the text representation of the binary value.

The M Code Behind Binary.ViewFunction

Now that we understand what the Binary.ViewFunction function does, let's take a look at the M code behind it. The M code for the function is relatively simple and straightforward:

```
(Binary as any, optional Encoding as nullable number) as text =>
  let
BinaryFormat.7BitEncodedSignedInteger
```

Let’s take a closer look at the M code behind this function and how it works.

The BinaryFormat.7BitEncodedSignedInteger function

The BinaryFormat.7BitEncodedSignedInteger function takes an integer as its input and returns a binary value that represents the compressed 7-bit format of that integer.

Here is an example of how to use this function:

```
BinaryFormat.7BitEncodedSignedInteger(100)
```

This would return the binary value `11000100`.

Understanding the M code

The M code behind the BinaryFormat.7BitEncodedSignedInteger function uses a combination of bitwise operators and conditional statements to compress the integer into a 7-bit format.

Here is the M code for the BinaryFormat.7BitEncodedSignedInteger function:

```
(n as number) as binary =>
let
  result = if n >= 0 then
    if n < 128 then n
    else if n < 16384 then
      (n % 128) + 128 + (n / 128)  256
    else if n < 2097152 then
      (n % 128) + 128 + ((n / 128) % 128)  256 + (n / 16384)  65536
    else if n < 268435456 then
      (n % 128) + 128 + ((n / 128) % 128)  256 + ((n / 16384) % 128)  65536 + (n / 2097152)  16777216
```

BinaryFormat.7BitEncodedUnsignedInteger

Understanding the Basics of Binary Encoding

Before we dive into the specifics of the BinaryFormat.7BitEncodedUnsignedInteger function, it's helpful to have a basic understanding of binary encoding. Binary encoding is a method of representing data using only two symbols: 0 and 1. These symbols are used to represent the two possible states of an electronic circuit, and they can be combined to represent any other data.

When we talk about binary encoding, we usually mean that we are representing data as a sequence of bits. A bit is either a 0 or a 1, and multiple bits can be combined to make up a byte. A byte is a group of 8 bits, and it can represent any integer value from 0 to 255.

What is the BinaryFormat.7BitEncodedUnsignedInteger Function?

The BinaryFormat.7BitEncodedUnsignedInteger function in Power Query allows you to convert a series of integers into a compressed binary format that is much more efficient to store and transmit than the original numbers. To use this function, you simply need to provide a list of integers and specify the number of bits to use for each integer.

The resulting binary data is encoded using a combination of 7-bit and 8-bit bytes. Each 7-bit byte is used to represent a single integer value between 0 and 127, while each 8-bit byte is used to represent an integer value between 128 and 255.

How Does the Function Work?

The M code behind the BinaryFormat.7BitEncodedUnsignedInteger function is relatively complex, but it can be broken down into a few basic steps. Here's a high-level overview of how the function works:

1. First, the function converts each input integer into its binary representation. This is done using the Number.ToText function and specifying a base of 2.
2. Next, the function pads each binary representation with leading zeros until it is the specified number of bits long.
3. The function then groups the binary representations into sets of 7 bits each. If the last group has fewer than 7 bits, it is padded with zeros until it is exactly 7 bits long.
4. Each 7-bit group is then converted into an 8-bit byte by adding a leading bit with a value of 1. This is done to indicate that the byte is a 7-bit group rather than a regular 8-bit byte.
5. Finally, all of the 8-bit bytes are concatenated together to form the final compressed binary data.

Advantages of Using the BinaryFormat.7BitEncodedUnsignedInteger Function

The BinaryFormat.7BitEncodedUnsignedInteger function has several advantages over other methods of storing and transmitting integer data. Some of the main advantages include:

– Smaller File Sizes: Because the binary data is compressed, it takes up much less space than the original integer values. This can be a

BinaryFormat.Binary

What is BinaryFormat.Binary?

BinaryFormat.Binary is a Power Query M function that enables users to convert any value to a binary representation. The binary representation is a sequence of 0s and 1s that represent the value's bits. For example, the binary representation of the decimal number 5 is 101. BinaryFormat.Binary takes two arguments: the value to be converted and the number of bits to use in the binary representation.

M Code Behind BinaryFormat.Binary

The M code behind BinaryFormat.Binary is relatively simple. The function first checks whether the value to be converted is null. If the value is null, the function returns null. If the value is not null, the function converts the value to a binary representation.

Here is the M code for BinaryFormat.Binary:

```
(BinaryValue as any, Optional Bits as number) as binary =>
  if BinaryValue = null then
    null
  else if Bits = null or Bits = 0 then
    Binary.FromNumber(BinaryValue, 0)
  else
    Binary.FromNumber(BinaryValue, Bits)
```

The function takes two arguments: BinaryValue and Bits. BinaryValue is the value to be converted, and Bits is the number of bits to use in the binary representation. If Bits is not specified, the function defaults to using the minimum number of bits required to represent the value.

The first line of the function checks whether BinaryValue is null. If BinaryValue is null, the function returns null. This is to ensure that the function does not try to convert a null value to a binary representation.

The second line of the function checks whether Bits is null or zero. If Bits is null or zero, the function returns the binary representation of BinaryValue using the minimum number of bits required.

The third line of the function converts BinaryValue to a binary representation using the specified number of bits.

BinaryFormat.Byte

What is BinaryFormat.Byte?

BinaryFormat.Byte is a Power Query M function that converts a number into a binary representation of a specified length. The function takes two arguments: the number to be converted and the length of the binary representation.

Understanding the M Code Behind BinaryFormat.Byte

The M code behind BinaryFormat.Byte is relatively straightforward. It involves the use of the Number.ToText and Number.Mod functions to perform calculations and obtain the binary representation of the number.

```
(BinLength, Number) =>
 let
  Binary = List.Transform(
   List.Generate(
    () => [Value = Number, Bit = 0],
    each [Bit] < BinLength,
    each [Value = Number  2, Bit = Number.Mod(Number, 2)]),
   each Text.From([Bit])),
  BinaryString = Text.Combine(List.Reverse(Binary), “”)
 in
  Number.FromText(BinaryString, 2)
```

The function takes two parameters, BinLength and Number, and begins by first defining a list of 0's and 1's to represent the binary digits. This is done using the List.Generate function, which generates a list of values based on the initial value and a condition that must be met for each value generated.

The List.Generate function takes three parameters: the initial value, the condition, and the function used to generate the next value. In this case, the initial value is set to [Value = Number, Bit = 0], where Value is the number to be converted and Bit is initially set to 0.

The condition for generating the next value is set to [Bit] < BinLength, which ensures that the function generates values until the

BinaryFormat.ByteOrder

Understanding Byte Order

Before we dive into the M code behind BinaryFormat.ByteOrder, it’s important to understand what byte order is and why it matters. Byte order refers to the way in which a computer stores multi-byte data types (such as integers) in memory. There are two common byte orders: big-endian and little-endian.

In big-endian byte order, the most significant byte (i.e. the byte with the highest value) is stored first. In little-endian byte order, the least significant byte (i.e. the byte with the lowest value) is stored first. The byte order used by a particular computer architecture can have a significant impact on the way data is interpreted and processed.

Using BinaryFormat.ByteOrder

BinaryFormat.ByteOrder is a Power Query M function that can be used to change the byte order of binary data. The function takes two arguments: the binary data to be converted, and a value indicating the desired byte order. The byte order value can be either “BigEndian” or “LittleEndian”.

Here’s an example of how BinaryFormat.ByteOrder can be used to convert binary data from little-endian to big-endian byte order:

```
let
  data = Binary.FromText(“01020304”, BinaryEncoding.Hex),
  bigEndianData = BinaryFormat.ByteOrder(data, “BigEndian”)
in
  bigEndianData
```

In this example, we start with a binary value of "01020304", which represents the integer value 16909060 in little-endian byte order. We use Binary.FromText to convert this value into binary data, and then use BinaryFormat.ByteOrder to convert it to big-endian byte order. The resulting binary data is "04030201", which represents the same integer value in big-endian byte order.

The M Code Behind BinaryFormat.ByteOrder

So, what's the M code behind BinaryFormat.ByteOrder? Let's take a look:

```
BinaryFormat.Choice
```

What is BinaryFormat.Choice?

BinaryFormat.Choice is a function in Power Query that allows you to format binary data. It’s particularly useful when working with data that has been stored in a binary format, such as PDFs, images, or compressed files. With BinaryFormat.Choice, you can extract specific pieces of data from a binary file and format them in a meaningful way.

The Syntax of BinaryFormat.Choice

Before we dive into the M code behind BinaryFormat.Choice, let’s take a quick look at its syntax. Here’s an example of how to use BinaryFormat.Choice:

```
BinaryFormat.Choice(binary, choices)
```

The first argument, 'binary', is the binary data that you want to format. The second argument, 'choices', is a list of options that you want to extract from the binary data. Each option in the list should be a tuple that specifies the format of the option and the number of bytes it occupies.

The M Code Behind BinaryFormat.Choice

Now that we understand the syntax of BinaryFormat.Choice, let's take a closer look at its M code. Here's an example:

```
BinaryFormat.Choice(
  binary,
  {{
    {0, 1},
    {1, 1},
    {2, 4},
    {6, 1},
    {7, 1}
BinaryFormat.Decimal
```

Understanding the BinaryFormat.Decimal Function

The BinaryFormat.Decimal function is a powerful feature in Power Query M that converts a binary number to a decimal number. This function utilizes the IEEE 754 standard for binary floating-point arithmetic to ensure accuracy and precision in the conversion process. The syntax for the BinaryFormat.Decimal function is as follows:

```
BinaryFormat.Decimal(binary as any, optional places as nullable number, optional culture as nullable text) as nullable number
```

Here, the 'binary' parameter represents the binary number that needs to be converted to a decimal number. The 'places' parameter specifies the number of decimal places to be displayed in the output, and the 'culture' parameter specifies the culture to use for formatting the decimal number.

How to Use the BinaryFormat.Decimal Function

To use the BinaryFormat.Decimal function, you first need to have a binary number that needs to be converted to a decimal number. Once you have the binary number, you can use the BinaryFormat.Decimal function to convert it to a decimal number.

For example, let's say you have a binary number '1010'. To convert this binary number to a decimal number using the BinaryFormat.Decimal function, you can use the following M code:

```
BinaryFormat.Decimal(0b1010)
```

In this code, '0b' is used to represent a binary number in the M language. The output of this code will be '10', which is the decimal equivalent of the binary number '1010'.

Benefits of Using the BinaryFormat.Decimal Function

The BinaryFormat.Decimal function has several benefits that make it a popular choice in data analysis and processing. Some of the key benefits of using this function are:

What is a Double-Precision Floating-Point Number?

Before we dive into the M code behind the BinaryFormat.Double function, it's important to understand what a double-precision floating-point number is. In computing, a floating-point number is a representation of a real number that can contain a fractional part. Double-precision floating-point numbers are a specific type of floating-point number that use twice as many bits to represent the number as a single-precision floating-point number.

Double-precision floating-point numbers have a higher precision than single-precision floating-point numbers, which makes them useful for applications where a high degree of accuracy is required. They are also used in scientific and engineering applications where small variations in the values can have a significant impact on the results.

The BinaryFormat.Double Function

The BinaryFormat.Double function is used to convert a binary value into a double-precision floating-point number. The syntax for the function is as follows:

```
BinaryFormat.Double(binary as binary, optional offset as number) as number
```

The function takes two parameters – the binary value that is being converted and an optional offset value. If an offset value is provided, the function will read the binary value starting from that offset, rather than from the beginning of the value.

The M Code Behind the BinaryFormat.Double Function

The M code behind the BinaryFormat.Double function is relatively simple. The function reads the binary value that is being converted and then uses a series of bitwise operations to convert the value into a double-precision floating-point number.

Here is the M code for the BinaryFormat.Double function:

```
let
  BinaryFormat.Double = (binary as binary, optional offset as number) as number =>
  let
BinaryFormat.Group
```

What is BinaryFormat.Group?
BinaryFormat.Group is a function in Power Query M that allows you to group data based on a specific byte size. It works by formatting the data into a binary representation, and then grouping the data by the specified byte size. The function takes two arguments: the data you want to group, and the byte size you want to group it by.
Here is an example of how BinaryFormat.Group works:

```
let
  data = {1, 2, 3, 4, 5, 6, 7, 8},
  groupSize = 2,
  result = BinaryFormat.Group(data, groupSize)
in
  result
```

In this example, we are grouping the data {1, 2, 3, 4, 5, 6, 7, 8} into groups of 2. The resulting binary representation would be:

```
00000001 00000010
00000011 00000100
00000101 00000110
00000111 00001000
```

The resulting data would be:

BinaryFormat.Length

Understanding BinaryFormat.Length

BinaryFormat.Length is a function used with binary values in Power Query M. It returns the length of a binary value in bytes. This function is useful when working with binary data, such as images, audio files, and documents.

The syntax for BinaryFormat.Length is as follows:

```
BinaryFormat.Length(binary as binary) as number
```

The function takes a binary value as its argument and returns the length of the binary value in bytes as a number.

M Code Example

Here's an example that demonstrates the use of BinaryFormat.Length in Power Query M:

```
let
  binaryValue = Binary.FromText(“Hello World”, BinaryEncoding.Base64),
  length = BinaryFormat.Length(binaryValue)
in
  length
```

In this example, we first create a binary value from the text "Hello World" using the Binary.FromText function. We then pass this binary value to the BinaryFormat.Length function, which returns the length of the binary value in bytes. The result is stored in the length variable.

Tips for Using BinaryFormat.Length

Here are some tips for using BinaryFormat.Length in Power Query M:

1. Check for Null Values

BinaryFormat.List

What is BinaryFormat.List?

BinaryFormat.List is a Power Query M function that takes a binary value and converts it into a list of values according to a specified format. The function is useful when you need to extract data from a binary file or a binary column in a table. The format parameter specifies how to interpret the binary data. The following format codes are supported:

– “b”: Byte (1 byte)
– “h”: Short integer (2 bytes)
– “i”: Integer (4 bytes)
– “q”: Long integer (8 bytes)
– “f”: Single-precision floating-point number (4 bytes)
– “d”: Double-precision floating-point number (8 bytes)

How to use BinaryFormat.List

The BinaryFormat.List function takes two parameters: the binary value and the format string. Here’s an example that shows how to use the function:

```
let
   binary = Binary.FromText(“0102030405”),
   list = BinaryFormat.List(binary, “b”)
in
   list
```

In this example, we create a binary value by converting a text value "0102030405" to binary using the Binary.FromText function. Then we call the BinaryFormat.List function with the binary value and the format string "b", which tells the function to interpret the binary data as a sequence of bytes. The result is a list of five values: [1, 2, 3, 4, 5].

Here's another example that shows how to use the function with multiple format codes:

BinaryFormat.Null

What is BinaryFormat.Null?

BinaryFormat.Null is a function in Power Query that represents null values in binary format. It is used to handle null values in binary data sources, such as working with binary files or reading data from a binary database. The function returns a 4-byte null value, represented in hexadecimal format as “00000000”.

Understanding the M Code Behind BinaryFormat.Null

The M code behind BinaryFormat.Null is quite simple. The function is represented as follows:

```
BinaryFormat.Null = () =>
   #binary({0,0,0,0})
```

The function takes no parameters and returns a binary value of four bytes, which represent the null value. This is achieved by calling the #binary function and passing in an array of four zeros.

Using BinaryFormat.Null in Power Query

To use BinaryFormat.Null in Power Query, you can simply call the function in your query. For example, if you are reading data from a binary file and encounter a null value, you can replace it with BinaryFormat.Null as follows:

```
let
   Source = Binary.Buffer(File.Contents(“C:DataBinaryFile.bin”)),
   ReplacedNull = Table.ReplaceValue(Source, null, BinaryFormat.Null, Replacer.ReplaceValue,{“Column1”})
in
   ReplacedNull
```

In this example, the Table.ReplaceValue function is used to replace any null values in "Column1" with BinaryFormat.Null. The result is a BinaryFormat.Record

Understanding BinaryFormat.Record

The BinaryFormat.Record function is used to specify the format of a binary file. This function takes in a list of record fields and their corresponding data types. Each record field defines a piece of data within the binary file. The data types can be one of the following:

– Int8
– Int16
– Int32
– Int64
– UInt8
– UInt16
– UInt32
– UInt64
– Float
– Double
– Decimal
– DateTime
– TimeSpan
– Text
– Binary

Creating a BinaryFormat.Record

To create a BinaryFormat.Record, you will need to define the record fields and their data types. Here is an example:

```
BinaryFormat.Record([
  [Field1 = Int16.Type],
  [Field2 = Text.Type],
  [Field3 = DateTime.Type]
])
```

BinaryFormat.SignedInteger16

Understanding the BinaryFormat.SignedInteger16 function

Before we delve into the M code behind the BinaryFormat.SignedInteger16 function, let us first understand what this function does. The BinaryFormat.SignedInteger16 function is used to convert data types into signed 16-bit integers. This function takes two arguments: the first argument is the data type that needs to be converted, and the second argument is the endianness of the byte order.

The data type argument can be any valid Power Query data type such as text, number, date, or time. The endianness argument is an optional argument that determines the byte order of the output. If this argument is not specified, the function assumes a little-endian byte order.

The M code behind the BinaryFormat.SignedInteger16 function

Now that we understand what the BinaryFormat.SignedInteger16 function does, let us dive into the M code behind this function. The M code for the BinaryFormat.SignedInteger16 function is as follows:

```
let
  BinaryFormat.SignedInteger16 = (value as nullable any, optional endianness as nullable any) as nullable any =>
    let
      binaryValue = BinaryFormat.Integer16(value, endianness),
      signedValue = if binaryValue > 32767 then binaryValue – 65536 else binaryValue
    in
      signedValue
in
  BinaryFormat.SignedInteger16
```

As you can see from the code, the BinaryFormat.SignedInteger16 function is simply a wrapper for the BinaryFormat.Integer16 function. The BinaryFormat.Integer16 function is used to convert the input value into a 16-bit signed integer, and the BinaryFormat.SignedInteger16 function then checks if the value is greater than 32767. If the value is greater than 32767, the function subtracts 65536 from the value to get the correct signed integer value.

BinaryFormat.SignedInteger32

The M code behind the BinaryFormat.SignedInteger32 function is relatively simple. It takes a binary value as input and returns an integer value with a length of 32 bits. Here's the M code for the BinaryFormat.SignedInteger32 function:

```
let
  BinaryToInt32 = (binary) =>
    let
      binaryAsList = Binary.ToList(binary),
      paddedBinary = List.Combine({List.Repeat({0}, 32 – List.Count(binaryAsList)), binaryAsList}),
      sign = Number.FromBinary(List.First(paddedBinary)),
      magnitude = List.Accumulate(
        List.Skip(paddedBinary, 1),
        0,
        (state, current) => state  2 + Number.FromBinary(current)
      )
    in
      if sign = 0 then magnitude else magnitude – 2147483648
in
  BinaryToInt32
```

Let's break down this code for a better understanding:
- The BinaryFormat.SignedInteger32 function takes one parameter, which is the binary data to be converted to an integer.
- The Binary.ToList function converts the binary data into a list of 1's and 0's.
- The List.Repeat function adds padding to the binary data so that it has a total of 32 bits.
- The List.Combine function combines the original binary data and the padded bits into a single list.
- The Number.FromBinary function converts the first bit of the binary data to a number, which represents the sign of the integer.

BinaryFormat.SignedInteger64

In this article, we will explore the M code behind the BinaryFormat.SignedInteger64 function and provide examples of how it can be used in Power Query.

Understanding the BinaryFormat.SignedInteger64 function

The BinaryFormat.SignedInteger64 function is used to convert a binary value to a signed 64-bit integer value or vice versa. The syntax for this function is as follows:

```
BinaryFormat.SignedInteger64(binary as binary, optional endian as nullable number) as number
```

The binary parameter is a binary value that represents a signed 64-bit integer. The optional endian parameter specifies the byte order of the binary value. If no endian value is specified, the function assumes a little-endian byte order.

The function returns a signed 64-bit integer value.

Example 1: Converting binary data to a signed 64-bit integer

Suppose we have the following binary data:

```
11010011 01100111 10111100 00001111 00000000 00000000 00000000 00000001
```

We can use the BinaryFormat.SignedInteger64 function to convert this binary value to a signed 64-bit integer as follows:

```
let
  binaryData = Binary.FromText("11010011011001111011110000001111000000000000000000000000000000001",
BinaryEncoding.Base2),
  signedInt = BinaryFormat.SignedInteger64(binaryData)
```

BinaryFormat.Single

What is the BinaryFormat.Single function?

The BinaryFormat.Single function is used to convert a binary value into a single-precision floating-point number. The function takes a binary value as input and returns a floating-point number. The function is commonly used when working with data types that are not supported by Power Query, such as binary data.

Understanding the M code behind the BinaryFormat.Single function

The M code behind the BinaryFormat.Single function is fairly simple. The function takes a binary value as input and converts it into a single-precision floating-point number. Here is the M code for the BinaryFormat.Single function:

```
(BinaryValue as binary) =>
let
  SingleValue = BinaryFormat.ByteOrder(BinaryValue, ByteOrder.LittleEndian),
  Result = Number.FromBinary(SingleValue, 1),
in
  Result
```

As you can see, the function takes a binary value as input and then performs two operations on it. First, it uses the BinaryFormat.ByteOrder function to specify the byte order of the binary value. The byte order can be either LittleEndian or BigEndian. In LittleEndian byte order, the least significant byte of the binary value is stored first, while the most significant byte is stored last. In BigEndian byte order, the most significant byte is stored first, while the least significant byte is stored last.

After specifying the byte order, the function uses the Number.FromBinary function to convert the binary value into a floating-point number. The Number.FromBinary function takes two arguments: the binary value to convert and the number of bytes to use for the conversion. In this case, we are using a single byte to convert the binary value into a floating-point number.

Using the BinaryFormat.Single function in Power Query

Now that we understand the M code behind the BinaryFormat.Single function, let's see how we can use it in Power Query.

To use the BinaryFormat.Single function, we first need to have a binary value. Let's say we have a table that contains a column called BinaryFormat.Text

What is BinaryFormat.Text?

BinaryFormat.Text is an M function that converts binary data into a string of text. This can be useful when working with data that is stored in a binary format, such as images, PDFs, or other files. By converting the binary data into text, you can more easily read and manipulate the data in your Power Query transformations.

How to Use BinaryFormat.Text

To use BinaryFormat.Text, you first need to understand the parameters that the function accepts. The function has two parameters: the binary data that you want to convert, and the encoding type that you want to use for the conversion.

The binary data can be any binary data source that you have loaded into your Power Query project. This can include data from a file, a web page, or a database. The encoding type determines how the binary data is converted into text. Common encoding types include ASCII, Unicode, and UTF-8.

Once you have the binary data and encoding type, you can use the BinaryFormat.Text function to convert the data into text. Here is an example of how to use the function:

```
let
  binaryData = File.Contents(“C:MyFile.pdf”),
  textData = BinaryFormat.Text(binaryData, Encoding.Ascii)
in
  textData
```

In this example, we are loading a PDF file (stored as binary data) and converting it into text using the ASCII encoding type. The resulting text can then be used for further Power Query transformations.

Understanding the M Code

Now that we understand how to use BinaryFormat.Text, let's take a closer look at the M code behind the function. The function is defined as follows:

```
BinaryFormat.Transform
```

In this article, we will explore the M code behind the BinaryFormat.Transform function and how it can be used to transform binary data in Power Query.

What is Binary Data?

Binary data is a type of data that is represented in a binary format, which consists of only two digits, 0 and 1. It is commonly used in computer systems to represent data such as images, videos, audio files, and executable files. Binary data is often difficult to read and interpret, which is why it is necessary to convert it into a more readable format.

The BinaryFormat.Transform Function

The BinaryFormat.Transform function is a powerful function in Power Query M that allows you to convert binary data into different formats such as text, date, time, and number. It takes two arguments, the first argument is the binary data that you want to transform, and the second argument is the format that you want to transform the binary data into.

Here is the syntax of the BinaryFormat.Transform function:

```
BinaryFormat.Transform(binaryData as binary, format as text) as any
```

The first argument, binaryData, is the binary data that you want to transform. It must be provided as a binary value.

The second argument, format, is the format that you want to transform the binary data into. It must be provided as a text value. The format argument can be any of the following:

- "text": Converts the binary data into a text string.
- "date": Converts the binary data into a date.
- "time": Converts the binary data into a time.
- "datetime": Converts the binary data into a datetime.
- "datetimezone": Converts the binary data into a datetimezone.
- "duration": Converts the binary data into a duration.
- "number": Converts the binary data into a number.

How to Use BinaryFormat.Transform Function

BinaryFormat.UnsignedInteger16

What is the BinaryFormat.UnsignedInteger16 function?

The BinaryFormat.UnsignedInteger16 function is part of the Power Query M language. It is used to convert binary data into a 16-bit unsigned integer value. This function takes a binary value as input and returns a 16-bit integer value. The function is useful for working with binary data that represents numbers, such as in network protocols or file formats.

How does the BinaryFormat.UnsignedInteger16 function work?

The BinaryFormat.UnsignedInteger16 function works by taking a binary value as input and converting it into a 16-bit unsigned integer value. The function uses the little-endian byte order, which means that the least significant byte is stored first in memory.

To understand how the function works, let’s take a closer look at the M code behind it. The M code for the BinaryFormat.UnsignedInteger16 function is as follows:

```
BinaryFormat.UnsignedInteger16(binary as binary) as number =>
 let
    value = BinaryFormat.Record(ByteOrder.LittleEndian, {{“value”, BinaryFormat.UnsignedInteger16}}, binary),
    result = value[value]
 in
    result
```

The function takes a binary value as input, which is stored in the binary parameter. The function then uses the BinaryFormat.Record function to create a record that contains the binary data. The record is created with the ByteOrder.LittleEndian option, which specifies that the least significant byte is stored first. The record contains a single field called "value", which is of type BinaryFormat.UnsignedInteger16.

The BinaryFormat.UnsignedInteger16 function is then called on the "value" field of the record to convert the binary data into a 16-bit unsigned integer value. The result of the function is stored in the result variable and returned as the output of the function.

Examples of using the BinaryFormat.UnsignedInteger16 function

Let's take a look at some examples of using the BinaryFormat.UnsignedInteger16 function.

BinaryFormat.UnsignedInteger32

What is BinaryFormat.UnsignedInteger32?

BinaryFormat.UnsignedInteger32 is a function that is used to convert binary data, represented as a byte array, into 32-bit unsigned integers. The function takes two arguments: the binary data and the byte order. The byte order specifies whether the most significant byte or the least significant byte comes first in the byte array.

The M Code Behind BinaryFormat.UnsignedInteger32

To understand the M code behind BinaryFormat.UnsignedInteger32, let's take a look at an example. Consider the following byte array: `{0x01, 0x02, 0x03, 0x04}`

This byte array represents the unsigned 32-bit integer 0x04030201 in little-endian byte order. To convert this byte array into an unsigned 32-bit integer using the BinaryFormat.UnsignedInteger32 function, we can use the following M code:

```
let
  binary = {0x01, 0x02, 0x03, 0x04},
  uint32 = BinaryFormat.UnsignedInteger32(binary, ByteOrder.LittleEndian)
in
  uint32
```

Let's break down this code. First, we define a variable `binary` that contains the byte array `{0x01, 0x02, 0x03, 0x04}`. Then, we call the BinaryFormat.UnsignedInteger32 function and pass in the `binary` variable as well as the ByteOrder.LittleEndian value to specify the byte order. Finally, we assign the result to a variable `uint32`.

Byte Order

Byte order is an important consideration when working with binary data. It specifies whether the most significant byte or the least significant byte comes first in the byte array. There are two byte orders: big-endian and little-endian.

In big-endian byte order, the most significant byte comes first in the byte array. In little-endian byte order, the least significant byte comes first in the byte array. Most modern computers use little-endian byte order, but some older computers and network protocols use big-endian byte order.

BinaryFormat.UnsignedInteger64

What is BinaryFormat.UnsignedInteger64?

BinaryFormat.UnsignedInteger64 is a Power Query M function that takes a number as its input and returns an unsigned 64-bit binary string representation of that number. This function is particularly useful for working with binary data, such as when dealing with binary file formats or when performing bitwise operations.

The function takes two optional arguments: the first argument specifies the number of bits to use (default is 64), while the second argument specifies whether to use little-endian or big-endian byte order (default is false, meaning big-endian).

Here is an example of how to use the BinaryFormat.UnsignedInteger64 function:

```
BinaryFormat.UnsignedInteger64(1234567890)
```

This will return the binary string "00000000 00000000 00000000 00000000 01001010 01100110 10000001 00110010".

The M code behind BinaryFormat.UnsignedInteger64

The M code behind BinaryFormat.UnsignedInteger64 is actually quite simple. Here is the complete function definition:

```
(BinaryFormat.UnsignedInteger64) =>
let
  binary = Number.ToText(_, “################0”),
  padded = Text.PadStart(binary, 64, “0”),
  grouped = List.Transform({0..7}, each Text.Combine(List.Reverse(List.Range(Text.ToList(padded), _ 8, 8)), “”)),
  result = Text.Combine(grouped, ” “)
in
  result
```

Byte.From

What is Byte.From?

Byte.From is a Power Query M function that converts a binary value to a text value. It takes a binary value as input and returns a text value. The syntax of the function is as follows:

```
Byte.From(binary as binary) as text
```

The binary parameter is the binary value that you want to convert to text. It can be a single value or a list of values.

Understanding the M Code

The M code behind the Byte.From function is relatively simple. It first checks if the input value is null. If it is, it returns null. If it is not null, it converts the binary value to text using the Text.From function. The Text.From function converts a value of any type to text. Here is the M code for the Byte.From function:

```
let
  Byte.From = (binary as binary) as text =>
    if binary = null then null else Text.From(binary)
in
  Byte.From
```

Using Byte.From

Byte.From can be used in a variety of ways to manipulate and transform data. Here are some examples:

Example 1: Converting Binary to Text

Suppose you have a binary value that you want to convert to text. You can use the Byte.From function to do this. Here is an example:

```
Cdm.Contents
```

In this article, we'll explore the M code behind the Cdm.Contents function and look at how it works.

What is Common Data Model (CDM)?

Before we dive into the M code behind Cdm.Contents, let's first understand what Common Data Model (CDM) is.

CDM is a standardized, extensible, and semantic data model that is used to simplify the process of integrating data from various sources. It defines entities, attributes, relationships, and semantic metadata for data integration. CDM also provides a consistent and well-documented way of representing data, making it easier to share and integrate data across different systems.

CDM is used by various Microsoft services like Power Apps, Power BI, Dynamics 365, and Azure Data Factory.

Understanding the M Code Behind Cdm.Contents

The Cdm.Contents function is used to connect to a CDM folder and retrieve data from it. Let's break down the M code behind this function and understand how it works.

```
Cdm.Contents(path as text, optional options as nullable record) as table
```

The Cdm.Contents function takes two arguments:

1. path: This is the path of the CDM folder that you want to connect to. It is a required argument and should be specified as a text string.
2. options: This is an optional argument that allows you to specify additional options for the connection. It should be specified as a record.

Now let's take a closer look at the Cdm.Contents function.

Step 1: Establish a Connection

The first step in the Cdm.Contents function is to establish a connection to the CDM folder specified in the path argument. This is done using the following code:

```
let
  source = Cdm.Contents(path, options)
```

Character.FromNumber

What is Power Query M?

Power Query M is a functional programming language used in Power Query, a data transformation and analysis tool available in Excel and Power BI. M is designed to work with data structures like tables, lists, and records, and can be used to transform and manipulate data in a variety of ways.

Introducing the Character.FromNumber Function

The Character.FromNumber function is one of the many functions available in Power Query M. It allows you to convert a number to a character based on its Unicode value. The function takes a single argument, the number to be converted, and returns the corresponding character.

Here is an example of how the Character.FromNumber function can be used to convert the Unicode value 65 to the character “A”:

```
Character.FromNumber(65)
```

The function returns the character "A".

How the Function Works

The M code behind the Character.FromNumber function is relatively simple. Here is the code:

```
(characterCode as number) as text => Character.FromUnicodeNumber(characterCode)
```

Let's break down this code. The function takes a single argument called characterCode, which is a number. This number represents the Unicode value of the character to be returned.

The function then calls the Character.FromUnicodeNumber function, passing in the characterCode argument. This function returns the character corresponding to the Unicode value.

Using the Function in Data Transformation

Character.ToNumber

Understanding the Character.ToNumber Function

The Character.ToNumber function is used to convert a character value into a numeric value. It takes a single parameter, which is the character to be converted. The function then returns the numeric value of the character. For example, if the character “4” is passed to the function, it will return the number 4.

This function is useful when working with data that is stored as text and needs to be converted to a numeric value. It can also be used in conjunction with other Power Query M functions to perform more complex transformations.

The M Code Behind the Character.ToNumber Function

The M code behind the Character.ToNumber function is relatively simple. It consists of a single line of code that uses the Text.ToNumber function to convert the character to a numeric value. Here is the M code:

```
(Text.ToNumber(Text.From()))
```

The Text.From function is used to convert the character value to a text value, while the Text.ToNumber function is used to convert the text value to a numeric value. The result is then returned as the output of the function.

Using the Character.ToNumber Function

The Character.ToNumber function can be used in various ways to improve data transformation. Here are some examples:

Converting a Column from Text to Numeric

Suppose you have a column in a dataset that contains numeric values stored as text. You can use the Character.ToNumber function to convert the text values to numeric values. Here is the M code:

```
= Table.TransformColumns(
,{{,Character.ToNumber}})
```

Combiner.CombineTextByDelimiter

Understanding the Combiner.CombineTextByDelimiter Function

The Combiner.CombineTextByDelimiter function is used to combine multiple columns of text into a single column separated by a delimiter. The function takes three arguments: the delimiter, the columns to combine, and an optional parameter to specify the output column name.

Here is an example of how the function works:

```
= Table.AddColumn(#”PreviousStep”, “Combined”, each Combiner.CombineTextByDelimiter(“, “, {“Column1”, “Column2”, “Column3”}))
```

In this example, we are adding a new column called "Combined" to a table called "PreviousStep". The new column will combine the values from columns "Column1", "Column2", and "Column3" separated by a comma and a space.

The M Code Behind Combiner.CombineTextByDelimiter

The M code behind the Combiner.CombineTextByDelimiter function is straightforward and easy to understand. Here is the M code for the function:

```
let
  CombineTextByDelimiter = (delimiter as text, columns as list, optional separator as text) as text =>
    let
      separator = if separator = null then “” else separator,
      columnValues = List.Transform(columns, each Text.From(_)),
      result = Text.Combine(columnValues, delimiter & separator)
    in
      result
in
Combiner.CombineTextByEachDelimiter
```

In this article, we will delve into the M code behind this powerful function, explore its capabilities, and provide some practical examples of how it can be used to manipulate data.

Overview of the Combiner.CombineTextByEachDelimiter Function

The Combiner.CombineTextByEachDelimiter function is a Power Query function that allows you to combine text values in a column, separated by one or more delimiters of your choice. This function is particularly useful when you have data that is not formatted in a way that makes it easy to analyze.

The syntax for the Combiner.CombineTextByEachDelimiter function is as follows:

```
Combiner.CombineTextByEachDelimiter(delimiters as list, optional quoteStyle as nullable number) as function
```

The `delimiters` parameter is a list of one or more characters that you want to use as delimiters to separate the text values in the column. The `quoteStyle` parameter is optional and allows you to specify the type of quotes to use when combining the text values.

Here is an example of how you can use the Combiner.CombineTextByEachDelimiter function to combine text values in a column separated by a comma:

```
let
  Source = Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("...", BinaryEncoding.Base64)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Column1 = _t, Column2 = _t])),
  CombinedText = Table.AddColumn(Source, "Combined Text", each Combiner.CombineTextByEachDelimiter({","})(List.RemoveNulls({[Column1], [Column2]})))
in
  CombinedText
```

Combiner.CombineTextByLengths

In this article, we will explore the details of the Combiner.CombineTextByLengths function in Power Query M, including its syntax, usage, and examples.

Syntax

The syntax for the Combiner.CombineTextByLengths function is as follows:

```
Combiner.CombineTextByLengths(textColumns as list, lengthRange as list, optional separator as nullable text) as text
```

The parameters for the function are as follows:

- textColumns: A list of column names containing the text data to be combined.
- lengthRange: A list of integer values representing the minimum and maximum length of the text to be included in the combined column.
- separator (optional): A text value that will be used to separate the combined text values.

Usage

The Combiner.CombineTextByLengths function is used to combine multiple columns of text data into a single column, while filtering the text values based on their length. This function is particularly useful in situations where you have multiple columns of text data that you want to combine, but only want to include certain portions of the text.

The function takes two lists of parameters as input: a list of column names containing the text data to be combined, and a list of integer values representing the minimum and maximum length of the text to be included in the combined column.

For example, if you had three columns of text data called "First Name", "Last Name", and "Address", you could use the Combiner.CombineTextByLengths function to combine the data from these columns into a single column, while only including portions of the text that meet a specific length criteria.

Examples

Here are some examples of how to use the Combiner.CombineTextByLengths function in Power Query M:

Example 1: Basic Usage

Suppose you have a table called "Sales" with the following columns:

Combiner.CombineTextByPositions

Overview of Combiner.CombineTextByPositions

The Combiner.CombineTextByPositions function allows you to combine text values from different columns by specifying the positions of the columns. For example, if you have three columns (Column1, Column2, and Column3), you can use the function to combine the values in Column1 and Column3 into a single column, while ignoring the values in Column2.

The function takes three arguments:

– The table to operate on

– A list of positions (in the form of integers) that specify the columns to combine

– An optional delimiter to use between the combined values (default is a comma)

Here is the basic syntax of the function:

```
Combiner.CombineTextByPositions(table as table, positions as list, optional delimiter as text)
```

How the Function Works

To understand how the Combiner.CombineTextByPositions function works, let's take a look at the M code behind it.

```
let
  CombineTextByPositions = (table as table, positions as list, optional delimiter as text) =>
  let
    columns = Table.ColumnNames(table),
    selectedColumns = List.Select(columns, each List.Contains(positions, columns.IndexOf(_))),
    combinedColumnName = “Combined”,
    combinedColumn = Table.AddColumn(table, combinedColumnName, each
Text.Combine(List.Transform(List.Select(selectedColumns, each not List.IsEmpty(_)), each Text.From(_)), delimiter), type text),
    removedColumns = List.RemoveFirstN(List.RemoveItems(columns, selectedColumns), 1),
```

Combiner.CombineTextByRanges

What is Combiner.CombineTextByRanges?

The Combiner.CombineTextByRanges function is a Power Query M function that allows you to combine text values by specifying a range of characters to include in the combined text. This function is useful when you need to merge text values from multiple columns or rows into a single text value.

How does Combiner.CombineTextByRanges work?

The Combiner.CombineTextByRanges function takes three arguments:

How to use Combiner.CombineTextByRanges?

1. In Power Query, select the table you want to transform.

```
= Combiner.CombineTextByRanges({[First Name], [Last Name]}, {{0, Text.Length([First Name])}, {0, Text.Length([Last Name])}}, “, “)
```

Comparer.FromCulture

The Comparer.FromCulture function is used to compare and sort text values in Power Query M based on a specific culture. This function is useful when dealing with text values that are in different languages or when sorting data based on a specific cultural ordering. In this article, we will take a closer look at the M code behind the Comparer.FromCulture function and how it works.

Understanding the Comparer.FromCulture Function

The Comparer.FromCulture function is part of the Text library in Power Query M and is used to create a comparer that compares two values based on a specific culture. The function takes a single argument, which is the culture to be used for comparison.

The syntax for the Comparer.FromCulture function is as follows:

```
Comparer.FromCulture(culture as text) as function
```

The culture parameter is a text value that specifies the culture to be used for comparison. This value can be any valid culture name or language identifier, such as en-US or fr-FR.

The function returns a comparer function that can be used to compare two values based on the specified culture. The comparer function takes two arguments, which are the values to be compared, and returns a result based on the comparison.

Using the Comparer.FromCulture Function in Power Query M

The Comparer.FromCulture function is used in Power Query M to compare and sort text values based on a specific culture. This function is useful when dealing with text values that are in different languages or when sorting data based on a specific cultural ordering.

To use the Comparer.FromCulture function in Power Query M, you need to first create a comparer function by calling the function with the desired culture. This comparer function can then be used in various functions that require a comparer, such as List.Sort or List.Max. For example, consider the following Power Query M code that sorts a list of text values in French:

```
let
  Source = {“Bonjour”, “Ça va?”, “Comment ça va?”, “Au revoir”, “À bientôt”},
  Sorted = List.Sort(Source, Comparer.FromCulture(“fr-FR”))
```

Comparer.Ordinal

In this article, we will dive into the M code behind the Comparer.Ordinal function and explain how it works.

Understanding the Basics of Comparer.Ordinal

Before we delve into the M code behind the Comparer.Ordinal function, let's first understand what it does. The Comparer.Ordinal function is used to compare two strings in a case-sensitive manner. This means that upper and lower case letters are treated differently by the function. For example, the string "Hello" is not equal to "hello" when compared using the Comparer.Ordinal function.

The Comparer.Ordinal function takes two arguments: the first argument is the first string to be compared, and the second argument is the second string to be compared. The function returns a number that indicates the result of the comparison. If the first string is less than the second string, the function returns a negative number. If the first string is equal to the second string, the function returns zero. If the first string is greater than the second string, the function returns a positive number.

The M Code Behind Comparer.Ordinal

The M code behind the Comparer.Ordinal function is actually quite simple. Here is the code for the function:

```
(text1 as text, text2 as text) =>
  let
    len1 = Text.Length(text1),
    len2 = Text.Length(text2),
    len = if len1 < len2 then len1 else len2,
    cmp = List.PositionOf(
      List.Transform(
        {0..len-1},
        each if Text.At(text1, _) = Text.At(text2, _) then 0 else 1
      ),
      1
    )
  in
    if cmp <> -1 then cmp else len1 – len2
```

Comparer.OrdinalIgnoreCase

Understanding the Comparer.OrdinalIgnoreCase Function in Power Query M

Before diving into the M code behind the Comparer.OrdinalIgnoreCase function, it is essential to understand what the function does. The Comparer.OrdinalIgnoreCase function is used to compare two strings in a case-insensitive manner. When sorting strings, the function sorts them based on their Unicode code points. However, it ignores any differences in casing between uppercase and lowercase characters.

Here is an example of how the Comparer.OrdinalIgnoreCase function works:

```
let
  Source = {“Apple”, “banana”, “cherry”, “berry”},
  Sorted = List.Sort(Source, Comparer.OrdinalIgnoreCase)
in
  Sorted
```

The output of this M code would be:

{“Apple”, “banana”, “berry”, “cherry”}

As you can see, the strings are sorted in alphabetical order, but the function ignores the casing of the characters.

The M Code Behind the Comparer.OrdinalIgnoreCase Function

The M code behind the Comparer.OrdinalIgnoreCase function is relatively straightforward. Here is the M code for the function:

```
Comparer.OrdinalIgnoreCase =
```

Controlling byte order

What is Byte Order?

Before we dive into the M code behind controlling byte order, it is essential to understand what byte order means. Byte order refers to the order in which bytes are arranged to represent data. There are two common byte order formats: Little-Endian and Big-Endian.

In the Little-Endian format, the least significant byte is stored first, followed by the next least significant byte, and so on until the most significant byte is stored last. In contrast, the Big-Endian format stores the most significant byte first, followed by the next most significant byte, and so on until the least significant byte is stored last.

Byte order is essential when working with binary data, such as network packets or file formats. If byte order is not correctly considered, data can be misinterpreted, resulting in errors or incorrect results.

Controlling Byte Order in Power Query

Power Query is a powerful tool for working with data, and its M function provides a simple way of controlling byte order when handling binary data. The M function can be used to read data in both Little-Endian and Big-Endian formats and convert between them.

The M function provides the following functions for controlling byte order:

ByteOrder.LittleEndian

The ByteOrder.LittleEndian function specifies that data should be read in Little-Endian format.

ByteOrder.BigEndian

The ByteOrder.BigEndian function specifies that data should be read in Big-Endian format.

Binary.ToText

The Binary.ToText function can be used to convert binary data to text and specify the byte order format. By default, Binary.ToText assumes Little-Endian format, but this can be changed by specifying the ByteOrder parameter.

Binary.FromText

The Binary.FromText function can be used to convert text to binary data and specify the byte order format. By default, Binary.FromText assumes Little-Endian format, but this can be changed by specifying the ByteOrder parameter.

Byte order is an important consideration when working with binary data, and Power Query's M function provides a simple and effective way of controlling it. By using the ByteOrder.LittleEndian and ByteOrder.BigEndian functions, along with the Binary.ToText and Binary.FromText functions, you can ensure that data is read and written in the correct byte order format.

In summary, the M code behind the Power Query M function provides a straightforward way of controlling byte order when working with data. By understanding byte order and how to use the M function, you can ensure that your data is accurately processed and avoid

Csv.Document

In this article, we will dive into the M code behind the Csv.Document function. We will explore how it works, what options it has, and how you can use it to optimize your data transformation workflows.

What is Csv.Document?

Csv.Document is a Power Query M function that reads a CSV file and returns a table. It takes a file path or a binary as input and produces a table as output. The table contains the data in the CSV file, with the first row as the header and the rest of the rows as data.

The Csv.Document function has a number of options that allow you to customize the way it reads the CSV file. These options include:

– Delimiter: Specifies the delimiter used in the CSV file. By default, Csv.Document uses a comma as the delimiter, but you can change it to any character you want.

– Encoding: Specifies the encoding used in the CSV file. By default, Csv.Document uses UTF-8 encoding, but you can change it to any encoding supported by Power Query.

– Header: Specifies whether the CSV file has a header row or not. By default, Csv.Document assumes that the CSV file has a header row.

– QuoteStyle: Specifies the quote style used in the CSV file. By default, Csv.Document assumes that the CSV file uses double quotes to enclose field values.

– Culture: Specifies the culture used to parse numeric and date/time values in the CSV file. By default, Csv.Document uses the current culture.

How does Csv.Document work?

Let’s take a closer look at how Csv.Document works. Here is an example of how to use Csv.Document to read a CSV file:

```
let
  Source = Csv.Document(File.Contents(“C:UsersJohnDoeDocumentsdata.csv”), [Delimiter=”,”, Encoding=1252,
QuoteStyle=QuoteStyle.None, Header=1]),
  #”Changed Type” = Table.TransformColumnTypes(Source,{{“Column1”, type text}, {“Column2”, Int64.Type}})
in
  #”Changed Type”
```

Cube.AddAndExpandDimensionColumn

One of Power Query's M functions is Cube.AddAndExpandDimensionColumn. This function adds a new column to a table and expands it into multiple columns, creating new rows for each combination of other columns in the table.
Syntax of Cube.AddAndExpandDimensionColumn Function
The syntax of Cube.AddAndExpandDimensionColumn function is as follows:

Cube.AddAndExpandDimensionColumn(table as table, column as text, values as list, newColumnNames as list, optional missingValue as nullable any) as table

The parameters are:
- table: A table or a reference to a table.
- column: The name of the column to expand.
- values: A list of values to expand into new columns.
- newColumnNames: A list of column names for the new expanded columns.
- missingValue (optional): The value to replace any missing values with.

How Cube.AddAndExpandDimensionColumn Function Works
The Cube.AddAndExpandDimensionColumn function works by adding a new column to the table, then expanding it into multiple columns, creating new rows for each combination of other columns in the table.
For example, consider the following table:

Table.InlineTable({{1, “A”, 10}, {1, “B”, 20}, {2, “A”, 30}, {2, “B”, 40}})

This table has three columns: "Column1", "Column2", and "Column3". We can use the Cube.AddAndExpandDimensionColumn function to add a new column "Column4" with values "X" and "Y", and expand it into two new columns "Column5" and "Column6":

Cube.AddMeasureColumn

Understanding Cube Measures

Before we dive into the M code behind Cube.AddMeasureColumn, it's important to understand what a cube measure is. In a cube, a measure is a calculation that is performed on a set of data. For example, you may have a sales cube that includes measures such as total sales, average sales, and maximum sales. These measures can be used to analyze the data in different ways, such as by region, product line, or salesperson.

When you add a measure to a cube, you can use it to create a calculated column in Power Query. This is where the Cube.AddMeasureColumn function comes in. It allows you to create a calculated column based on a measure in your cube.

The M Code Behind Cube.AddMeasureColumn

To use Cube.AddMeasureColumn, you need to understand the M code behind it. The function takes four arguments:

1. Cube: This is the name of the cube that contains the measure you want to use. For example, if your sales data is stored in a cube called "Sales Data," you would use "Sales Data" as the cube argument.
2. MeasureName: This is the name of the measure you want to use. For example, if you want to create a calculated column based on the total sales measure, you would use "Total Sales" as the MeasureName argument.
3. ColumnPrefix: This is the prefix that will be added to the name of the new calculated column. For example, if you use "Sales" as the ColumnPrefix argument, the new column will be called "Sales Total Sales."
4. ColumnSuffix: This is the suffix that will be added to the name of the new calculated column. For example, if you use "Amount" as the ColumnSuffix argument, the new column will be called "Total Sales Amount."

The M code for Cube.AddMeasureColumn looks like this:

```
Cube.AddMeasureColumn = (Cube as text, MeasureName as text, ColumnPrefix as text, ColumnSuffix as text) =>
let
  Source = Cube.CubeFunctions(Cube, [Query = "MDSCHEMA_MEASURES"]),
  FilteredRows = Table.SelectRows(Source, each ([CUBE_NAME] = Cube and [MEASURE_NAME] = MeasureName)),
  MeasureFormula = if Table.IsEmpty(FilteredRows) then "" else FilteredRows{0}[MEASURE_EXPRESSION],
  ColumnName = ColumnPrefix & " " & MeasureName & " " & ColumnSuffix,
  NewColumn = if MeasureFormula = "" then null else Table.AddColumn(Source, ColumnName, each
Cube.ApplyParameter
```

Understanding Cube.ApplyParameter

Cube.ApplyParameter is a powerful M function that allows users to pass parameters to a query and apply them to a cube. The function takes two arguments: the cube expression and a record that contains the parameter values. The cube expression is a valid M expression that returns a cube, while the parameter record is a record that contains the parameter values.

The Cube.ApplyParameter function is used to parameterize a query that references a cube. The function enables users to pass parameter values to a cube and apply them to the query. This allows users to create dynamic reports that can be updated based on the parameter values.

How to Use Cube.ApplyParameter

To use Cube.ApplyParameter, you need to create a parameter record that contains the parameter values. The parameter record should be created using the #shared keyword, which is used to define shared values in Power Query. Once you have created the parameter record, you can pass it to the Cube.ApplyParameter function along with the cube expression.

Here is an example of how to use Cube.ApplyParameter:

```
let
  parameterRecord = #shared [
    Parameter1 = “Value1”,
    Parameter2 = “Value2”
  ],
  cubeExpression = Cube.CubeExpression(“CubeName”),
  parameterizedQuery = Cube.ApplyParameter(
    cubeExpression,
    parameterRecord
  )
in
  parameterizedQuery
```

Cube.AttributeMemberId

One such function is the Cube.AttributeMemberId function. Let’s take a closer look at the M code behind this function and how it can be used to work with multidimensional data.

What is Cube.AttributeMemberId Function?

The Cube.AttributeMemberId function is used to retrieve a unique identifier for a member of a specified attribute hierarchy. This function requires two arguments: the first argument is the name of the attribute hierarchy, and the second argument is the member name.

Here’s the syntax of the Cube.AttributeMemberId function:

```
Cube.AttributeMemberId(attributeHierarchyName, memberName)
```

The function returns a unique identifier for the specified member. This unique identifier is useful when you need to reference a specific member in a measure or calculation.

How to Use Cube.AttributeMemberId Function

Let's take a look at an example of how to use the Cube.AttributeMemberId function.

Suppose we have a multidimensional data source that contains a Sales cube with a Date dimension and a Product dimension. The Product dimension has an attribute hierarchy called Product Category, which has the following members:

- Beverages
- Dairy
- Snacks
- Baked Goods

Suppose we want to retrieve the Sales data for the Snacks category only. We can use the Cube.AttributeMemberId function to retrieve the unique identifier for the Snacks member as follows:

```
Cube.AttributeMemberId(“Product Category”, “Snacks”)
```

Cube.AttributeMemberProperty

In this article, we'll take a deep dive into the M code behind Cube.AttributeMemberProperty. We'll explore the different parameters you can pass to the function and how they affect the output. We'll also look at some examples of how to use this function in practice. So, let's get started.

Understanding the Syntax of Cube.AttributeMemberProperty

Before we dive into the M code behind Cube.AttributeMemberProperty, let's take a moment to understand the syntax of the function. Here's the basic syntax of Cube.AttributeMemberProperty:

```
Cube.AttributeMemberProperty(connection as text, cube as text, measureOrDimension as text, attribute as text, member as text, property as text, optional culture as nullable text) as any
```

As you can see, the function takes several parameters:

- connection: The name of the Power BI data source or connection.
- cube: The name of the cube that contains the measure or dimension you want to retrieve.
- measureOrDimension: The name of the measure or dimension you want to retrieve the member attribute from.
- attribute: The name of the member attribute you want to retrieve.
- member: The name of the member you want to retrieve the property for.
- property: The name of the property you want to retrieve, such as "Name", "Caption", or "UniqueName".
- culture: An optional parameter that specifies the culture to use when retrieving the property. If not specified, the default culture of the current user is used.

Exploring the M Code Behind Cube.AttributeMemberProperty

Now that we understand the basic syntax of Cube.AttributeMemberProperty, let's take a look at the M code behind the function. Here's the code:

```
let
Cube.CollapseAndRemoveColumns
```

What is Cube.CollapseAndRemoveColumns?

Cube.CollapseAndRemoveColumns is a Power Query M function that allows you to collapse multiple columns into a single column. This function is particularly useful when working with large datasets that contain several similar columns, such as multiple columns containing sales data for different regions or months. By collapsing these columns, you can simplify your data and make it easier to work with.

In addition to collapsing columns, Cube.CollapseAndRemoveColumns also allows you to remove unwanted columns from your dataset. This can be useful when you have columns that contain redundant or unnecessary information.

How to use Cube.CollapseAndRemoveColumns

To use Cube.CollapseAndRemoveColumns, you'll need to create a new query in Power Query. Once you've created your query, you can use the following steps to collapse and remove columns:

1. First, select the columns that you want to collapse. You can select multiple columns by holding down the Ctrl key while clicking on each column header.
2. Right-click on one of the selected columns and choose "Transform" from the context menu.
3. Choose "Group By" from the Transform menu.
4. In the Group By dialog box, select the column that you want to collapse into, then click the "Advanced" button.
5. In the Advanced options, choose "All Rows" for the Operation field, and enter a new column name for the New column name field.
6. To remove unwanted columns, select the columns you want to remove from the column headers, right-click and choose "Remove".
7. Click "Close & Load" to apply your changes and load the new data into your workbook.

The M Code Behind Cube.CollapseAndRemoveColumns

If you're interested in the technical details behind Cube.CollapseAndRemoveColumns, you can take a look at the M code that powers this function. The M code behind Cube.CollapseAndRemoveColumns is as follows:

```
let
  Source = Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMlTSUXJz0qzOzy/KSVEwNlRqsQA=", BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Column1 = _t, Column2 = _t, Column3 = _t]),
```

Cube.Dimensions

What is the Cube.Dimensions Function?

The Cube.Dimensions function is used to retrieve the dimensions of a cube from a given connection. It takes a single argument, which is the connection to the cube. The function returns a table that contains information about the dimensions in the cube.

How Does it Work?

The Cube.Dimensions function works by sending an MDX query to the cube to retrieve information about the dimensions. The MDX query is constructed using the M language, which is the language used by Power Query.

The M code behind the Cube.Dimensions function is relatively simple. It starts by creating a new connection to the cube using the given connection string. Once the connection is established, the function sends an MDX query to the cube to retrieve information about the dimensions.

The MDX query used by the Cube.Dimensions function is as follows:

```
SELECT {[DIMENSION_NAME]} ON COLUMNS FROM [CUBE_NAME];
```

In this query, [DIMENSION_NAME] is the name of the dimension that you want to retrieve information about, and [CUBE_NAME] is the name of the cube that you want to retrieve the information from.

The query returns a table that contains information about the dimension. This information includes the name of the dimension, the number of hierarchies in the dimension, and the number of members in each hierarchy.

Example Usage

Here is an example of how you can use the Cube.Dimensions function to retrieve information about the dimensions in a cube:

```
let
  Source = AnalysisServices.Database(“localhost”, “Adventure Works DW”),
  Dimensions = Cube.Dimensions(Source{[Name=”Adventure Works DW”]}[Data])
in
Cube.DisplayFolders
```

Understanding Cube.DisplayFolders

Before we dive into the M code behind Cube.DisplayFolders, let’s first take a look at what this function does. Cube.DisplayFolders is a function that is used to organize columns in a PivotTable or a Power View report.

When you create a PivotTable or a Power View report, you may have a lot of columns in your data. However, not all of these columns are relevant or important for your report. This is where Cube.DisplayFolders comes in. This function allows you to group columns into folders, making it easier to find the columns you need.

The M Code Behind Cube.DisplayFolders

Now that we understand what Cube.DisplayFolders does, let’s take a look at the M code behind this function. When you create a PivotTable or a Power View report in Excel, you can use the Cube.DisplayFolders function to organize your data. Here is the M code that you can use to do this:

```
Cube.DisplayFolders = [
   Folder1 = {“Column1”, “Column2”, “Column3”},
   Folder2 = {“Column4”, “Column5”, “Column6”}
]
```

In the above code, we can see that Cube.DisplayFolders is a list of folders. Each folder is represented by a name and a list of columns. For example, let's say we have a dataset with six columns: Column1, Column2, Column3, Column4, Column5, and Column6. In the above code, we have created two folders: Folder1 and Folder2. Folder1 contains Column1, Column2, and Column3, while Folder2 contains Column4, Column5, and Column6.

Using Cube.DisplayFolders in Power Query

Now that we know how to use Cube.DisplayFolders in Excel, let's take a look at how we can use this function in Power Query.

In Power Query, Cube.DisplayFolders is called Cube.ColumnGroupings. Here is the M code that you can use to create column groups in Power Query:

Cube.MeasureProperties

What is Power Query?

Before we dive into the M code behind Cube.MeasureProperties, let’s first define what Power Query is and what it does. Power Query is a data transformation and cleansing tool that is part of the Power BI suite. It allows you to connect to various data sources, transform the data, and load it into a data model or visualization tool like Power BI.

Power Query is a versatile tool and offers a wide range of transformation functions, including splitting columns, merging tables, filtering rows, and aggregating data. One of the key features of Power Query is its ability to work with multidimensional cubes, which is where the Cube.MeasureProperties function comes in.

What is Cube.MeasureProperties?

Cube.MeasureProperties is a Power Query M function that allows you to retrieve the properties of a measure in a multidimensional cube. Measures are calculations in a cube that aggregate data, such as summing or averaging sales by product or region.

Cube.MeasureProperties allows you to retrieve information such as the measure name, format, and description, which is useful when building reports or analyzing data in Power BI.

The syntax for Cube.MeasureProperties is as follows:

```
Cube.MeasureProperties(cube as cube, measureNameOrExpression as text)
```

The function takes two arguments:

- cube: A multidimensional cube.
- measureNameOrExpression: The name of the measure or an expression that evaluates to the measure.

The function returns a record containing the properties of the measure.

The M code behind Cube.MeasureProperties

Now that we've covered what Cube.MeasureProperties does, let's take a look at the M code behind it. The M code for Cube.MeasureProperties is as follows:

```
Cube.MeasureProperty
```

What is Cube.MeasureProperty?

Cube.MeasureProperty is a Power Query M function that is used to retrieve the value of a measure property. A measure property is metadata associated with a measure that provides additional information about that measure. The Cube.MeasureProperty function takes two arguments:

1. The name of the measure whose property value is to be retrieved
2. The name of the property to be retrieved

The function returns the value of the specified measure property.

The M Code Behind Cube.MeasureProperty

The M code behind the Cube.MeasureProperty function is relatively simple. Here is an example of the M code for the Cube.MeasureProperty function:

```
let
   MeasureName = “Total Sales”,
   PropertyName = “Display Folder”,
   Cube = Cube(“Adventure Works”),
   Measure = Cube[Measures]{MeasureName},
   PropertyValue = Measure[PropertyName]
in
   PropertyValue
```

In this code, we first define the variables MeasureName and PropertyName to store the names of the measure and property, respectively. We then create a new variable called Cube and set it to the name of the cube that contains the measure. In this example, we are using the Adventure Works cube.

Next, we use the MeasureName variable to retrieve the specified measure from the cube. We do this by accessing the Measures property of the cube and then using the MeasureName variable to retrieve the specified measure.

```
Cube.Measures
```

What is Cube.Measures?

Cube.Measures is a Power Query M function that allows users to create measures in a cube or tabular model. Measures are calculations that are used to summarize and analyze data in a pivot table or chart. They can be used to perform a variety of tasks, including calculating totals, averages, percentages, and ratios.

Measures are created using the DAX language, which is a formula language used in Power Pivot, Power BI, and Analysis Services. However, Cube.Measures allows users to create measures using M code instead of DAX. This makes it easier for users who are more familiar with M code to create measures and perform complex data analysis.

How to Use Cube.Measures

To use Cube.Measures, you'll need to have a basic understanding of Power Query and M code. Once you have this, you can follow these steps:

1. Open Power Query and load your data into the query editor.
2. Click on the Home tab and select "New Source" > "Blank Query".
3. In the formula bar, type "Cube.Measures" followed by a set of parentheses.
4. Inside the parentheses, enter the name of your measure, followed by a comma.
5. Next, enter the M code that defines your measure.
6. Press enter to create your measure.

The M Code Behind Cube.Measures

Cube.Measures uses M code to create measures that can be used in a pivot table or chart. The M code defines the calculation that the measure will perform. Here's an example of M code that calculates the total sales for each region:

```
Cube.Measures("Total Sales", each List.Sum([Sales]))
```

Let's break down this code into its individual components:

- "Total Sales" is the name of the measure.
- "each List.Sum([Sales])" is the M code that defines the calculation. This code uses the List.Sum function to add up the values in the

Cube.Parameters

What is Cube.Parameters?

Cube.Parameters is a Power Query M function that allows users to create dynamic queries based on user input. This function is particularly useful when dealing with large datasets where it is not feasible to load all the data at once. By using Cube.Parameters, users can filter the data based on user input, which allows them to work with smaller datasets that are easier to manage.

How Does Cube.Parameters Work?

Cube.Parameters works by creating a parameter that can be used to filter data. This parameter can be any valid M expression, such as a text string or a numerical value. When the user runs the query, they are prompted to enter a value for the parameter. This value is then used to filter the data based on the specified criteria.

The M Code Behind Cube.Parameters

To understand how Cube.Parameters works, it is important to understand the M code that powers it. The following code demonstrates how to create a dynamic query using Cube.Parameters:

```
let
  Parameter1 = Cube.Parameters(“Parameter1”, type text),
  Source = Excel.CurrentWorkbook(){[Name=”Table1"]}[Content],
  FilteredRows = Table.SelectRows(Source, each [Column1] = Parameter1)
in
  FilteredRows
```

This code creates a parameter called "Parameter1" of type text. The source data is then loaded into the query, and the Table.SelectRows function is used to filter the data based on the value of Parameter1. When the query is run, the user is prompted to enter a value for Parameter1. This value is then used to filter the data based on the specified criteria.

Using Cube.Parameters in Practice

To demonstrate how Cube.Parameters can be used in practice, let's consider an example where we have a large dataset containing sales data for a company. We want to create a dynamic query that allows us to filter the data based on a specific product category.

Cube.Properties

Cube.Properties is a powerful function that allows you to retrieve metadata about a cube or a tabular model. It can be used to retrieve information about the structure of the model, the data sources, and the relationships between tables. In this article, we will explore the M code behind the Cube.Properties function and show you how to use it to retrieve useful information about your data model.

What is the Cube.Properties Function?

Before we dive into the M code behind the Cube.Properties function, let's first understand what it does. The Cube.Properties function is used to retrieve metadata about a cube or a tabular model. It takes two arguments: the server name and the cube name. Here is the syntax of the Cube.Properties function:

```
Cube.Properties(server as text, cube as text) as record
```

The function returns a record that contains various properties of the cube or tabular model. These properties include the name of the cube, the data sources used in the model, the tables and columns in the model, and the relationships between the tables.

Understanding the M Code behind Cube.Properties

Now that we know what the Cube.Properties function does, let's take a look at the M code behind it. The Cube.Properties function is actually a combination of several other M functions that work together to retrieve the metadata about the cube or tabular model.

Here are the M functions that make up the Cube.Properties function:

- `AnalysisServices.Database`: This function connects to the Analysis Services database and retrieves the metadata about the cube or tabular model.
- `AnalysisServices.Cube`: This function retrieves the metadata about the cube or tabular model.
- `AnalysisServices.DataSources`: This function retrieves the data sources used in the cube or tabular model.
- `AnalysisServices.Tables`: This function retrieves the tables in the cube or tabular model.
- `AnalysisServices.Columns`: This function retrieves the columns in the cube or tabular model.
- `AnalysisServices.Relationships`: This function retrieves the relationships between the tables in the cube or tabular model.

Each of these functions is used to retrieve a specific type of metadata about the cube or tabular model. The Cube.Properties function combines the results of these functions into a single record that contains all of the metadata.

Cube.PropertyKey

Introduction to Cube.PropertyKey

Cube.PropertyKey is a function in Power Query that returns the property key for a given property name in a cube. It takes in two arguments:

– cube: The name of the cube that you want to query

– propertyName: The name of the property for which you want to retrieve the key

This function is often used in combination with other functions like CubeValue and CubeSetProperty to retrieve or set cube property values.

Understanding the M Code Behind Cube.PropertyKey

To understand the M code behind Cube.PropertyKey, let’s start by looking at a simple example. Suppose we have a cube named “Sales” and we want to retrieve the property key for the “Category” property. We can use the following M code:

```
let
  cube = AnalysisServices.Database(“localhost”, “Sales”),
  propertyName = “Category”,
  propertyKey = Cube.PropertyKey(cube, propertyName)
in
  propertyKey
```

Let's break down this code step by step:

1. The first line of code creates a variable named "cube" and assigns it the value of the AnalysisServices.Database function with two arguments: "localhost" (the name of the server where the cube is stored) and "Sales" (the name of the cube).
2. The second line of code creates a variable named "propertyName" and assigns it the value of the string "Category" (the name of the property we want to retrieve the key for).
3. The third line of code creates a variable named "propertyKey" and assigns it the value of the Cube.PropertyKey function with two arguments: "cube" (the name of the cube we want to query) and "propertyName" (the name of the property for which we want to retrieve

Cube.ReplaceDimensions

Understanding Cube.ReplaceDimensions Function

Cube.ReplaceDimensions is a Power Query M function that is used to replace dimensions in a cube. It takes in three arguments: Cube, OldDimension, and NewDimension. The Cube argument specifies the cube that needs to be transformed. The OldDimension argument specifies the dimension that needs to be replaced, while the NewDimension argument specifies the replacement dimension.

The syntax for the Cube.ReplaceDimensions function is as follows:

```
Cube.ReplaceDimensions(Cube as any, OldDimension as any, NewDimension as any) as any
```

Here, the 'as any' keyword is used to specify that the arguments can take any data type.

The M Code Behind Cube.ReplaceDimensions

To understand the M code behind Cube.ReplaceDimensions, we need to understand how it works. Cube.ReplaceDimensions function replaces a dimension in a cube with a new dimension. It does this by creating a new cube that has the new dimension and then copying the data from the old cube to the new cube.

The M code behind Cube.ReplaceDimensions function can be broken down into the following steps:

1. Get the CubeData: The CubeData is the data that needs to be transformed. It is obtained by calling the Cube.GetCubeData function.
2. Get the OldDimensionData: The OldDimensionData is the data that needs to be replaced. It is obtained by calling the Cube.GetDimensionData function.
3. Get the NewDimensionData: The NewDimensionData is the replacement data. It can be obtained from a table or by calling another function that returns a table.
4. Replace the OldDimensionData with the NewDimensionData: This is done by using the Table.ReplaceMatchingRows function to replace the OldDimensionData with the NewDimensionData.
5. Create a new CubeData: The new CubeData is created by using the Cube.ReplaceDimensionData function. This function takes in the old CubeData and the new dimension data and creates a new CubeData.
6. Return the new CubeData: The new CubeData is returned as the output of the Cube.ReplaceDimensions function.

Example

Cube.Transform

But what is the M code behind Cube.Transform, and how does it work? In this article, we'll explore the inner workings of Cube.Transform and how you can use it to streamline your data analysis workflows.

Understanding the M Code Behind Cube.Transform

To understand the M code behind Cube.Transform, it's important to first understand what Cube.Transform does. At a high level, Cube.Transform allows you to connect to a multidimensional database, such as an OLAP cube, and import data into Power Query. Once the data is imported, you can use the M language to manipulate and analyze it.

The M code behind Cube.Transform is relatively complex and involves a number of different functions and operations. At its core, however, Cube.Transform is essentially a wrapper function that allows you to connect to a multidimensional database and import data into Power Query.

Here's a basic example of the M code behind Cube.Transform:

```
let
  Source = Cube.Transform("Data Source=MyServer;Initial Catalog=MyDatabase", "MyCube"),
  #"MyCube_Measures" = Source{[Schema="Measures"]}[Data],
  #"MyCube_Dimensions" = Source{[Schema="Dimensions"]}[Data]
in
  #"MyCube_Measures"
```

This M code connects to a multidimensional database located on MyServer and imports data from the "MyCube" cube. It then extracts the measures from the cube and returns them as a table.

Advanced M Code Techniques for Cube.Transform

While the basic M code for Cube.Transform is relatively simple, there are a number of advanced techniques you can use to take your data analysis workflows to the next level.

Here are some tips and tricks for working with the M code behind Cube.Transform:

Using Filters to Extract Specific Data

Currency.From

In this article, we will take a closer look at the M code behind the Power Query M function Currency.From. We will examine how this function works, what parameters it accepts, and how it can be used to convert currency values from one currency to another.

What is the Currency.From Function?

The Currency.From function is a built-in M function in Power Query that allows users to convert currency values from one currency to another. It takes two parameters: the amount to be converted, and the currency code of the original currency.

Here is the basic syntax of the Currency.From function:

```
Currency.From(amount as any, fromCurrency as text) as nullable number
```

The first parameter, amount, can be any value that can be converted to a number, such as a numeric value, a date/time value, or a string value. The second parameter, fromCurrency, is a text value that represents the currency code of the original currency.

For example, if you want to convert $100 USD to Euros, you would use the following formula:

```
Currency.From(100, “USD”)
```

This would return the equivalent value in Euros based on the current exchange rate.

How Does the Currency.From Function Work?

The Currency.From function works by using the currency exchange rates provided by a data source. By default, it uses the exchange rates provided by the European Central Bank (ECB), which are updated daily. However, users can also specify their own exchange rate data source if they have access to one.

The Currency.From function takes the amount to be converted and the currency code of the original currency as input. It then looks up the exchange rate for the original currency against the target currency and uses that exchange rate to calculate the equivalent value in the target currency.

Date.AddDays

Let’s take a closer look at the M code behind this function and how it works.
The Syntax of Date.AddDays
The syntax for Date.AddDays is as follows:

Date.AddDays(startDate as date, days as number) as date

`startDate` is the starting date to which you want to add days.
`days` is the number of days you want to add to the starting date.
The function returns a new date that is `days` days after the `startDate`.
An Example of Date.AddDays
Suppose you have a table named `SalesData` that contains a column named `OrderDate` with dates in the format of `yyyy-mm-dd`.
You want to add 90 days to each date in the `OrderDate` column and create a new column named `NewOrderDate`.
Here's the M code to achieve this:

```
let
  Source = SalesData,
  #”Added Custom” = Table.AddColumn(Source, “NewOrderDate”, each Date.AddDays([OrderDate], 90)),
  #”Changed Type” = Table.TransformColumnTypes(#”Added Custom”,{{“NewOrderDate”, type date}})
in
  #”Changed Type”
```

The above code adds a custom column named `NewOrderDate` to the `SalesData` table. The `each` keyword is used to apply the Date.AddDays function to each value in the `OrderDate` column. The resulting table has a new column named `NewOrderDate` with

Date.AddMonths

Syntax
The syntax of the Date.AddMonths function is as follows:

```
Date.AddMonths (start_date as date, months as number) as date
```

The function takes two arguments:
- `start_date`: A date value, which is the date from which the months will be added or subtracted.
- `months`: A number value, which is the number of months to be added or subtracted. A positive value adds the months, while a negative value subtracts them.

The function returns a new date value that is the result of adding or subtracting the specified number of months to the start date.
Example
Let's take an example to understand the usage of the Date.AddMonths function. Consider a dataset that contains a column "Order Date" with values representing the date of an order. We want to create a new column "Due Date," which is the date that is three months ahead of the order date.
To achieve this, we can use the following M code:

```
#”Added Custom” = Table.AddColumn(#”PreviousStep”, “Due Date”, each Date.AddMonths([Order Date], 3), type date)
```

In the above code, we are adding a custom column to our dataset using the Table.AddColumn function. The "each" keyword is used to apply the Date.AddMonths function to each row of the dataset. The first argument of the Date.AddMonths function is the [Order Date] column, which represents the start date, and the second argument is the number 3, which represents the number of months to add. Finally, we specify the data type of the new column as "date".
Features

Date.AddQuarters

Understanding the Date.AddQuarters Function

The Date.AddQuarters function is used to add a specified number of quarters to a given date. The syntax for the function is as follows:

```
Date.AddQuarters(dateTime as any, numberOfQuarters as number) as any
```

The function takes two arguments. The first argument, dateTime, is the date to which the quarters will be added. The second argument, numberOfQuarters, is the number of quarters to be added to the date.

For example, if we wanted to add two quarters to the date 1/1/2021, we would use the following formula:

```
Date.AddQuarters(#”1/1/2021", 2)
```

This would result in the date 7/1/2021.

The M Code Behind the Date.AddQuarters Function

Behind the scenes, the Date.AddQuarters function is powered by M code. M code is the language used by Power Query to perform data transformations. When we use the Date.AddQuarters function, Power Query generates M code that performs the necessary calculations.

The M code for the Date.AddQuarters function is as follows:

```
() => (dateTime as any, numberOfQuarters as number) =>
DateTime.LocalNow() + #duration(0, (numberOfQuarters  3), 0, 0)
```

Date.AddWeeks

The `Date.AddWeeks` function is a part of the M language, which is used in Power Query. The M language is a functional programming language that is used to build custom functions and expressions in Power Query.
In this article, we will explore the M code behind the `Date.AddWeeks` function and how it can be used to manipulate dates in Power Query.
Syntax of the Date.AddWeeks Function
The `Date.AddWeeks` function takes two arguments: the date to which you want to add or subtract weeks, and the number of weeks to add or subtract. The syntax of the `Date.AddWeeks` function is as follows:

```
Date.AddWeeks(date as any, weeks as number) as any
```

The first argument, `date`, can be any valid date format, including a date, datetime, or datetimezone. The second argument, `weeks`, is the number of weeks to add or subtract from the `date` argument. The function returns the resulting date after adding or subtracting the specified number of weeks.
Examples of the Date.AddWeeks Function
Let us consider some examples to understand how the `Date.AddWeeks` function works.
Example 1:
Suppose we have a date, 01/01/2022, and we want to add 2 weeks to it. The M code to achieve this is:

```
Date.AddWeeks(#date(2022,1,1), 2)
```

The resulting date will be 15/01/2022.
Example 2:
Suppose we have a datetime, 01/01/2022 12:00:00 AM, and we want to subtract 3 weeks from it. The M code to achieve this is:

Date.AddYears

Date.AddYears Function

The Date.AddYears function is used to add or subtract a specified number of years from a given date. Its syntax is as follows:

```
Date.AddYears(dateTime as any, numberOfYears as number) as any
```

The function takes two arguments: dateTime, which is the date to which the years are added or subtracted, and numberOfYears, which is the number of years to be added or subtracted. The function returns a new date value after the years have been added or subtracted.

M Code Behind Date.AddYears

Behind the scenes, the Date.AddYears function is implemented in the M language using a combination of other functions. The following M code is equivalent to the Date.AddYears function:

```
let
  AddYears = (dateTime as any, numberOfYears as number) =>
    Date.AddDays(Date.From(dateTime), numberOfYears  365),
  result = AddYears(dateTime, numberOfYears)
in
  result
```

The code defines a new function called AddYears, which takes the same two arguments as the Date.AddYears function. The function converts the dateTime value to a date using the Date.From function, which extracts the date portion of the dateTime value. It then multiplies the numberOfYears value by 365 to get the number of days to add or subtract. This value is then added to the date using the Date.AddDays function, which returns a new date value with the specified number of days added or subtracted. The result of the AddYears function is then returned as the result of the M code.

Date.Day

Introduction to Power Query M Language

Power Query is built on a functional language called Power Query M. M is a case-sensitive language that supports functions, operators, and variables. It is used to create custom functions, create transformations, and interact with data sources. M code is written in a formula bar and can be edited using the Advanced Editor. Understanding M language is essential to perform complex data transformations.

Date.Day Function in Power Query

The Date.Day function is used to extract the day of the month from a date value. It is a part of the Date/Time functions available in Power Query. The syntax for the Date.Day function is:

```
Date.Day(dateTime as any) as any
```

where dateTime is the input date value in datetime format, and the output is the day of the month as an integer.

M Code Behind the Date.Day Function

The M code behind the Date.Day function is straightforward. It uses the DateTime.LocalNow function to get the current date and time value. Then, it uses the Date.Day function to extract the day of the month from the date value. Here is the M code for the Date.Day function:

```
let
  currentDateTime = DateTime.LocalNow(),
  currentDay = Date.Day(currentDateTime)
in
  currentDay
```

Date.DayOfWeek

Understanding the Date.DayOfWeek Function

Before we dive into the M code behind the `Date.DayOfWeek` function, let's first understand how this function works. The `Date.DayOfWeek` function takes a date value as its input and returns a number between 0 and 6, representing the day of the week:

– 0: Sunday
– 1: Monday
– 2: Tuesday
– 3: Wednesday
– 4: Thursday
– 5: Friday
– 6: Saturday

For example, if you pass the date value "2022-01-01" to the `Date.DayOfWeek` function, it will return 5, since January 1, 2022 is a Saturday.

Exploring the M Code Behind Date.DayOfWeek

Now that we understand the basic functionality of the `Date.DayOfWeek` function, let's take a closer look at the M code behind it. You can view the M code for any Power Query function by clicking the "Advanced Editor" button in the "View" tab of the Power Query Editor. Here is the M code for the `Date.DayOfWeek` function:

```
(Date as any) as nullable number =>
let
  DayOfWeek = Date.DayOfWeek(Date.DayOfYear(Date)),
  Result = if DayOfWeek = 0 then 7 else DayOfWeek
in
  Result
```

Let's break down this code line by line.

Date.DayOfWeekName

What is the Date.DayOfWeekName Function?

The Date.DayOfWeekName function is a Power Query M function that returns the name of the day of the week for a given date. It is a simple function that takes a date as an argument and returns a string value representing the name of the day of the week. For example, if you provide the date 1/1/2021 as an argument, the function will return “Friday” as the output.

How Does the Function Work?

The M code behind the Date.DayOfWeekName function is relatively simple. It uses the List.Dates function to generate a list of dates for a given period and then applies the Date.DayOfWeek function to each date in the list to get the day of the week. Finally, it uses the List.Transform function to convert the numeric day of the week value into a string value representing the name of the day of the week.

Here is the M code behind the Date.DayOfWeekName function:

```
(Date as date) =>
let
  dayNumber = Date.DayOfWeek(Date),
  dayNames = {“Sunday”, “Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”, “Saturday”},
  dayName = dayNames{dayNumber}
in
  dayName
```

Let's break down this code step by step:

1. The function takes a date as an argument.
2. The DayOfWeek function is used to get the numeric value representing the day of the week for the given date.
3. The dayNames variable is defined as an array of strings representing the names of the days of the week.
4. The dayName variable is defined as the value of the element in the dayNames array corresponding to the numeric day of the week value.

How Can You Use the Date.DayOfWeekName Function?

Date.DayOfYear

Understanding the Date.DayOfYear Function

The Date.DayOfYear function is a member of the Date class in Power Query M. The function takes a single argument, which is a date value. The function then returns the day of the year for that date value. For example, if the date is January 1, the function returns 1. If the date is December 31, the function returns 365 (or 366 in a leap year).

The syntax for the Date.DayOfYear function is as follows:

```
Date.DayOfYear(date as date) as number
```

The M Code Behind the Function

The M code behind the Date.DayOfYear function is relatively simple. The function simply takes the date value and subtracts the first day of the year (January 1) from it. This gives the number of days that have elapsed since the beginning of the year. The code for the Date.DayOfYear function is as follows:

```
let
  Date.DayOfYear = (date as date) as number =>
    Date.Day(date) – Date.Day(Date.FromText(“1/1/” & Number.ToText(Date.Year(date)), “M/d/yyyy”)) + 1
in
  Date.DayOfYear
```

Let's break down this code into its component parts:

- The "let" statement is used to define a function in M. The function is named Date.DayOfYear.
- The function takes a single argument, which is a date value.
- The function returns a number value, which is the day of the year for the given date value.

Date.DaysInMonth

What is the Date.DaysInMonth Function?

The Date.DaysInMonth function is used to calculate the number of days in a month for a given date. This can be useful for a variety of tasks, such as calculating the number of days between two dates or determining the number of working days in a month.

The function takes two arguments: a date, and an optional culture. The culture argument is used to specify the culture to be used for the calculation. If the culture argument is not specified, the default culture for the system is used.

The M Code Behind Date.DaysInMonth

Behind the scenes, the Date.DaysInMonth function is actually implemented in M code. The M code for the function is as follows:

```
(Date as date, optional Culture as nullable text) as number =>
Date.EndOfMonth(Date.StartOfMonth(Date, Culture), Culture) – Date.StartOfMonth(Date, Culture) + 1
```

Let's break down the code and see how it works.

The Date.EndOfMonth Function

The first part of the code uses the Date.EndOfMonth function to determine the last day of the month for the given date. The Date.EndOfMonth function takes a date and an optional culture as arguments, and returns the last day of the month for the given date.

The Date.StartOfMonth Function

The second part of the code uses the Date.StartOfMonth function to determine the first day of the month for the given date. The Date.StartOfMonth function takes a date and an optional culture as arguments, and returns the first day of the month for the given date.

Calculating the Number of Days

Once we have the first and last days of the month, we can calculate the number of days in the month. To do this, we simply subtract the first day of the month from the last day of the month and add 1.

In this article, we took a closer look at the M code behind the Power Query M function Date.DaysInMonth. We saw how the function is implemented using the Date.EndOfMonth and Date.StartOfMonth functions to calculate the number of days in a month for a given date. Understanding the M code behind functions like Date.DaysInMonth can be extremely helpful when working with Power Query, as it allows you to create more complex transformations and queries.

Date.EndOfDay

Understanding the Date.EndOfDay Function

The Date.EndOfDay function is used to find the last moment of the day, which is 23:59:59.9999999. It is used to convert a given date and time to the end of the day. For instance, if you have a date and time of January 1, 2021, 5:30 PM, the Date.EndOfDay function will convert it to January 1, 2021, 11:59:59.9999999 PM.

The M Code Behind Date.EndOfDay Function

The M code behind the Date.EndOfDay function is fairly simple. It involves using the DateTime.LocalNow function to get the current date and time, and then using the DateTime.Date function to get the date part of the current date and time. Finally, the DateTime.From function is used to combine the date part with the maximum time value of the day, which is 23:59:59.9999999.

The M code for the Date.EndOfDay function is as follows:

```
(DateTime.From(
  DateTime.Date(
    DateTime.LocalNow()
  )
)
+ #duration(0, 23, 59, 59, 9999999)
)
```

Let's break down the code into its individual parts and understand how it works.

DateTime.LocalNow

The DateTime.LocalNow function is used to get the current date and time in the local time zone. It returns a datetime value that represents the current date and time in the format "yyyy-MM-ddThh:mm:ss.fffffff".

DateTime.Date

The DateTime.Date function is used to extract the date part from the datetime value. It takes a datetime value as input and returns a date value that represents the date portion of the input datetime.

Date.EndOfMonth

The Date.EndOfMonth Function

The Date.EndOfMonth function in Power Query returns the last day of the month for a given date value. The syntax for this function is as follows:

```
Date.EndOfMonth(dateTime as any) as any
```

The dateTime argument is a date and time value that you want to find the end of the month for. The function returns the end of the month as a date and time value.

The M Code Behind the Function

To understand how the Date.EndOfMonth function works in Power Query, we need to take a look at the M code behind it. The M code is a functional programming language used in Power Query to manipulate data. Here is the M code for the Date.EndOfMonth function:

```
(dateTime as any) => Date.EndOfMonth(dateTime)
```

The code is in the form of a lambda function, which is a function without a name. The argument dateTime is passed to the function as any data type, which allows for any data type to be passed as an argument. The function then calls the built-in Date.EndOfMonth function and passes the dateTime argument to it. The result of the function is the end of the month as a date and time value.

Example Usage

To demonstrate how the Date.EndOfMonth function can be used in Power Query, consider the following example. Let's say we have a table of sales data that includes a column of dates. We want to add a column that shows the last day of the month for each date in the table.

1. Open Power Query and load the sales data into it.
2. Select the Date column and click on the Add Column tab.

Date.EndOfQuarter

Understanding the Date.EndOfQuarter Function

The Date.EndOfQuarter function in Power Query is used to return the last day of the quarter that a given date falls in. For example, if you have a date of January 15th, 2022, the Date.EndOfQuarter function would return March 31st, 2022, as this is the last day of the first quarter of the year.

To use the Date.EndOfQuarter function, you simply need to pass a date value to it. For example, if you have a column named “Date” in your data table, you could create a new column named “End of Quarter” with the following formula:

```
= Table.AddColumn(#”PreviousStep”, “End of Quarter”, each Date.EndOfQuarter([Date]))
```

This would create a new column in your data table that contains the last day of the quarter for each date in the "Date" column.

The M Code Behind Date.EndOfQuarter

Now that we understand what the Date.EndOfQuarter function does, let's take a closer look at the M code behind it. The function is actually quite simple, and consists of just a few lines of code:

```
let
  EndOfQuarter = (date) => Date.EndOfQuarter(date),
  #”End of Quarter” = EndOfQuarter
in
  #”End of Quarter”
```

Let's break down each of these lines of code to better understand what's happening:

1. The first line of code creates a new variable named "EndOfQuarter" and assigns it an anonymous function. This function takes a single argument, "date", and simply calls the built-in Date.EndOfQuarter function with that argument.

Date.EndOfWeek

Understanding the Date.EndOfWeek Function

The Date.EndOfWeek function is used to return the end of the week date for a given date. The syntax for this function is as follows:

Date.EndOfWeek(dateTime as any, optional firstDayOfWeek as nullable number) as nullable any

The `dateTime` parameter is required and represents the date for which you want to calculate the end of the week. The `firstDayOfWeek` parameter is optional and represents the first day of the week. If this parameter is not specified, the default value is 0, which represents Sunday.

How Date.EndOfWeek Works

The M code behind the Date.EndOfWeek function is fairly simple. The function uses the DateTime.LocalNow function to get the current date and time in the local time zone. It then uses the DateTime.DayOfWeek function to determine the day of the week for the `dateTime` parameter.

Once the day of the week is determined, the function subtracts the day of the week from 7 to get the number of days until the end of the week. It then adds this number of days to the `dateTime` parameter to get the end of the week date.

Here is the M code for the Date.EndOfWeek function:

```
let
  EndOfWeek = (dateTime as any, optional firstDayOfWeek as nullable number) =>
  let
    dayOfWeek = Date.DayOfWeek(dateTime),
    daysUntilEndOfWeek = if dayOfWeek <= (if firstDayOfWeek <> null then firstDayOfWeek else 0) then 7 – (if firstDayOfWeek <> null then firstDayOfWeek else 0) + dayOfWeek else 7 – dayOfWeek,
    endOfWeekDate = Date.AddDays(dateTime, daysUntilEndOfWeek)
  in
Date.EndOfYear
```

One of the date transformation functions available in Power Query M language is the Date.EndOfYear function. In this article, we will explore the M code behind the Date.EndOfYear function and how it can be used to transform dates in Power Query.

Understanding the Date.EndOfYear Function

The Date.EndOfYear function is a Power Query M function that returns the last day of the year for a given date value. The function takes a single argument, which is the date value for which the end of year date is to be calculated.

The syntax for the Date.EndOfYear function is as follows:

```
Date.EndOfYear(date as any) as date
```

The argument 'date' can be any valid date value, such as a column of date values in a table or a single date value.

Using the Date.EndOfYear Function in Power Query

To use the Date.EndOfYear function in Power Query, follow these steps:

1. Open a new Power Query Editor window in Excel.
2. Connect to a data source that contains a column of date values.
3. Select the column of date values.
4. Click on the 'Add Column' tab in the ribbon.
5. Click on the 'Date' dropdown menu.
6. Select 'End of Year'.

This will add a new column to the table that contains the end of year date for each date value in the selected column.

The M Code Behind the Date.EndOfYear Function

The M code behind the Date.EndOfYear function is relatively simple. It uses the Date.Year function to extract the year from the input date value and then constructs a new date value using the last day of that year.

The M code for the Date.EndOfYear function is as follows:

```
Date.From
```

But how does it work? In this article, we'll take a closer look at the M code behind the Date.From function and explain exactly how it converts text strings into dates.

Understanding the Date.From Function

Before we dive into the code behind the Date.From function, it's important to understand how it works. Essentially, the function takes a text string in a specific format and converts it into a date format that can be used in your data analysis.

The text string can be in a variety of formats, including "yyyy-mm-dd", "mm/dd/yyyy", "dd-mm-yyyy", and more. The function then uses this format to extract the year, month, and day values from the text string and combine them into a date format that can be used in your analysis.

The M Code Behind the Date.From Function

Now that we understand the basic functionality of the Date.From function, let's take a closer look at the M code behind it.

The Date.From function is actually a simple wrapper for the Date.FromText function. The Date.FromText function is responsible for taking the text string and converting it into a date format that can be used in your analysis.

Here's an example of the M code that powers the Date.From function:

```
let
  Source = #"SomeData",
  #"Converted to Date" = Table.TransformColumns(Source, {{"DateColumn", each Date.FromText(_, "yyyy-mm-dd"), type date}})
in
  #"Converted to Date"
```

Let's break down this code step by step:

1. We start by defining our data source. In this case, we're calling it "SomeData".
2. Next, we use the Table.TransformColumns function to transform the "DateColumn" in our data source. This is the column that contains the text strings we want to convert to dates.
3. In the transformation, we use the Date.FromText function to convert the text strings in the "DateColumn" into date formats. We

Date.FromText

What is the Date.FromText function?

The Date.FromText function is used to convert a text string into a date value. The function takes a text string as input and returns a date value. The syntax of the function is as follows:

Date.FromText(text as text, optional culture as nullable text)

- text: This is the text string that you want to convert to a date value.
- culture: This is an optional parameter that specifies the culture to use when converting the text string to a date value. If you don't specify a culture, the function will use the culture of your operating system.

How does the Date.FromText function work?

The Date.FromText function works by using a set of predefined date formats to parse the text string and convert it to a date value. The function tries each date format in turn until it finds a match. If none of the date formats match the text string, the function returns an error.

The predefined date formats that are used by the Date.FromText function depend on the culture that is specified. For example, if you specify the culture "en-US", the function will use the date formats that are commonly used in the United States. If you specify the culture "fr-FR", the function will use the date formats that are commonly used in France.

Here is an example of how the Date.FromText function works:

Date.FromText(“10/31/2021”)

In this example, the Date.FromText function will try to parse the text string "10/31/2021" using the predefined date formats. If the function is successful, it will return a date value of October 31, 2021.

The M code behind the Date.FromText function

Date.IsInCurrentDay

What is the Date.IsInCurrentDay function?

The Date.IsInCurrentDay function is a Power Query M function that can be used to determine if a given date falls within the current day. It takes a single argument, which is the date that needs to be checked. If the date falls within the current day, the function returns true. If the date does not fall within the current day, the function returns false.

How does the Date.IsInCurrentDay function work?

The Date.IsInCurrentDay function works by first retrieving the current date and time using the DateTime.LocalNow function. This returns a datetime value that represents the current date and time on the computer that is running the Power Query query.

Next, the function extracts the date portion of the current datetime value using the Date.From function. This returns a date value that represents the current date.

Finally, the function compares the input date to the current date by checking if the input date is equal to the current date using the Date.Equals function. If the input date is equal to the current date, the function returns true. Otherwise, it returns false.

Using the Date.IsInCurrentDay function

The Date.IsInCurrentDay function can be very useful when working with date ranges in Power Query. For example, you may want to filter a table to only include rows where the date falls within the current day. To do this, you can use the Date.IsInCurrentDay function in combination with the Table.SelectRows function.

Here's an example of how this can be done:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMlSK1YlWMlSK1YlWMlSK1YlWMlSK1YlWMlSK1YlWMlSK1YlWMlSK1YlWMlSK1YlWMlSK1YlWMlSK1YlWMlSK1YlWMlSK1YlWIVkZUeD0pMLCvVi1WqZgB", BinaryEncoding.Base64)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [#"Date" = _t]),
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"Date", type date}})
in
  #"Changed Type"
```

Date.IsInCurrentMonth

In this article, we are going to take a deep dive into the M code behind the Date.IsInCurrentMonth function. We will also explore some examples of how to use this function in Power Query.

Understanding the M Code

The M code behind the Date.IsInCurrentMonth function is relatively simple. It consists of a single line of code that checks whether a given date falls within the current month.

Here's the M code for the Date.IsInCurrentMonth function:

```
(Date) => Date.IsInCurrentMonth(Date)
```

Let's break down this code line by line.

The first line of code `(Date) =>` is the argument that the function takes. This argument specifies the date that we want to check.

The second line of code `Date.IsInCurrentMonth(Date)` is the code that checks whether the given date falls within the current month. If the given date is in the current month, the function returns `true`. If not, it returns `false`.

Examples of Using the Date.IsInCurrentMonth Function

Now that we understand the M code behind the Date.IsInCurrentMonth function, let's take a look at some examples of how to use this function.

Example 1: Finding Rows with Dates in Current Month

Suppose you have a table that contains a column of dates. You want to filter out all rows that do not have dates in the current month. Here's how you can achieve that using the Date.IsInCurrentMonth function.

1. Select the column of dates that you want to filter.
2. Click on the `Add Column` tab in the Power Query editor.
3. Click on `Custom Column`.
4. In the `Custom Column` dialog box, enter a name for the new column, e.g., "IsInCurrentMonth".
5. In the `Custom Column` dialog box, enter the following formula: `Date.IsInCurrentMonth([Date])`
6. Click `OK` to create the new column.

Date.IsInCurrentQuarter

Understanding the Date.IsInCurrentQuarter Function

Before we dive into the M code behind the Date.IsInCurrentQuarter function, it's important to understand what the function actually does. As mentioned earlier, the function checks whether a given date is in the current quarter. This means that it takes a single argument, which is the date that you want to check.

The function returns a boolean value, which is either True or False. If the date is within the current quarter, the function returns True. If not, it returns False.

The M Code Behind Date.IsInCurrentQuarter

Now that we understand what the Date.IsInCurrentQuarter function does, let's take a look at the M code behind it. The M code for the function is relatively simple and looks like this:

```
let Date.IsInCurrentQuarter = (date as date) =>
  let
    quarterStart = Date.StartOfQuarter(Date.From(DateTime.LocalNow())),
    quarterEnd = Date.EndOfQuarter(Date.From(DateTime.LocalNow()))
  in
    date >= quarterStart and date <= quarterEnd
in
  Date.IsInCurrentQuarter
```

The code starts by defining the Date.IsInCurrentQuarter function using the "let" keyword. The function takes a single argument, which is "date" and is of the data type "date".

Inside the function, two additional variables are defined using the "let" keyword: "quarterStart" and "quarterEnd". These variables are set using the "Date.StartOfQuarter" and "Date.EndOfQuarter" functions, respectively.

The "Date.StartOfQuarter" function takes a date as its argument and returns the first day of the quarter that contains that date. The "Date.EndOfQuarter" function takes a date as its argument and returns the last day of the quarter that contains that date.

Date.IsInCurrentWeek

What is Power Query M?

Power Query is a data transformation and cleansing tool that is part of Microsoft Excel and Power BI. It uses a functional programming language called M to perform various data manipulations. M is a case-sensitive language and is used to define queries, manipulate data, and create custom functions.

Date.IsInCurrentWeek Function

The Date.IsInCurrentWeek function is used to determine whether a given date falls within the current week or not. The syntax of this function is as follows:

```
Date.IsInCurrentWeek(dateTime as any) as logical
```

The function takes a single argument, which is a date/time value. It returns a logical value that indicates whether the provided date/time falls within the current week or not.

The M Code Behind Date.IsInCurrentWeek

The M code behind the Date.IsInCurrentWeek function is as follows:

```
(dateTime) =>
  let
    startOfWeek = Date.StartOfWeek(DateTime.LocalNow()),
    endOfWeek = Date.EndOfWeek(DateTime.LocalNow()),
    result = dateTime >= startOfWeek and dateTime <= endOfWeek
  in
    result
```

Date.IsInCurrentYear

Understanding the Date.IsInCurrentYear Function

Before we dive into the M code behind Date.IsInCurrentYear, let's first understand what this function does. This function takes in a single argument – a date value – and returns a Boolean value indicating whether the date falls within the current year or not.

Here's the syntax for using this function:

```
Date.IsInCurrentYear(date as any) as logical
```

Let's take a look at a few examples to understand how this function works.

Example 1

Suppose we have a table of sales data that looks like this:

Date	Sales
2020-01-01	1000
2020-02-01	1500
2021-01-01	2000
2021-02-01	2500

If we want to filter this table to show only the sales data from the current year, we can use the following M code:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMjQwVNJRMgYygQJQYGBgYGBgYGJgYmjgxo2aIaFmZmZgYA==", BinaryEncoding.Base64)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Date = _t, Sales = _t])),
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"Date", type date}, {"Sales", Int64.Type}}),
  #"Filtered Rows" = Table.SelectRows(#"Changed Type", each Date.IsInCurrentYear([Date]))
```

Date.IsInNextDay

Syntax of Date.IsInNextDay Function

The syntax of the Date.IsInNextDay function is as follows:

```
Date.IsInNextDay(dateTime as any) as logical
```

The function takes in a parameter called dateTime, which can be any value that can be converted to a date/time value. The function returns a logical value that indicates whether the given date and time value is within the next day of the current date and time.

How Date.IsInNextDay Function Works

The Date.IsInNextDay function works by comparing the given date and time value with the current date and time value. It first converts the given dateTime value to a date-only value by using the Date.FromDateTime function. It then adds one day to the current date and time value by using the DateTime.LocalNow function and the Duration.From function. The result is a date-only value that represents the next day of the current date and time.

The function then compares the given dateTime value with the next day value by using the Date.Compare function. If the given dateTime value is less than the next day value, the function returns true. Otherwise, it returns false.

Examples of Using Date.IsInNextDay Function

Here are some examples of using the Date.IsInNextDay function:

Example 1:

```
Date.IsInNextDay(#date(2022, 12, 31))
```

This example returns false because the date value #date(2022, 12, 31) is not within the next day of the current date and time.

Example 2:

```
Date.IsInNextMonth
```

In this article, we will take a closer look at the M code behind the Date.IsInNextMonth function, and explore how this function works.

Understanding the Date.IsInNextMonth Function

The Date.IsInNextMonth function is a Boolean function that returns either true or false, depending on whether a given date falls within the next month. This function takes a single argument, which is the date you want to check.

Here is an example of how you can use the Date.IsInNextMonth function in Power Query:

```
let
  Source = #table({“Date”},{{#date(2022, 4, 25)}, {#date(2022, 5, 25)}, {#date(2022, 6, 25)}}),
  #”Filtered Rows” = Table.SelectRows(Source, each Date.IsInNextMonth([Date]))
in
  #”Filtered Rows”
```

In this example, we are creating a table with three dates, and then filtering the table to only include dates that fall within the next month. The Date.IsInNextMonth function is used as the filter condition.

Exploring the M Code Behind Date.IsInNextMonth

Now that we understand how the Date.IsInNextMonth function works, let's dive into the M code behind this function.

The M code for the Date.IsInNextMonth function is relatively simple. Here is the code:

```
(Date) => Date.Month(Date.AddMonths(Date.From(Date),1)) = Date.Month(Date.AddMonths(Date.From(Date.IsDateTime(Date)),1))
```

Let's break down this code into its individual components:

- `(Date)` is the argument passed to the function. This is the date that we want to check whether it falls within the next month.
- `Date.IsDateTime(Date)` converts the input date to a DateTime value. This is required because the Date.AddMonths function only

Date.IsInNextNDays

Understanding the Date.IsInNextNDays Function

Before we dive into the M code behind the Date.IsInNextNDays function, let’s first understand what this function does. The Date.IsInNextNDays function takes in two arguments: a date and a number of days (N). It then checks whether the given date falls within the next N number of days.

For example, if today is January 1st, 2022, and we want to check whether January 7th, 2022 falls within the next 7 days, we can use the Date.IsInNextNDays function as follows:

```
Date.IsInNextNDays(#date(2022,1,7), 7)
```

This will return true, since January 7th, 2022 falls within the next 7 days from January 1st, 2022.

The M Code Behind the Date.IsInNextNDays Function

Now that we understand what the Date.IsInNextNDays function does, let's take a look at the M code behind this function. The M code for the Date.IsInNextNDays function is as follows:

```
(date as date, days as number) =>
  let
    today = Date.From(DateTime.LocalNow()),
    endDate = Date.AddDays(today, days),
    result = date <= endDate
  in
    result
```

Let's break down this code into smaller parts to understand how it works.

Date.IsInNextNMonths

Understanding the Date.IsInNextNMonths Function

The Date.IsInNextNMonths function is a handy tool that allows you to filter data based on dates falling within the next N number of months. It takes two arguments:

– Date: the date column you want to filter

– N: the number of months you want to filter for

The function returns a true or false value, indicating whether or not a date falls within the next N months.

The M Code Behind Date.IsInNextNMonths

The M code behind the Date.IsInNextNMonths function is relatively simple. Let’s take a look at the code:

```
(Date as date, N as number) =>
  Date.IsInNextNMonths = (date) =>
    let
      today = Date.StartOfDay(DateTime.LocalNow()),
      nextNMonths = Date.AddMonths(today, N)
    in
      date >= today and date < nextNMonths
```

As you can see, the function takes two arguments: the date column and the number of months. It then returns a Boolean value indicating whether or not the date falls within the next N months.

The code does this by first defining the variable "today" as the start of the current day using the Date.StartOfDay function, which extracts the date portion of the datetime value.

Next, the code defines the variable "nextNMonths" as the date N months from today using the Date.AddMonths function. This is the end date of the range we are interested in.

Lastly, the code checks if the date falls within the range of today and the next N months. If the date is greater than or equal to today and less than the next N months, the function returns true, indicating that the date falls within the next N months.

Date.IsInNextNQuarters

In this article, we will explore the M code behind the Date.IsInNextNQuarters function, how it works, and how you can use it in your own projects.

What is M code?

M code is the programming language behind Power Query. It is a functional language that is used to transform and clean data, and can be used to create custom functions and queries. M code is similar to other functional programming languages, such as F# and Haskell, but it is specifically designed for data analysis and transformation.

The Date.IsInNextNQuarters function

The Date.IsInNextNQuarters function is a Power Query function that checks if a given date is in the next N quarters. It takes two arguments, a date and a number of quarters, and returns a boolean value. If the date is in the next N quarters, the function returns true, otherwise it returns false.

Here is the M code for the Date.IsInNextNQuarters function:

```
(Date as date, Quarters as number) as logical =>
  let
    Today = Date.From(DateTime.LocalNow()),
    EndOfQuarter = Date.EndOfQuarter(Date.AddQuarters(Today, Quarters)),
    Result = Date.IsInNextNDays(Date, Duration.Days(Duration.From(EndOfQuarter – Today)))
  in
    Result
```

Let's break this code down and see how it works.

Breaking down the M code

The Date.IsInNextNQuarters function takes two arguments, Date and Quarters. Date is the date that you want to check, and Quarters is the number of quarters you want to check ahead. For example, if you set Quarters to 1, the function will check if the given date is in the next quarter.

Date.IsInNextNWeeks

What is Date.IsInNextNWeeks?

The Date.IsInNextNWeeks function is used to determine if a date falls within a certain number of weeks from the current date. It takes two arguments: the date to be tested, and the number of weeks from the current date. If the date falls within the specified number of weeks, the function returns true, otherwise it returns false.

The M Code Behind Date.IsInNextNWeeks

The M code behind the Date.IsInNextNWeeks function is relatively simple. Here is the code:

```
let
  IsInNextNWeeks = (date as date, numberOfWeeks as number) as logical =>
    let
      currentDate = Date.From(DateTime.LocalNow()),
      endDate = Date.AddDays(currentDate, numberOfWeeks  7)
    in
      date <= endDate
in
  IsInNextNWeeks
```

The code defines a function called IsInNextNWeeks that takes two arguments: the date to be tested and the number of weeks from the current date. The function returns a logical value indicating whether the date falls within the specified number of weeks.

The function first gets the current date using the DateTime.LocalNow() function, which returns the current date and time in the local time zone. The Date.From function is then used to extract the date from the current date and store it in a variable called currentDate.

Next, the function calculates the end date by adding the specified number of weeks to the current date using the Date.AddDays function. The result is stored in a variable called endDate.

Finally, the function checks whether the date to be tested is less than or equal to the end date. If it is, the function returns true, indicating that the date falls within the specified number of weeks. If it is not, the function returns false.

Date.IsInNextNYears

Understanding Date.IsInNextNYears Function

The Date.IsInNextNYears function is used to check whether a given date falls within the next n years from the current date. This function takes two arguments – a date and a number of years. If the date falls within the next n years, the function returns true. Otherwise, it returns false.

The syntax for the Date.IsInNextNYears function is as follows:

```
Date.IsInNextNYears(date as any, years as number) as logical
```

Let's take a look at an example to understand how this function works.

Suppose we have a table that contains a list of employees and their hire dates. We want to create a new column that indicates whether an employee is due for a performance review in the next 2 years. We can use the Date.IsInNextNYears function to achieve this.

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WSsosyczIzclPTcrPz0wpKi9R1UvOzwkvLlYwNzQwqzS11M1OLU7MzYzipWJKS1QwqA1DB1kqUMwNzQzMK1KzGpIBGJwCUwMzQzqzQxMk0VqJmFiZl5iTmliQrAEMVjI0NkFkZS1mZjAzLTQ2MjctODlyYS04NjhmOWQ3ZDc1MWEA", BinaryEncoding.Base64)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Employee = _t, HireDate = _t]),
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"Employee", type text}, {"HireDate", type date}}),
  #"Added Custom" = Table.AddColumn(#"Changed Type", "Performance Review Due", each Date.IsInNextNYears([HireDate], 2))
in
  #"Added Custom"
```

Date.IsInNextQuarter

Understanding the Date.IsInNextQuarter Function

The Date.IsInNextQuarter function is a useful function in Power Query that allows you to check whether a given date falls in the next quarter. The function returns a Boolean value, which is either True or False. If the date falls in the next quarter, the function will return True, otherwise, it will return False.

The Syntax for the Date.IsInNextQuarter Function

The syntax for the Date.IsInNextQuarter function is as follows:

```
Date.IsInNextQuarter(date as any) as logical
```

The function takes a single argument, which is a date as any data type. The function returns a logical data type, which is either True or False.

The M Code Behind the Date.IsInNextQuarter Function

The M code behind the Date.IsInNextQuarter function is as follows:

```
let
  IsInNextQuarter = (date as any) as logical =>
  let
    CurrentQuarter = Number.RoundUp(Date.Month(date) / 3),
    CurrentQuarterEndDate = Date.EndOfQuarter(date),
    NextQuarterStartDate = Date.AddDays(CurrentQuarterEndDate, 1),
    NextQuarterEndDate = Date.EndOfQuarter(Date.AddQuarters(date, 1)),
    IsInNextQuarter = if date >= NextQuarterStartDate and date <= NextQuarterEndDate then true else false
  in
    IsInNextQuarter
```

Date.IsInNextWeek

Understanding the Date.IsInNextWeek Function

The Date.IsInNextWeek function is a logical function that returns true if a given date falls within the next week, otherwise it returns false. The syntax for this function is as follows:

```
Date.IsInNextWeek(dateTime as any, optional firstDayOfWeek as nullable number) as logical
```

The dateTime parameter is the date that you want to check if it falls within the next week. The firstDayOfWeek parameter is an optional parameter that specifies the first day of the week. If this parameter is not specified, the function uses the default first day of the week (Sunday).

The M Code Behind the Date.IsInNextWeek Function

The M code behind the Date.IsInNextWeek function is quite simple. Here is the M code for this function:

```
(dateTime as any, optional firstDayOfWeek as nullable number) as logical =>
  let
    nextWeekStart = Date.StartOfWeek(DateTime.LocalNow(),
          if firstDayOfWeek <> null then firstDayOfWeek else Day.Sunday),
    nextWeekEnd = Date.AddDays(nextWeekStart, 6),
    dateToCheck = Date.From(dateTime)
  in
    dateToCheck >= nextWeekStart and dateToCheck <= nextWeekEnd
```

The code above defines an anonymous function that takes two parameters: dateTime and firstDayOfWeek. The function first determines the start and end date of the next week using the Date.StartOfWeek and Date.AddDays functions. It then checks if the dateTime

Date.IsInNextYear

Understanding the Date.IsInNextYear Function

Before we dive deep into the M code behind the Date.IsInNextYear function, let’s first understand what this function does. The Date.IsInNextYear function takes a date value as an argument and returns a Boolean value indicating whether the date is in the next year or not.

For example, consider the following date value:

```
Date.IsInNextYear(#date(2021, 12, 31))
```

The above expression will return a value of "True" because the date value represents a date in the next year.

The M Code Behind the Date.IsInNextYear Function

Behind the scenes, the Date.IsInNextYear function is written in M code. M is the formula language used in Power Query. It is used to create custom functions, perform data transformations, and load data into Excel.

The M code behind the Date.IsInNextYear function is relatively simple. It consists of a single line of code:

```
each Date.Year([Date]) = Date.Year(Date.AddYears(DateTime.LocalNow(),1))
```

Let's break down this code to understand how it works.

The each Keyword

The each keyword is used to indicate that the following expression should be evaluated for each row of data in the table.

The Date.Year Function

The Date.Year function is used to extract the year value from the date value.

Date.IsInPreviousDay

Introduction to Date.IsInPreviousDay

Before we dive into the M code behind the Date.IsInPreviousDay function, let's first understand what this function does. Essentially, this function returns a Boolean value that indicates whether a given date falls in the previous day or not. For example, if the current date is 10th March 2021, the function will return TRUE for any date that falls on 9th March 2021.

Understanding the M Code Behind Date.IsInPreviousDay

The M code behind the Date.IsInPreviousDay function is relatively simple. It consists of a single line of code that uses the Date.AddDays function to subtract one day from the current date and then checks if the given date falls in the range of dates between the start and end of the previous day.

Here's the M code for the Date.IsInPreviousDay function:

```
(Date) => Date.IsInPreviousDay = (Date.AddDays(DateTime.LocalNow(), -1), Date.AddDays(DateTime.LocalNow(), -1) + #duration(1, 0, 0, 0))
```

Let's break down this code line by line:

- `(Date)` is the parameter that is passed to the function. This is the date that we want to check if it falls in the previous day or not.
- `=>` is the lambda operator that separates the parameter from the function body.
- `Date.IsInPreviousDay` is the name of the function that we are defining.
- `=` is the assignment operator that assigns the value of the function to the expression on the right-hand side.
- `Date.AddDays(DateTime.LocalNow(), -1)` subtracts one day from the current date using the Date.AddDays function. This gives us the start of the previous day.
- `Date.AddDays(DateTime.LocalNow(), -1) + #duration(1, 0, 0, 0)` adds one day to the start of the previous day using the #duration function. This gives us the end of the previous day.
- `Date.IsInPreviousDay = ...` checks if the given date falls in the range between the start and end of the previous day. If it does, the function returns TRUE, otherwise, it returns FALSE.

Using Date.IsInPreviousDay in Power Query

Date.IsInPreviousMonth

This function comes in handy when you need to filter data based on whether it falls within the previous month. For instance, if you want to extract data from your sales database for the previous month, you can use the Date.IsInPreviousMonth function to filter out the relevant data.

In this article, we will explore the M code behind the Date.IsInPreviousMonth function and how it works.

Understanding the Date.IsInPreviousMonth Function

The Date.IsInPreviousMonth function is a Power Query M function that checks whether a given date falls within the previous month. The function takes a date value as its argument and returns a logical value (True or False) indicating whether the date falls within the previous month.

The syntax for the Date.IsInPreviousMonth function is as follows:

```
Date.IsInPreviousMonth(date as any) as logical
```

Here, the `date` argument can be any date value, such as a column of date values in a table.

The M Code Behind the Function

The M code behind the Date.IsInPreviousMonth function is relatively simple. It uses the Date.Year, Date.Month, and Date.DaysInMonth functions to determine the start and end dates of the previous month and then checks whether the input date falls within this range.

Here is the M code for the Date.IsInPreviousMonth function:

```
(date as any) =>
let
  currentMonth = Date.Month(date),
  currentYear = Date.Year(date),
  daysInMonth = Date.DaysInMonth(#date(currentYear, currentMonth, 1)),
  startDate = #date(currentYear, currentMonth-1, 1),
```

Date.IsInPreviousNDays

Understanding the Date.IsInPreviousNDays Function

The Date.IsInPreviousNDays function is used to filter data based on a date range that falls within a certain number of days from the current date. The syntax of the function is as follows:

```
Date.IsInPreviousNDays(dateTime as any, numberOfDays as number) as logical
```

The function takes two arguments. The first argument is the dateTime value that we want to check, and the second argument is the numberOfDays value that specifies the number of days from the current date that we want to include in our filter.

The function returns a logical value of either true or false, depending on whether the dateTime value falls within the specified number of days from the current date.

The M Code Behind the Date.IsInPreviousNDays Function

The M code behind the Date.IsInPreviousNDays function is relatively simple. Here is the code:

```
(dateTime as any, numberOfDays as number) =>
  let
    today = DateTime.LocalNow(),
    dateRangeStart = Date.AddDays(today, -numberOfDays),
    result = dateTime >= dateRangeStart and dateTime <= today
  in
    result
```

The code defines an anonymous function that takes the two arguments, dateTime and numberOfDays. The first line of the function defines a variable named today that stores the current date and time using the DateTime.LocalNow() function.

Date.IsInPreviousNMonths

How Does Date.IsInPreviousNMonths Work?

The function Date.IsInPreviousNMonths takes two parameters, the first is the date column you want to filter on, and the second is the number of months in the past you want to include. The function will return all rows where the date is within the specified range.

Here’s an example of how this function can be used. Suppose you have a dataset containing sales data for the past year, and you want to filter out all sales that occurred more than six months ago. Here’s how you can use the Date.IsInPreviousNMonths function to accomplish this:

```
= Table.SelectRows(Sales, each Date.IsInPreviousNMonths([Sale Date], 6))
```

In this example, "Sales" is the name of the table containing the sales data, "Sale Date" is the name of the column containing the date of each sale, and "6" is the number of months in the past that we want to include.

The M Code Behind Date.IsInPreviousNMonths

If you're curious about the M code behind the Date.IsInPreviousNMonths function, here it is:

```
(date as any, months as number) as logical =>
let
  fromDate = Date.AddMonths(DateTime.LocalNow(), -months),
  toDate = DateTime.LocalNow()
in
  fromDate <= date and date <= toDate
```

Let's break down this code a bit. The function takes two parameters, "date" and "months". The "date" parameter is the date that we're checking against, and the "months" parameter is the number of months in the past that we want to include.

Date.IsInPreviousNQuarters

What is the Date.IsInPreviousNQuarters function?

The Date.IsInPreviousNQuarters function is a Power Query M function that returns a boolean value indicating whether a given date falls within the previous N quarters. The function takes two arguments: a date value and an integer value indicating the number of quarters to check.

Here is the syntax for the Date.IsInPreviousNQuarters function:

```
Date.IsInPreviousNQuarters(date as any, quarters as number) as logical
```

How does the function work?

The Date.IsInPreviousNQuarters function works by first calculating the start and end dates for the previous N quarters based on the current date. It then checks whether the given date falls within this date range and returns a boolean value accordingly.

Here is the M code for the Date.IsInPreviousNQuarters function:

```
let
  IsInPreviousNQuarters = (date as any, quarters as number) as logical =>
    let
      currentDate = Date.From(DateTime.LocalNow()),
      startQuarter = Date.StartOfQuarter(Date.AddQuarters(currentDate, -quarters)),
      endQuarter = Date.EndOfQuarter(Date.AddQuarters(currentDate, -1))
    in
      date >= startQuarter and date <= endQuarter
in
  IsInPreviousNQuarters
```

Date.IsInPreviousNWeeks

Understanding the Date.IsInPreviousNWeeks function

The Date.IsInPreviousNWeeks function takes two parameters: the date to check and the number of weeks to go back. It returns a boolean value: true if the date falls within the previous N weeks, false otherwise. Here is an example of how to use this function:

```
let
  Source = #table({“Date”}, {{“2022-02-01”}, {“2022-01-01”}, {“2021-12-01”}}),
  PreviousWeeks = 4,
  CustomFunction = (date as date) => Date.IsInPreviousNWeeks(date, PreviousWeeks),
  AddColumn = Table.AddColumn(Source, “IsInPreviousWeeks”, each CustomFunction([Date])),
  RemoveColumns = Table.RemoveColumns(AddColumn, {“Date”})
in
  RemoveColumns
```

In this code, we create a table with a Date column, define the number of previous weeks to check (4), and create a custom function that uses the Date.IsInPreviousNWeeks function. We then add a new column to the table that applies this function to each value in the Date column, and finally remove the original Date column to keep only the boolean result.

Breaking down the M code

The M code behind the Date.IsInPreviousNWeeks function is straightforward:

```
(Date.ToWeekNumber(DateTime.LocalNow()) – Date.ToWeekNumber(date)) <= weeksAgo
```

This code calculates the week number of the current date (using DateTime.LocalNow()), subtracts the week number of the date to check (using Date.ToWeekNumber()), and compares the result to the number of previous weeks specified (weeksAgo). If the difference

Date.IsInPreviousNYears

What is the Date.IsInPreviousNYears function?

The Date.IsInPreviousNYears function is a Power Query M function that returns true if a date falls within the specified number of years from the current date. It takes two arguments:

– Date: The date to be checked.

– Years: The number of years to check against the current date.

This function is useful for creating dynamic reports that require data from a specific time frame, such as the last 3 years.

How does the Date.IsInPreviousNYears function work?

The Date.IsInPreviousNYears function works by subtracting the specified number of years from the current date using the Date.AddYears function, and then checking if the date is greater than or equal to the resulting date.

Here is the M code for the Date.IsInPreviousNYears function:

```
(Date as date, Years as number) as logical =>
  Date >= Date.AddYears(DateTime.LocalNow(), -Years) and
  Date < DateTime.LocalNow()
```

The function takes the Date and Years arguments and compares the date to the result of subtracting the Years from the current date. If the date falls within the specified time frame, the function returns true, otherwise false.

How to use the Date.IsInPreviousNYears function?

To use the Date.IsInPreviousNYears function, you need to follow these steps:

1. Load your data into Power Query.
2. Select the column that contains the dates you want to filter.
3. Open the Transform tab and click on the "Date" dropdown menu.
4. Select the "Is in Previous N Years" option.
5. Enter the number of years you want to check against the current date.

After applying the filter, only the dates that fall within the specified time frame will be displayed.

Date.IsInPreviousQuarter

One of the many useful functions available in Power Query is the Date.IsInPreviousQuarter function. This function allows you to determine if a given date falls within the previous quarter. In this article, we will take a closer look at the M code behind the Date.IsInPreviousQuarter function and how it works.

What is Power Query M Language?

Power Query M language is a functional programming language that is used to create queries in Power Query. It is a powerful language that allows you to perform a wide range of data transformations, including filtering, sorting, grouping, and more.

The M language is based on the functional programming paradigm, which means that it is designed to be highly modular and composable. This makes it easy to create complex queries by combining simple building blocks together.

Understanding the Date.IsInPreviousQuarter Function

The Date.IsInPreviousQuarter function is a simple function that takes a single argument, a date value, and returns a Boolean value that indicates whether the date falls within the previous quarter. The function is defined as follows:

```
Date.IsInPreviousQuarter(date as any) as logical
```

The function takes a single parameter, "date," which can be any date value, including a date literal or a column reference. The function returns a logical value, which is either true or false, depending on whether the date falls within the previous quarter.

The M Code Behind the Date.IsInPreviousQuarter Function

The M code behind the Date.IsInPreviousQuarter function is relatively straightforward. The function is implemented using a series of M language expressions that perform various date calculations.

The first expression in the function calculates the start date of the previous quarter. This is done using the Date.StartOfQuarter function, which takes the current date and returns the start date of the quarter that the current date falls within. The function then subtracts three months from the start date to get the start date of the previous quarter.

```
let
Date.IsInPreviousWeek
```

In this article, we will explore the M code behind the Power Query M function Date.IsInPreviousWeek.

Overview of the Date.IsInPreviousWeek function

The Date.IsInPreviousWeek function is a useful function in Power Query that allows you to determine whether a date falls within the previous week. The function takes a single argument, which is the date to be tested, and returns a Boolean value indicating whether the date falls within the previous week.

Here's an example of how to use the Date.IsInPreviousWeek function:

```
= Date.IsInPreviousWeek(#date(2022, 5, 2))
```

This formula returns the value `true` if the date `May 2, 2022` falls within the previous week, and `false` otherwise.

Understanding the M code behind the Date.IsInPreviousWeek function

To understand the M code behind the Date.IsInPreviousWeek function, we need to look at the function definition in the Advanced Editor of Power Query. To do this, follow these steps:

1. Open Power Query and create a new query.
2. Click on the "View" tab and select "Advanced Editor."
3. In the Advanced Editor, enter the following code:

```
(Date as date) as logical =>
let
  PreviousWeekStart = Date.AddDays(Date.StartOfWeek(Date), -7),
  PreviousWeekEnd = Date.AddDays(PreviousWeekStart, 7),
  IsInPreviousWeek = Date >= PreviousWeekStart and Date < PreviousWeekEnd
in
  IsInPreviousWeek
```

Date.IsInPreviousYear

Understanding the Date.IsInPreviousYear Function

Before we dive into the M code, let’s first understand the Date.IsInPreviousYear function. This function takes a date value as its argument and returns a Boolean value indicating whether the date is in the previous year. Here’s an example:

```
let
  myDate = #date(2022, 1, 1),
  result = Date.IsInPreviousYear(myDate)
in
  result
```

In this example, we create a date value representing January 1st, 2022. We then pass this value to the Date.IsInPreviousYear function, which returns false because the date is not in the previous year.

The M Code Behind Date.IsInPreviousYear

Now that we understand the function, let's take a look at the M code behind it. Here's the full code:

```
(Date) =>
  let
    PrevYear = Date.Year(Date.AddYears(Date.From(Date.LocalNow()),-1)),
    PrevYearStart = #date(PrevYear, 1, 1),
    PrevYearEnd = #date(PrevYear, 12, 31)
  in
    Date.InRange(Date, PrevYearStart, PrevYearEnd)
```

Date.IsInYearToDate

In this article, we will explore the M code behind the Date.IsInYearToDate function, and provide a detailed explanation of how it works.

What is the Date.IsInYearToDate Function?

The Date.IsInYearToDate function in Power Query M is a Boolean function that takes a date value as its argument, and returns true if the date falls within the current year-to-date period, and false otherwise.

The function has the following syntax:

```
Date.IsInYearToDate(date as date) as logical
```

The argument 'date' is a required input, and must be a valid date value.

How Does the Function Work?

The Date.IsInYearToDate function works by first calculating the start date of the current year-to-date period, and then comparing the input date to this start date.

To calculate the start date of the current year-to-date period, the function uses the following expression:

```
StartDate = Date.StartOfYear(DateTime.LocalNow())
```

This expression returns the start date of the current year, which is then used to calculate the start date of the year-to-date period:

```
YearToDateStart = Date.StartOfDay(Date.AddYears(StartDate, -1))
```

This expression subtracts one year from the start date of the current year, and then calculates the start of the day for this date. This gives

Date.IsLeapYear

What is a Leap Year?

Before we dive into the M code behind the Date.IsLeapYear function, let's first understand what a leap year is. A leap year is a year that is evenly divisible by 4, except for years that are divisible by 100 but not by 400. For example, the year 2000 was a leap year because it is divisible by 4 and by 400, but the year 1900 was not a leap year because it is divisible by 4 and by 100, but not by 400.

The M Code Behind the Date.IsLeapYear Function

The M code behind the Date.IsLeapYear function is relatively simple. The function takes a single input parameter, which is the year to be tested for leap year. The M code then performs a series of calculations to determine whether the year is a leap year or not.

Here is the M code behind the Date.IsLeapYear function:

```
(Date.Year(Date.From(DateTime.LocalNow())) mod 4 = 0 and Date.Year(Date.From(DateTime.LocalNow())) mod 100 <> 0) or Date.Year(Date.From(DateTime.LocalNow())) mod 400 = 0
```

Let's break down this M code and understand how it works.

1. The Date.Year function takes a date value as input and returns the year component of the date.
2. The Date.From function takes a datetime value as input and returns the date component of the datetime value.
3. The DateTime.LocalNow function returns the current local date and time as a datetime value.
4. The mod operator performs a modulo operation, which returns the remainder of the division of the two operands.
5. The and and or operators perform logical conjunction and disjunction, respectively.

Putting all of these elements together, the M code behind the Date.IsLeapYear function first gets the current year using the DateTime.LocalNow function. It then checks whether the year is evenly divisible by 4 using the mod operator. If the year is divisible by 4, the function checks whether the year is not divisible by 100 using the <> operator. If the year is not divisible by 100 or is divisible by 400, the function returns true, indicating that the year is a leap year. Otherwise, the function returns false, indicating that the year is not a leap year.

Using the Date.IsLeapYear Function

To use the Date.IsLeapYear function in Power Query, simply pass a year value as input to the function. For example, to test whether the

Date.Month

What is the Date.Month Function?

The Date.Month function is a built-in function in Power Query that extracts the month from a date value. The function takes a date value as its input and returns the month as an integer between 1 and 12. For example, if you have a date value of January 1, 2022, the Date.Month function would return 1.

The M Code Behind the Date.Month Function

The Date.Month function is actually a simple piece of M code that you can write yourself. Here is the M code for the Date.Month function:

```
(Date) => Date.Month(Date)
```

As you can see, the function takes a date value as its input (which we have named "Date" in this case) and then calls the built-in Date.Month function to extract the month from the date. That's it! With this simple code, you can create your own Date.Month function in Power Query.

Using the Date.Month Function in Power Query

Now that you understand the M code behind the Date.Month function, let's look at how you can use it in your own Power Query projects. Here is an example of how to use the Date.Month function to extract the month from a date column in a table:

1. Open Power Query and load the table that contains the date column.
2. Select the date column and click on the "Add Column" tab.
3. Click on "Custom Column" and enter the following formula:

```
= Date.Month([Date])
```

4. Click "OK" to create the new column.

Date.MonthName

But have you ever wondered how the Date.MonthName function actually works? In this article, we will take a deep dive into the M code behind the Date.MonthName function to understand how this powerful function is built.

The Basic Syntax

Before we dive into the code, let's first take a look at the basic syntax of the Date.MonthName function. The syntax is as follows:

Date.MonthName(month as number, optional abbreviated as nullable logical, optional culture as nullable text) as text

The function takes three arguments:

- `month` (required): This is the numerical value of the month that you want to convert to a month name. This value must be between 1 and 12.
- `abbreviated` (optional): This argument is a logical value that determines whether you want the abbreviated or full month name. If this argument is set to `true`, then the abbreviated month name is returned. Otherwise, the full month name is returned. The default value is `false`.
- `culture` (optional): This argument is a text value that specifies the culture to use when converting the month name. If this argument is not provided, then the culture of the current user is used.

The M Code Behind the Function

Now let's take a look at the M code behind the Date.MonthName function. The M code is what actually makes the function work behind the scenes.

```
let
  MonthNames = {“January”, “February”, “March”, “April”, “May”, “June”, “July”, “August”, “September”, “October”, “November”, “December”},
  AbbreviatedMonthNames = {“Jan”, “Feb”, “Mar”, “Apr”, “May”, “Jun”, “Jul”, “Aug”, “Sep”, “Oct”, “Nov”, “Dec”},
  MonthIndex = month – 1,
```

Date.QuarterOfYear

In this article, we'll take a closer look at the M code behind the Date.QuarterOfYear function, and how you can use it to manipulate your data.

Understanding the Date.QuarterOfYear Function

The Date.QuarterOfYear function is a simple function that takes a date value as input and returns the quarter of the year that the date falls in. For example, if the date is January 1st, the function will return 1, since it falls in the first quarter of the year. If the date is April 1st, the function will return 2, since it falls in the second quarter of the year.

The syntax for the function is as follows:

```
Date.QuarterOfYear(date as any) as number
```

The "date" parameter can be any valid date value, including a date time value or a date value extracted from a column in a table.

The M Code Behind the Date.QuarterOfYear Function

Now that we understand what the Date.QuarterOfYear function does, let's take a closer look at the M code behind it.

The M code for the Date.QuarterOfYear function is relatively simple, and looks like this:

```
(date) =>
let
  quarter = Number.RoundUp(Date.Month(date)/3),
in
  quarter
```

Let's break this code down line by line.

The first line defines a function that takes a single parameter, "date", which represents the date value that we want to extract the quarter

Date.StartOfDay

What is Date.StartOfDay?

Date.StartOfDay is a Power Query M function that is used to get the start of a day for a given date. This function returns a datetime value that represents the start of the day for the given datetime value. The time portion of the returned datetime value is set to 12:00:00 AM.

The M Code Behind Date.StartOfDay

The M code behind Date.StartOfDay is quite simple. The following is the M code for the Date.StartOfDay function:

```
(Date as datetime) as datetime => DateTime.Date(Date) & #time(0,0,0)
```

This code is a function that takes a datetime value as input and returns a datetime value. Let's break down this code step by step.

Step 1: DateTime.Date(Date)

The first part of the code, DateTime.Date(Date), is used to get the date portion of the datetime value. This function returns a date value that represents the date portion of the given datetime value. For example, if the input datetime value is "2022-01-01 10:30:00", the output date value will be "2022-01-01".

Step 2: & #time(0,0,0)

The second part of the code, "& #time(0,0,0)", is used to append the time portion to the date value. This function returns a time value that represents the time portion of the given datetime value. In this case, the time value is set to 12:00:00 AM, which is the start of the day.

Combining the Steps

When we combine these two steps, we get a datetime value that represents the start of the day for the given datetime value. For example, if the input datetime value is "2022-01-01 10:30:00", the output datetime value will be "2022-01-01 12:00:00 AM".

Examples of Date.StartOfDay Usage

Let's look at some examples of how Date.StartOfDay can be used in Power Query.

Example 1: Filter by Start of Day

Suppose we have a table that contains a datetime column, and we want to filter the rows that have a datetime value that falls on the start of the day. We can use the following M code to achieve this:

Date.StartOfMonth

Syntax
The syntax for the Date.StartOfMonth function is as follows:

```
Date.StartOfMonth(dateTime as any) as any
```

The function takes a single parameter, dateTime, which can be any valid date/time value. The function returns a date/time value representing the first day of the month for the given dateTime value.
Examples
Here are some examples of how the Date.StartOfMonth function can be used:
Example 1
Suppose you have a table of sales data that includes a column for the date of each sale. You want to group the sales data by month and calculate the total sales for each month. You can use the following code to achieve this:

```
let
  Source = Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMlTSUXLIyCxKLU0tKMlXKMpNzSspVrJSaiQK", BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Date = _t, Sales = _t]),
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"Date", type date}, {"Sales", Int64.Type}}),
  #"Grouped Rows" = Table.Group(#"Changed Type", {"Date.StartOfMonth"}, {{"Total Sales", each List.Sum([Sales]), type nullable number}})
in
  #"Grouped Rows"
```

Date.StartOfQuarter

Understanding the Date.StartOfQuarter Function

The Date.StartOfQuarter function is used to return the first day of the quarter for a given date. This function takes a date as an input and returns a date. The syntax for the Date.StartOfQuarter function is as follows:

```
Date.StartOfQuarter(date as date) as date
```

The Date.StartOfQuarter function takes a single argument, which is a date value. This argument can be a reference to a cell that contains a date, or a date value entered directly into the function.

The M Code Behind the Date.StartOfQuarter Function

The M code behind the Date.StartOfQuarter function is relatively simple. The following code snippet shows the M code behind the Date.StartOfQuarter function:

```
(date as date) => Date.StartOfQuarter(date)
```

This code defines an anonymous function that takes a date as an input and calls the Date.StartOfQuarter function with the input date. The output of the Date.StartOfQuarter function is returned as the output of the anonymous function.

The Date.StartOfQuarter function uses the Date.From function to convert the input date into a date value that can be used for calculations. The following code snippet shows the M code for the Date.From function:

```
Date.From(date as any) as nullable date
```

Date.StartOfWeek

In this article, we're going to dive into the M code behind the Date.StartOfWeek function and explore how it works. We'll also look at some examples of how you can use this function to transform your data.

The Basics of Date.StartOfWeek

Before we dive into the M code, let's go over the basics of the Date.StartOfWeek function. This function takes a date value as its first argument and an optional argument that specifies the day of the week to use as the start of the week. If you don't specify a day of the week, Power Query M will use the default value of Monday.

Here's an example of the basic syntax for the Date.StartOfWeek function:

```
Date.StartOfWeek(date as date, optional firstDayOfWeek as nullable number) as date
```

In this example, "date" is the input date value and "firstDayOfWeek" is the optional argument that specifies the day of the week to use as the start of the week. The function returns a date value.

The M Code Behind Date.StartOfWeek

Now that we know the basics of the Date.StartOfWeek function, let's take a look at the M code that powers this function.

When you call the Date.StartOfWeek function in your Power Query M code, Power Query M will execute the following code:

```
(DateTime.LocalNow() – Duration.FromDays(DateTime.LocalNow().DayOfWeek – firstDayOfWeek – 1))
```

Let's break down this code to understand how it works:

- DateTime.LocalNow() returns the current date and time.
- DateTime.LocalNow().DayOfWeek returns the day of the week for the current date and time.
- "firstDayOfWeek" is the optional argument that specifies the day of the week to use as the start of the week.
- "- firstDayOfWeek - 1" calculates the number of days to subtract from the current date to get to the start of the week, based on the

Date.StartOfYear

Understanding the Date.StartOfYear Function

Before we dive into the M code, let's take a moment to understand what the Date.StartOfYear function does. This function takes a single argument, a date value, and returns a new date value that represents the first day of the year for that date. For example, if you pass in the date value 1/15/2022, the function will return the date value 1/1/2022.

Exploring the M Code Behind Date.StartOfYear

To better understand how the Date.StartOfYear function works, let's take a look at the M code behind it. You can view the M code by clicking on the "Advanced Editor" button in the Power Query Editor and finding the Date.StartOfYear function in the list of available functions.

Here is the M code for the Date.StartOfYear function:

```
(date as nullable date) as nullable date =>
  let
    year = Date.Year(date),
    result = #date(year, 1, 1)
  in
    result
```

Let's break down this code line by line:

- `(date as nullable date) as nullable date =>` - This line defines the function signature. It takes one argument, `date`, which is a nullable date value, and returns a nullable date value.
- `let` - This line starts a `let` expression, which allows us to define variables and perform calculations.
- `year = Date.Year(date),` - This line defines a variable `year` and sets its value to the year component of the `date` argument using the Date.Year function.
- `result = #date(year, 1, 1)` - This line defines a variable `result` and sets its value to a new date value using the `#date` syntax. The first argument is the year component of the `date` argument, the second argument is `1` to represent the first month of the year, and

Date.ToRecord

Understanding Date Values in Power Query

Before we dive into the M code for Date.ToRecord, it’s important to have a basic understanding of how dates are represented in Power Query. In Power Query, dates are stored as serial numbers, with each day represented by a whole number starting from 1 (January 1st, 1900). For example, January 1st, 2010 would be represented as the serial number 40179.

While these serial numbers might seem arbitrary, they allow for easy manipulation and calculation of dates within Power Query. For example, adding 1 to a date serial number will result in the next day, and subtracting 7 will result in a date one week earlier.

The M Code Behind Date.ToRecord

Now that we understand how dates are represented in Power Query, let’s take a look at the M code behind the Date.ToRecord function:

```
(date as date) as record =>
  [
    Year = Date.Year(date),
    Month = Date.Month(date),
    Day = Date.Day(date)
  ]
```

The code above defines the Date.ToRecord function, which takes a single argument - the date value that we want to convert to a record. The function then returns a record with three fields: Year, Month, and Day.

Let's break down the code in more detail. The first line of the code specifies the input parameter for the function - in this case, a date value. The "as record" at the end of the function definition specifies the output type of the function - in this case, a record.

The next few lines of code are where the real work happens. The function uses three built-in M functions - Date.Year, Date.Month, and Date.Day - to extract the year, month, and day from the input date value. These values are then used to create a new record with fields for Year, Month, and Day.

Using Date.ToRecord in Your Data Transformations

Now that we understand the M code behind the Date.ToRecord function, let's look at how we can use it in our own data transformations.

Date.ToText

Introduction to Power Query

Before we dive into the M code behind the Date.ToText function, let's first take a quick look at what Power Query is and what it does. Power Query allows you to connect to a variety of data sources, including databases, Excel files, and CSV files, and then transform that data in a variety of ways.

Power Query is a visual interface that allows you to perform these transformations using a drag and drop interface. However, behind the scenes, all of these transformations are actually being performed using M code.

The Date.ToText Function

One of the most commonly used functions in Power Query is the Date.ToText function. This function takes a date as input and returns a text string representing that date in a specified format. The syntax for the Date.ToText function is as follows:

```
Date.ToText(dateTime as any, optional format as nullable text, optional culture as nullable text) as text
```

The first argument, `dateTime`, is the date that you want to format. The second argument, `format`, is an optional text string that specifies the format that you want to use. If you don't specify a format, the function will use the default format for your region. The third argument, `culture`, is also optional, and specifies the culture that you want to use for formatting the date.

Examples of Date.ToText

Let's take a look at some examples of the Date.ToText function in action. Suppose you have a date column in your data that looks like this:

Date
01/01/2021
02/01/2021
03/01/2021

You can use the Date.ToText function to format these dates in a variety of ways. For example, the following formula will format the dates as "January 1, 2021":

Date.WeekOfMonth

What is the Date.WeekOfMonth function?

The Date.WeekOfMonth function is a built-in function in Power Query that returns the week number of the month in which the given date falls. The function takes a date value as input and returns a whole number between 1 and 5, representing the week number of the month. If the date falls in the first week of the month, the function returns 1. If the date falls in the second week of the month, it returns 2, and so on, up to 5 for the last week of the month.

Here is an example of how the function works:

`Date.WeekOfMonth(#date(2022, 11, 15))`

This returns `3`, because November 15, 2022 falls in the third week of the month.

The M code behind the Date.WeekOfMonth function

The M code behind the Date.WeekOfMonth function is relatively simple. Here is the code:

```
let
  WeekOfMonth = (dateValue as date) =>
    let
      FirstDayOfMonth = Date.StartOfMonth(dateValue),
      DayOfWeek = Date.DayOfWeek(FirstDayOfMonth),
      DaysFromStart = Date.Day(dateValue) – 1,
      DaysFromFirstWeekStart = 7 – DayOfWeek,
      WeeksFromStart = Number.RoundDown((DaysFromStart + DaysFromFirstWeekStart) / 7),
      WeekOfMonth = WeeksFromStart + 1
    in
      WeekOfMonth
in
  WeekOfMonth
```

Date.WeekOfYear

What is the Date.WeekOfYear Function?

The Date.WeekOfYear function is a Power Query M function that returns the week number of a given date. The function takes a single argument, which is the date to be evaluated. The week number is determined based on the ISO 8601 standard, which defines the first week of the year as the week containing the fourth day of January.

Understanding the M Code Behind Date.WeekOfYear

To understand how the Date.WeekOfYear function works, we need to take a closer look at the M code behind it. The M code for the function is as follows:

```
(Date) =>
let
  DayOfWeek = Date.DayOfWeek(Date.AddDays(Date,”-01-04"), Day.Monday),
  YearStart = Date.AddDays(Date,”-01-01"),
  WeekNumber = Number.RoundDown((Date – YearStart + (DayOfWeek-1))/7) + 1
in
  WeekNumber
```

Let's break this code down into its individual components and understand what each part does.

The Date Parameter

The first line of the code defines the Date parameter, which is the date for which we want to find the week number. This parameter is passed to the function when it is called, and the function uses it to determine the correct week number.

DayOfWeek Calculation

The next line of the code calculates the DayOfWeek value, which is the day of the week that corresponds to January 4th of the year in question. This value is calculated by subtracting one day from January 4th and then using the DayOfWeek function to determine the day of the week.

YearStart Calculation

Date.Year

One of the key features of Power Query is the M language, which is used to define the data transformation steps. The M language is a functional programming language that is used to define data transformation queries. In this article, we are going to take a closer look at the M code behind the Power Query M function Date.Year.

What is the Date.Year Function?

The Date.Year function is a built-in function in Power Query that is used to extract the year from a given date. The function takes a date value as input and returns the year value as an integer. For example, the formula `Date.Year(#date(2021,1,1))` returns the value 2021.

Understanding the M Code Behind the Date.Year Function

When you use the Date.Year function in Power Query, you are actually using underlying M code to perform the data transformation. Let's take a look at the M code behind the Date.Year function.

```
let
  dateYear = (date) =>
    Date.Year(date)
in
  dateYear
```

The M code above defines a function called `dateYear`, which takes a date value as input and returns the year value. The function is defined using the `let` keyword, which is used to define local variables. In this case, we are defining a variable called `dateYear` that is a function.

The function `dateYear` is defined using an arrow function syntax, which is a shorthand way of defining functions in M. The function takes a single parameter called `date`, which represents the date value that is passed to the Date.Year function.

Inside the function, we simply call the built-in Date.Year function, passing the `date` parameter as input. This returns the year value for the given date.

Finally, the function is returned using the `in` keyword, which is used to define the output of the function.

Using the Date.Year Function in Power Query

DateTime.AddZone

The code for the DateTime.AddZone function is written in the M language, which is the primary programming language used by Power Query. In this article, we will explore the M code behind the DateTime.AddZone function and how it works.

Understanding the DateTime.AddZone Function

The DateTime.AddZone function takes three arguments: the datetime value to adjust, the number of hours to add or subtract, and the time zone to adjust for. The function returns a new datetime value that reflects the adjusted time and time zone.

The function works by first converting the datetime value to a UTC value, which is a standardized time format based on the Coordinated Universal Time (UTC) time zone. The function then adds or subtracts the specified number of hours to the UTC value and adjusts for the specified time zone.

Here is an example of how the DateTime.AddZone function works:

```
DateTime.AddZone(DateTime.LocalNow(), 2, “Pacific Standard Time”)
```

In this example, the function takes the current local datetime value, adds 2 hours, and adjusts for the Pacific Standard Time time zone. The function returns a new datetime value that reflects the adjusted time and time zone.

The M Code Behind the DateTime.AddZone Function

The M code for the DateTime.AddZone function is relatively straightforward. Here is the code for the function:

```
(DateTime as datetime, Hours as number, TimeZone as text) as datetime =>
  let
    UTCValue = DateTime.ToUtc(DateTime),
    AdjustedValue = DateTime.AddHours(UTCValue, Hours),
    TimeZoneValue = DateTimeZone.FromText(TimeZone),
    Result = DateTimeZone.ToLocal(AdjustedValue, TimeZoneValue)
  in
DateTime.Date
```

Introduction to DateTime.Date function

The `DateTime.Date` function is a part of the DateTime module in Power Query. It takes a datetime value as an input and returns a date value. The syntax for this function is as follows:

```
DateTime.Date(dateTime as datetime) as date
```

The `dateTime` parameter is the input datetime value that needs to be converted into a date value. The function returns the date portion of the input `dateTime` value.

Understanding the M code behind DateTime.Date

The M code behind the `DateTime.Date` function is quite simple. It uses the `Date.From` function to extract the date portion from the input datetime value. The M code for this function is as follows:

```
let
  Source = (dateTime as datetime) => Date.From(dateTime)
in
  Source
```

In the above M code, the `Source` variable is a function that takes a `dateTime` parameter and returns the output of the `Date.From` function. The `Date.From` function takes a datetime value as an input and returns the date portion of that value.

Examples of using DateTime.Date

Let's look at some examples of using the `DateTime.Date` function in Power Query.

Example 1 - Extracting date from a datetime column

Suppose we have a table `Sales` with a datetime column `DateSold`. We want to extract the date portion from this column and create

DateTime.FixedLocalNow

What is DateTime.FixedLocalNow?

The DateTime.FixedLocalNow function is an M function that returns a fixed date and time value in the local time zone. This function is useful when you need to add a timestamp to your data. It returns the current date and time in the local time zone, which is based on the time zone of the computer that is running the Power Query.

The syntax for the DateTime.FixedLocalNow function is as follows:

```
DateTime.FixedLocalNow()
```

This function does not take any arguments. It simply returns the current date and time in the local time zone.

Understanding the M Code Behind DateTime.FixedLocalNow

To understand the M code behind the DateTime.FixedLocalNow function, we need to break it down into its individual components. The M code for this function is as follows:

```
DateTime.FixedLocalNow = () =>
  DateTime.LocalNow() & “Z”,
  type text
```

Let's break down this code into its individual components:

- `DateTime.FixedLocalNow`: This is the name of the function. It is a user-defined function that has been created using the `=` operator.
- `() =>`: This is a lambda operator that is used to define the function. The `()` indicates that this function does not take any arguments.
- `DateTime.LocalNow()`: This is a built-in M function that returns the current date and time in the local time zone. It is used to get the current date and time value.

DateTime.From

In this article, we will dive into the M code behind the DateTime.From function and explain how it works. We will explore the syntax, parameters, and examples of this function to give you a better understanding of its capabilities.

Syntax of the DateTime.From Function

The syntax of the DateTime.From function is straightforward and follows the format:

```
DateTime.From(dateTimeValue as any) as nullable datetime
```

The function takes a single parameter, which is the dateTimeValue, and returns a nullable datetime value. The dateTimeValue parameter can be any value that can be converted into a datetime value, including text, numbers, and other datetime values.

Parameters of the DateTime.From Function

The dateTimeValue parameter of the DateTime.From function is the only parameter required for this function. However, it is essential to understand the format of the dateTimeValue parameter to ensure that the function works correctly.

The dateTimeValue parameter must be in one of the following formats:

- A datetime value, such as #2022-01-01 00:00:00#
- A text representation of a date and time in a recognizable format, such as "01/01/2022 12:00:00 AM"
- A number representing the number of days since December 30, 1899, and the time as a decimal fraction of a day

The DateTime.From function will automatically recognize the format of the dateTimeValue parameter and convert it into a datetime value.

Examples of the DateTime.From Function

Let's take a look at some examples to understand how the DateTime.From function works. Suppose we have a table that contains a column of dates in various formats:

Date
2022-01-01 00:00:00
01/02/2022 12:00:00 PM

DateTime.FromFileTime

Understanding File Time

Before discussing the M code behind the DateTime.FromFileTime function, it’s crucial to understand what file time is. A file time is a 64-bit value that represents the number of 100-nanosecond intervals since January 1, 1601 (UTC). It’s a Windows-based timestamp used to keep track of file and folder creation, access, and modification times.

The DateTime.FromFileTime function converts this file time value into a DateTime value, which is a standard format for date and time values.

The Syntax for DateTime.FromFileTime

The syntax for the DateTime.FromFileTime function is straightforward. The function takes the file time value as a parameter and returns the corresponding DateTime value.

DateTime.FromFileTime(fileTime as number) as datetime

The `fileTime` parameter is a 64-bit integer that represents a file time value. The function returns a DateTime value.

The M Code Behind DateTime.FromFileTime

The DateTime.FromFileTime function is a built-in function in Power Query M. It's a part of the DateTime module in M language that provides various functions related to date and time values.

The M code for the DateTime.FromFileTime function is as follows:

```
(FileTime as number) as nullable datetime =>
  let
    timeDiff = #duration(0, 0, 0, FileTime / 10000000 – #datetime(1601, 1, 1, 0, 0, 0)),
    result = #datetimezone(DateTime.LocalNow() – timeDiff, DateTimeZone.Local)
  in
    result
```

DateTime.FromText

What is the M language?

M is the language used in Power Query to write data transformation scripts. It is a functional programming language that is designed to be easy to learn and use. M is used to perform various data transformation tasks, such as filtering, sorting, and grouping. In addition, M has a wide range of built-in functions that make data transformation tasks easier.

Understanding DateTime.FromText

DateTime.FromText is a function used to convert text to a datetime value. The function syntax is:

```
DateTime.FromText(text as text, optional culture as nullable text)
```

The function takes two arguments:

- text: The text to be converted to a datetime value.
- culture: An optional argument that specifies the culture to be used for the conversion.

By default, the function uses the culture of the current system. However, you can also specify a custom culture to be used for the conversion.

The M code behind DateTime.FromText

To understand how DateTime.FromText works, let's examine the M code behind the function. The M code for DateTime.FromText is:

```
let
  Source = DateTimeZone.FromText(text, culture),
  Result = DateTime.LocalNow() + Duration.From(Source – DateTimeZone.LocalNow())
in
  Result
```

DateTime.IsInCurrentHour

In this article, we will take a closer look at the M code behind the Power Query M function DateTime.IsInCurrentHour.

Overview of DateTime.IsInCurrentHour

The DateTime.IsInCurrentHour function is a Power Query M function that returns a boolean value indicating whether a given datetime value falls within the current hour. The function takes a single parameter, datetime, which represents the datetime value to be tested. The syntax of the function is as follows:

```
DateTime.IsInCurrentHour(datetime as any) as logical
```

Here, datetime is any valid datetime value, and the function returns a logical value (true or false) indicating whether the datetime falls within the current hour.

The M Code Behind the Function

The M code behind the DateTime.IsInCurrentHour function is relatively simple. The function is implemented using the DateTime.LocalNow function, which returns the current date and time in the local time zone, and the DateTime.Hour function, which extracts the hour component of a datetime value.

Here is the M code for the DateTime.IsInCurrentHour function:

```
(DateTime.LocalNow() – DateTime.FromText(“00:00:00”))/#duration(0, 1, 0, 0) –
(DateTime.From(DateTime.LocalNow()))/#duration(0, 1, 0, 0) = 0
and
(DateTime.Hour(datetime) = DateTime.Hour(DateTime.LocalNow()))
```

Let's break down the code step by step:

1. The first line of the code calculates the number of minutes that have elapsed since midnight of the current day. This is done by

DateTime.IsInCurrentMinute

What is the DateTime.IsInCurrentMinute Function?

The DateTime.IsInCurrentMinute function is a built-in function in Power Query that returns a Boolean value indicating whether the specified date and time is in the current minute of the day. The syntax of the function is as follows:

```
DateTime.IsInCurrentMinute(dateTime as any) as logical
```

The function takes a single argument, which is a DateTime value representing the date and time to be checked. The function returns a logical value (true or false) indicating whether the specified date and time is in the current minute of the day.

The M Code Behind the DateTime.IsInCurrentMinute Function

The M code behind the DateTime.IsInCurrentMinute function is relatively simple. It uses the DateTime.LocalNow function to get the current date and time and then compares it to the specified date and time. Here is the M code:

```
(dateTime as any) as logical =>
  let
    CurrentDateTime = DateTime.LocalNow(),
    CurrentMinute = Time.Minute(CurrentDateTime),
    SpecifiedMinute = Time.Minute(dateTime)
  in
    CurrentMinute = SpecifiedMinute
```

The code takes the specified date and time as the `dateTime` parameter. It then uses the `let` keyword to declare two variables: `CurrentDateTime` and `CurrentMinute`. The `CurrentDateTime` variable gets the current date and time using the `DateTime.LocalNow()` function. The `CurrentMinute` variable uses the `Time.Minute()` function to extract the minute component

DateTime.IsInCurrentSecond

What is DateTime.IsInCurrentSecond?

DateTime.IsInCurrentSecond is a Power Query M function that checks whether a given datetime value falls within the current second. The function returns a boolean value of true if the datetime value falls within the current second, and false otherwise.

How to use DateTime.IsInCurrentSecond

The DateTime.IsInCurrentSecond function takes a single argument, which is a datetime value to be checked. Here's the syntax:

```
DateTime.IsInCurrentSecond(datetime as datetime) as logical
```

Let's take a look at an example. Suppose we have a table containing datetime values in the column "Date". We can use the DateTime.IsInCurrentSecond function to filter the table to only show records where the datetime value falls within the current second. Here's the M code:

```
let
  Source = Table.FromRows({{“2022-01-01T12:34:56.123”}, {“2022-01-01T12:34:57.456”}, {“2022-01-01T12:34:58.789”}}, {“Date”}),
  #”Changed Type” = Table.TransformColumnTypes(Source,{{“Date”, type datetime}}),
  #”Filtered Rows” = Table.SelectRows(#”Changed Type”, each DateTime.IsInCurrentSecond([Date]))
in
  #”Filtered Rows”
```

In this example, the Source step creates a table with three datetime values. The Changed Type step converts the "Date" column to the datetime data type. Finally, the Filtered Rows step uses the DateTime.IsInCurrentSecond function to filter the table to only show records where the datetime value falls within the current second.

Understanding the M code behind DateTime.IsInCurrentSecond

DateTime.IsInNextHour

What is DateTime.IsInNextHour Function?

The DateTime.IsInNextHour function is used to determine whether a given datetime value falls within the next hour. It returns a Boolean value that is true if the datetime value falls within the next hour, and false if it does not.

The syntax for DateTime.IsInNextHour function is:

```
DateTime.IsInNextHour(dateTime as any) as logical
```

The dateTime argument is a datetime value that you want to test.

The M Code Behind DateTime.IsInNextHour Function

Under the hood, the DateTime.IsInNextHour function is written in the M language, which is the programming language used in Power Query. The M language is a functional programming language that is easy to read and write.

The M code behind DateTime.IsInNextHour function is:

```
(dateTime) => DateTime.LocalNow() <= DateTime.LocalFromText(Text.Combine({Date.ToText(dateTime, "yyyy-MM-dd"), " ", Time.ToText(Time.AddHours(Time.FromText(Text.End(Text.FromText(Time.ToText(dateTime, "hh:mm:ss"), 8), 2)), 1), "hh:mm:ss")}), "yyyy-MM-dd hh:mm:ss")
```

Let's break down this code to understand how it works.

Step 1: Convert datetime to text

The first step in this function is to convert the given datetime value to text format using the Date.ToText and Time.ToText functions. We use these functions to extract the date and time components separately.

DateTime.IsInNextMinute

What is Power Query?

Before we dive in, let’s take a step back and talk about what Power Query is. Power Query is a data connection technology that allows you to extract data from various sources, transform that data into the format you need, and load it into your destination. It’s a part of the Microsoft Power BI suite, but it also comes built into Excel and can be used as a standalone tool.

Power Query uses a functional language called M to perform transformations on your data. M is a powerful and flexible language that allows you to write complex transformations with ease.

DateTime.IsInNextMinute

DateTime.IsInNextMinute is a function that checks whether a given datetime value is in the next minute. The function takes a single argument, which is the datetime value you want to check.

Here’s an example of how to use DateTime.IsInNextMinute:

```
let
  dateTimeValue = #datetime(2022, 1, 1, 12, 34, 56),
  result = DateTime.IsInNextMinute(dateTimeValue)
in
  result
```

In this example, we're checking whether the datetime value `2022-01-01 12:34:56` is in the next minute. The result of this function call would be `true`, since the datetime value is in the next minute.

The M Code Behind DateTime.IsInNextMinute

Now that we've covered the basics of the DateTime.IsInNextMinute function, let's take a look at the M code behind it.

```
let
  isDateTimeInNextMinute = (dateTimeValue as datetime) as logical =>
```

DateTime.IsInNextNHours

What is the DateTime.IsInNextNHours function?

The DateTime.IsInNextNHours function is a useful function in Power Query that allows users to filter data based on a specific timeframe. This function returns a Boolean value, which is true if the date and time fall within the specified timeframe.

The function takes two arguments: the first argument is a column containing date and time information, and the second argument is the number of hours for which you want to check. For example, if you want to check whether a date and time fall within the next 3 hours, you would use the following formula:

```
= DateTime.IsInNextNHours([DateTimeColumn], 3)
```

How does the function work?

The DateTime.IsInNextNHours function works by comparing the date and time in the specified column to the current date and time plus the specified number of hours. If the date and time in the column fall within the specified timeframe, the function returns true. Otherwise, it returns false.

Here's the M code behind the DateTime.IsInNextNHours function:

```
(DateTimeColumn as any, Hours as number) =>
  let
    CurrentDateTime = DateTime.LocalNow(),
    EndDateTime = CurrentDateTime + #duration(0, Hours, 0, 0),
    Result = DateTimeColumn >= CurrentDateTime and DateTimeColumn <= EndDateTime
  in
    Result
```

As you can see, the function first gets the current date and time using the DateTime.LocalNow function. It then adds the specified number of hours to the current date and time using the #duration function, which returns a duration value.

Finally, the function compares the date and time in the specified column to the current date and time plus the specified duration value. If the date and time fall within the specified timeframe, the function returns true. Otherwise, it returns false.

Examples of using the DateTime.IsInNextNHours function

Let's take a look at some examples of using the DateTime.IsInNextNHours function in Power Query.

Example 1: Filtering data based on the next 3 hours

Suppose you have a table containing a column called "DateTime" that contains date and time information. You want to filter the table to

DateTime.IsInNextNMinutes

The M code behind this function is relatively straightforward, but it can be useful to understand how it works if you want to modify it or create your own custom date and time functions in Power Query M.

The Function Syntax

Before we dive into the M code behind DateTime.IsInNextNMinutes, let's take a look at the function syntax:

```
DateTime.IsInNextNMinutes(dateTime as any, minutes as number) as logical
```

This function takes two arguments:

- `dateTime`: The date and time value that you want to check.
- `minutes`: The number of minutes from the current date and time that you want to check.

The function returns a logical value (`true` or `false`) depending on whether the specified date and time value falls within the specified number of minutes from the current date and time.

The M Code

Now let's take a look at the M code behind this function:

```
(dateTime – DateTime.LocalNow()) / Duration.FromMinutes(1) < minutes
```

This code may look a bit intimidating at first, but it's actually quite simple. Let's break it down:

- `DateTime.LocalNow()` returns the current date and time value.
- `dateTime - DateTime.LocalNow()` subtracts the current date and time value from the specified date and time value, resulting in the time difference between the two values.
- `(dateTime - DateTime.LocalNow()) / Duration.FromMinutes(1)` converts the time difference to minutes.
- `< minutes` checks if the time difference is less than the specified number of minutes.

DateTime.IsInNextNSeconds

Understanding DateTime.IsInNextNSeconds

DateTime.IsInNextNSeconds is a Power Query M function that takes two arguments: a datetime value and a number of seconds. It returns a Boolean value indicating whether the datetime value falls within the specified number of seconds in the future. This function is particularly useful when working with time-critical data, such as stock prices, weather data, or sensor readings.

The syntax for the DateTime.IsInNextNSeconds function is as follows:

```
DateTime.IsInNextNSeconds(dateTime as any, seconds as number) as logical
```

Here, dateTime is the date and time value to be evaluated, and seconds is the number of seconds in the future to check for. The function returns a logical value (true or false) indicating whether the dateTime value falls within the specified number of seconds in the future.

The M code behind DateTime.IsInNextNSeconds

The M code behind the DateTime.IsInNextNSeconds function is complex, but it is well-optimized to provide fast and accurate results. The function works by first converting the dateTime value to a Unix timestamp, which is the number of seconds since January 1, 1970. It then adds the number of seconds specified in the seconds argument to this timestamp, and converts the resulting timestamp back to a datetime value.

Here is the M code for the DateTime.IsInNextNSeconds function:

```
(datetime as any, seconds as number) =>
let
  targetTimestamp = DateTimeZone.ToLocal(DateTimeZone.UtcNow()) + #duration(0, 0, seconds),
  inputTimestamp = DateTimeZone.ToLocal(datetime),
  diff = DateTimeZone.ToLocal(targetTimestamp) – inputTimestamp,
  result = diff.TotalSeconds >= 0 and diff.TotalSeconds <= seconds
in
DateTime.IsInNextSecond
```

One of the functions available in Power Query M language is the DateTime.IsInNextSecond function. This function checks if a given date/time value falls within the next second. In this article, we will explore the M code behind this function and how it can be useful in data analysis.

Syntax and Arguments

The syntax for the DateTime.IsInNextSecond function is as follows:

```
DateTime.IsInNextSecond(dateTime as any) as logical
```

The function takes a single argument, dateTime, which can be any valid date/time value. The function returns a logical value indicating whether the given dateTime falls within the next second.

M Code Explanation

The DateTime.IsInNextSecond function is implemented in M code as follows:

```
(dateTime as any) as logical =>
  let
    nextSecond = DateTime.LocalNow() + #duration(0,0,1),
    result = dateTime >= DateTime.LocalNow() and dateTime < nextSecond
  in
    result
```

The code begins by defining a parameter, dateTime, which represents the input date/time value.

Next, the code creates a variable called nextSecond, which is calculated by adding one second (#duration(0,0,1)) to the current local date/time (DateTime.LocalNow()). This will give us the date/time for the start of the next second.

DateTime.IsInPreviousHour

Syntax of DateTime.IsInPreviousHour Function

The DateTime.IsInPreviousHour function takes in a single parameter, which is the date and time value that you want to check. The syntax for this function is as follows:

```
DateTime.IsInPreviousHour(dateTime as any) as logical
```

Here, the dateTime parameter represents the date and time value that you want to check. The function returns a logical value, which is true if the provided value falls within the previous hour, and false otherwise.

Understanding the M Code behind DateTime.IsInPreviousHour Function

The DateTime.IsInPreviousHour function is a combination of several M functions that perform various calculations to determine whether the given date and time value falls within the previous hour. Let's break down the M code behind this function to understand how it works.

```
(dateTime) =>
let
  timeDifference = DateTime.LocalNow() – dateTime,
  seconds = Duration.TotalSeconds(timeDifference),
  isInHour = seconds >= 0 and seconds <= 3600
in
  isInHour
```

The code above shows the M code behind the DateTime.IsInPreviousHour function. Let's go through each line of the code to understand how it works.

DateTime.IsInPreviousMinute

What is DateTime.IsInPreviousMinute Function?

The DateTime.IsInPreviousMinute function is a Power Query M function used to determine if a given date and time value falls in the previous minute relative to the current date and time. It takes a single argument, which is a datetime value that needs to be evaluated. The function returns a boolean value, which is true if the datetime value falls in the previous minute and false otherwise.

Syntax

The syntax for the DateTime.IsInPreviousMinute function is as follows:

```
DateTime.IsInPreviousMinute(dateTime as any) as logical
```

The function takes a single argument, dateTime, which can be any valid datetime value.

How Does the Function Work?

The DateTime.IsInPreviousMinute function works by comparing the given datetime value with the current datetime value. It calculates the time difference between the two values and checks if the difference is less than or equal to one minute. If the time difference is less than or equal to one minute, the function returns true; otherwise, it returns false.

Example

Suppose we have a datetime value of "2022-01-01 12:30:45". To check if this value falls in the previous minute, we can use the DateTime.IsInPreviousMinute function as follows:

```
DateTime.IsInPreviousMinute(#datetime(2022, 1, 1, 12, 30, 45))
```

This will return false because the datetime value is exactly on the minute, and the function checks if the value falls in the previous minute.

If we change the datetime value to "2022-01-01 12:29:45", the function will return true because the value falls in the previous minute

DateTime.IsInPreviousNHours

To understand how this function works, it’s important to take a closer look at the M code behind it. In this article, we’ll break down the M code used by DateTime.IsInPreviousNHours and explore some examples of how it can be used in practice.

Understanding the M Code

The M code behind DateTime.IsInPreviousNHours is relatively straightforward. It uses the DateTime.LocalNow function to retrieve the current date and time, and then subtracts a specified number of hours from that value. This creates a new datetime value that can be used to filter data.

Here’s the M code used by DateTime.IsInPreviousNHours:

```
DateTime.IsInPreviousNHours = (datetime as any, hours as number) =>
  datetime >= DateTime.LocalNow() – #duration(0, hours, 0, 0)
```

Let's break down what's happening in this code:

- "DateTime.IsInPreviousNHours" is the name of the function we're defining.
- "(datetime as any, hours as number)" is the list of arguments that the function takes. In this case, we're expecting a datetime value and a number of hours.
- "datetime >= DateTime.LocalNow() - #duration(0, hours, 0, 0)" is the logic of the function. We're checking whether the datetime value is greater than or equal to the current datetime minus the specified number of hours.

So, if we were to call DateTime.IsInPreviousNHours with a datetime value of "10/1/2021 3:00 PM" and a number of hours of "2", the function would return "true" if the datetime falls within the range of "10/1/2021 1:00 PM" to "10/1/2021 3:00 PM".

Examples of DateTime.IsInPreviousNHours in Practice

Now that we understand the M code behind DateTime.IsInPreviousNHours, let's explore some examples of how it can be used in practice.

Example 1: Filtering Data based on the Last 24 Hours

Suppose we have a table of customer orders that includes a "Timestamp" column indicating when each order was placed. We want to filter this table to show only orders that were placed within the last 24 hours.

DateTime.IsInPreviousNMinutes

Overview of the DateTime.IsInPreviousNMinutes Function

The DateTime.IsInPreviousNMinutes function is used to filter data based on a specific time frame. It takes two parameters: the column that contains the date and time values, and the number of minutes to consider. The function returns a Boolean value of true or false, depending on whether the value in the column is within the specified time frame.

The function is useful when you want to extract data that falls within a certain time frame, such as the last hour or the last day. For example, you may want to extract all the sales data from the last hour to see how your business is performing in real-time.

The M Code Behind the Function

The M code behind the DateTime.IsInPreviousNMinutes function is quite simple. It uses the DateTime.LocalNow function to get the current date and time, subtracts the number of minutes specified by the second parameter, and compares the result to the date and time value in the specified column.

Here is the M code for the function:

```
(Table as table, ColumnName as text, Minutes as number) =>
  let
    Source = Table.SelectRows(Table, each [ColumnName] >= DateTime.LocalNow() – #duration(0,0,Minutes,0))
  in
    Source
```

The function takes three parameters: the table that contains the data, the column name that contains the date and time values, and the number of minutes to consider. It then uses the Table.SelectRows function to filter the data based on the specified time frame.

Using the Function in Power Query

To use the DateTime.IsInPreviousNMinutes function in Power Query, you need to create a new custom function. Here are the steps:

1. Open the Power Query Editor by clicking on the Edit Queries button in the Home tab of Excel.
2. Click on the New Source button and select Blank Query.
3. In the Query Editor, click on the View tab and select Advanced Editor.

DateTime.IsInPreviousNSeconds

Understanding the DateTime.IsInPreviousNSeconds Function

The DateTime.IsInPreviousNSeconds function allows users to filter data based on whether or not a date and time value falls within a specified number of seconds before the current date and time. For example, if we wanted to filter a list of orders to only show those that were placed within the last 10 minutes, we could use the DateTime.IsInPreviousNSeconds function.

The function takes two arguments: the first is the column containing the date and time values, and the second is the number of seconds to look back. Here is the syntax for the function:

```
DateTime.IsInPreviousNSeconds(dateTimeColumn as any, seconds as number) as any
```

Let's take a look at an example. Suppose we have a table of sales data with a column named "Order Date" that contains date and time values. We want to create a new table that only includes orders that were placed within the last 5 minutes. We can use the following M code:

```
let
  Source = Sales_Data,
  FilteredRows = Table.SelectRows(Source, each DateTime.IsInPreviousNSeconds([Order Date], 300))
in
  FilteredRows
```

In this code, we first define the source table as "Sales_Data". We then create a new table called "FilteredRows" by using the Table.SelectRows function to filter the data based on the DateTime.IsInPreviousNSeconds function. We pass in the "Order Date" column as the first argument and 300 (which is 5 minutes in seconds) as the second argument.

Breaking Down the M Code

DateTime.IsInPreviousSecond

Understanding the DateTime.IsInPreviousSecond Function

Before we dive into the M code behind DateTime.IsInPreviousSecond, let’s first explore what this function does. The DateTime.IsInPreviousSecond function takes a single argument, a datetime value, and returns a boolean value indicating whether the datetime value is in the previous second.

This function is particularly useful when working with streaming data, where new data is constantly being added to a dataset. By using DateTime.IsInPreviousSecond, you can quickly determine if the new data point is within the previous second, allowing you to aggregate the data in real-time.

Breaking Down the M Code Behind DateTime.IsInPreviousSecond

To understand how the DateTime.IsInPreviousSecond function works, we need to first look at the M code behind it. The M code for this function is as follows:

```
let
  IsInPreviousSecond = (datetime as datetime) =>
    let
      PrevSecond = DateTime.LocalNow() – #duration(0, 0, 0, 1),
      Result = datetime > PrevSecond and datetime <= DateTime.LocalNow()
    in
      Result
in
  IsInPreviousSecond
```

Let's break down this code into its individual components:

- The first line defines a function called "IsInPreviousSecond", which takes a single argument, a datetime value.
- The next line defines a variable called "PrevSecond", which is set to the current local date and time minus one second.
- The third line defines a variable called "Result", which checks if the datetime value is greater than the previous second and less than or

DateTime.LocalNow

What is DateTime.LocalNow?

Before we dive into the M code, let’s review what DateTime.LocalNow actually does. This function returns the current date and time in the user’s local time zone. For example, if you’re in New York and run DateTime.LocalNow, you’ll get the current date and time in Eastern Standard Time.

The M Code

The M code for DateTime.LocalNow is relatively simple. Here’s what it looks like:

```
() => DateTimeZone.LocalNow()
```

Let's break this down piece by piece.

The Anonymous Function

The code starts with an anonymous function, denoted by the parentheses with nothing inside. An anonymous function is simply a function without a name – it's defined on the spot and is only used once. In this case, the anonymous function is used to call the DateTimeZone.LocalNow function.

The DateTimeZone.LocalNow Function

The DateTimeZone.LocalNow function is what actually returns the current date and time in the user's local time zone. This function is part of the DateTimeZone type, which provides various functions for working with dates and times in different time zones.

What's the Point of Using an Anonymous Function?

You might be wondering why the M code for DateTime.LocalNow uses an anonymous function at all. After all, couldn't we just call DateTimeZone.LocalNow directly?

The reason for the anonymous function is that it allows us to delay the execution of the DateTimeZone.LocalNow function until it's actually needed. In other words, the code inside the anonymous function doesn't get executed until the function is called.

This might not seem like a big deal, but it can actually be quite useful in certain scenarios. For example, if you have a query that's pulling in data from multiple sources and doing a lot of transformations, you might not want to waste time calculating the current date and time until the very end of the query. By using an anonymous function, you can delay the execution of DateTimeZone.LocalNow until the final

DateTime.Time

Understanding DateTime.Time Function

Before we look at the M code, let’s first understand the DateTime.Time function. This function is used to extract the time component from a datetime value. The syntax of this function is as follows:

```
DateTime.Time(dateTime as any) as any
```

The parameter `dateTime` can be any valid datetime value. The function returns the time component of the datetime value as a time value.

The M Code Behind DateTime.Time Function

Now that we know what the DateTime.Time function does, let's see how it is implemented in M code.

The M code for the DateTime.Time function is as follows:

```
(dateTime as any) =>
Time.From(dateTime – Date.From(dateTime))
```

Let's break down this code into smaller parts and see what each part does.

Part 1: `(dateTime as any) =>`

This part defines a lambda function that takes a single parameter `dateTime` of any data type.

Part 2: `Time.From(`

This part calls the `Time.From` function, which converts a datetime value to a time value.

Part 3: `dateTime - Date.From(dateTime)`

This part calculates the time component of the datetime value. It does this by subtracting the date component of the datetime value from the datetime value itself. The `Date.From` function is used to extract the date component of the datetime value.

DateTime.ToRecord

Understanding the DateTime.ToRecord Function

The DateTime.ToRecord function is used to convert a datetime value into a record format. This function takes a single argument, which is the datetime value that we want to convert. The output of this function is a record that contains separate fields for year, month, day, hour, minute, second, and millisecond values.

Here's an example of how we can use the DateTime.ToRecord function:

```
let
  myDateTime = #datetime(2022, 1, 1, 10, 30, 0),
  recordValue = DateTime.ToRecord(myDateTime)
in
  recordValue
```

In this example, we first create a datetime value using the #datetime function. We then pass this value to the DateTime.ToRecord function, which returns a record containing the individual date and time components.

Dive into the M Code

Now let's take a closer look at the M code behind this function. Here's the code for the DateTime.ToRecord function:

```
(DateTimeValue as datetime) =>
  #datetimezone(
    Date.Year(DateTimeValue),
    Date.Month(DateTimeValue),
    Date.Day(DateTimeValue),
    Time.Hour(DateTimeValue),
    Time.Minute(DateTimeValue),
```

DateTime.ToText

Understanding the DateTime.ToText Function

The DateTime.ToText function converts a date or time value to a text format using a specified format string. The syntax for the function is as follows:

DateTime.ToText(dateTime as any, optional format as nullable text, optional culture as nullable text) as text

The dateTime parameter is the date or time value that needs to be converted. The format parameter is an optional argument that specifies the format of the output text. If this parameter is not specified, the function uses the default format. The culture parameter is also an optional argument that specifies the culture to use for formatting the text.

Here is an example of how the function can be used:

DateTime.ToText(#datetime(2022,1,1,0,0,0), “dd/MM/yyyy”)

This code will return the text string "01/01/2022". The function has used the specified format string to convert the date into the desired text format.

The M Code Behind the DateTime.ToText Function

The M code behind the DateTime.ToText function is responsible for performing the conversion from a date or time value to a text format. The code is written in the M language, which is used by Power Query to perform data transformations.

The DateTime.ToText function is defined in the M code as follows:

```
let
   DateTime.ToText = (dateTime as any, optional format as nullable text, optional culture as nullable text) =>
DB2.Database
```

In this article, we will dive into the M code behind the DB2.Database function and understand how it works.

Understanding the DB2.Database Function

The DB2.Database function is used to connect to a DB2 database and retrieve data. The function takes three parameters:

1. Server: The name of the DB2 server.
2. Database: The name of the DB2 database.
3. Options: A record that contains additional connection options.

Here is an example of how to use the DB2.Database function in Power Query M:

```
let
  Source = DB2.Database(“myDB2Server”, “myDB2Database”, [Query=”SELECT  FROM myTable”])
in
  Source
```

This code connects to the DB2 server "myDB2Server" and retrieves all the data from the "myTable" table in the "myDB2Database" database.

The M Code behind the DB2.Database Function

Under the hood, the DB2.Database function is made up of several M functions that work together to establish a connection and retrieve data from the database.

Here is the M code that makes up the DB2.Database function:

```
let
  Connect = (server, database, options) =>
    let
      connectionString = “Provider=IBMDADB2;Database=” & database & “;Hostname=” & server & “;” & options,
Decimal.From
```

In this article, we will explore the M code behind the Decimal.From function and demonstrate how it can be used to convert text values to decimal numbers in Power Query.

Understanding the Decimal.From Function

The Decimal.From function in Power Query M is used to convert a text value to a decimal number. This can be particularly useful when working with data that contains numeric values stored as text.

The syntax for the Decimal.From function is as follows:

```
Decimal.From(text as text, optional culture as nullable text) as nullable number
```

The function takes two parameters. The first parameter, "text," is the text value that you want to convert to a decimal number. The second parameter, "culture," is an optional parameter that specifies the culture to use when converting the text value to a decimal number. If the culture parameter is not specified, the function will use the current culture of the system.

Using Decimal.From in Power Query

To use the Decimal.From function in Power Query, you can follow these steps:

1. Open the Power Query Editor in Excel.
2. Select the column containing the text values that you want to convert to decimal numbers.
3. Click on the "Transform Data" button in the "Home" tab to open the Power Query Editor.
4. In the Power Query Editor, click on the "Add Column" tab and select "Custom Column" from the dropdown menu.
5. In the "Custom Column" dialog box, enter a name for the new column.
6. In the "Custom Column" dialog box, enter the following expression:

```
Decimal.From([Text Column])
```

```
Diagnostics.ActivityId
```

Understanding Diagnostics.ActivityId

Before we dive into the M code, let's first understand what Diagnostics.ActivityId is and how it works. In simple terms, Diagnostics.ActivityId is a function that generates a unique identifier for each query execution. This identifier can be used to track the execution of the query and diagnose any issues that might arise.

When you use Diagnostics.ActivityId in your Power Query M code, it generates a GUID (Globally Unique Identifier) that represents the current query execution. This GUID is then stored in the Query Diagnostics table, which is generated automatically by Power Query. You can access this table to view the GUIDs and track the execution of the queries.

The M Code Behind Diagnostics.ActivityId

Now that we understand what Diagnostics.ActivityId is and how it works, let's take a closer look at the M code behind it. When you use Diagnostics.ActivityId in your Power Query M code, it generates the following M code:

`` ` = let activityId = Text.NewGuid() in activityId ` ``

As you can see, the M code generates a new GUID using the Text.NewGuid() function and assigns it to the variable activityId. This variable is then returned by the function, which allows you to use it in your M code.

Using Diagnostics.ActivityId

Now that we understand the M code behind Diagnostics.ActivityId, let's explore some practical use cases for this function. Here are some examples of how you can use Diagnostics.ActivityId:

Troubleshooting

One of the primary uses of Diagnostics.ActivityId is troubleshooting. When you encounter an issue with your query, you can use the GUID generated by Diagnostics.ActivityId to track the execution of the query and diagnose the issue.

To do this, you can filter the Query Diagnostics table based on the GUID generated by Diagnostics.ActivityId. This will show you all the steps in the query execution that led up to the issue, allowing you to pinpoint the problem.

Auditing

If you need to audit your query execution, you can use the GUID generated by Diagnostics.ActivityId to track when and how the query was executed. You can store this information in a database or log file for future reference.

Parallelism

Diagnostics.ActivityId can also be useful when working with parallel query execution. When multiple queries are executed in parallel, each query will generate a unique GUID using Diagnostics.ActivityId. This allows you to track the execution of each query independently

Diagnostics.Trace

In this article, we will take a closer look at the M code behind the Power Query M function Diagnostics.Trace. We will explore how this function works, and how it can help you to diagnose and solve complex data processing problems.

Understanding Diagnostics.Trace

The Diagnostics.Trace function is a built-in M function that allows you to log messages, warnings, and errors during the data processing pipeline. This function is especially useful when working with large and complex data sets, where it can be difficult to track down issues and errors.

The basic syntax for the Diagnostics.Trace function is:

```
Diagnostics.Trace([Message], [Severity])
```

The [Message] parameter is the message that you want to log, and the [Severity] parameter is the severity level of the message. There are three severity levels that you can use:

- Error
- Warning
- Information

By default, Diagnostics.Trace logs messages with the Information severity level. However, you can specify a different severity level by providing a value from the Severity enumeration:

- Severity.Error
- Severity.Warning
- Severity.Information

How Diagnostics.Trace Works

The Diagnostics.Trace function works by logging messages to the Power Query trace log. When you execute a query in Power Query, the trace log records all of the M functions that are called, along with their input and output values.

By using the Diagnostics.Trace function, you can add custom log messages to the trace log. These messages can provide valuable insight into the data processing pipeline, and help you to identify and resolve issues more quickly.

DirectQueryCapabilities.From

In this article, we will delve into the M code behind this function, exploring its syntax, common use cases, and tips for optimizing its performance.

Syntax

The syntax for `DirectQueryCapabilities.From` is relatively simple. It takes in a single argument, which is a record that defines the capabilities of a DirectQuery source. The record has the following fields:

– `SupportsTop`: A boolean field that indicates whether the DirectQuery source supports the Top clause.
– `SupportsGroupBy`: A boolean field that indicates whether the DirectQuery source supports the Group By clause.
– `SupportsOrderBy`: A boolean field that indicates whether the DirectQuery source supports the Order By clause.
– `SupportsAggregates`: A boolean field that indicates whether the DirectQuery source supports aggregate functions such as SUM, AVG, MIN, and MAX.
– `SupportsJoin`: A boolean field that indicates whether the DirectQuery source supports Join operations.
– `SupportsInnerJoin`: A boolean field that indicates whether the DirectQuery source supports Inner Join operations.
– `SupportsLeftOuterJoin`: A boolean field that indicates whether the DirectQuery source supports Left Outer Join operations.
– `SupportsRightOuterJoin`: A boolean field that indicates whether the DirectQuery source supports Right Outer Join operations.
– `SupportsFullOuterJoin`: A boolean field that indicates whether the DirectQuery source supports Full Outer Join operations.
– `SupportsCrossJoin`: A boolean field that indicates whether the DirectQuery source supports Cross Join operations.
– `SupportsCount`: A boolean field that indicates whether the DirectQuery source supports the Count function.
– `SupportsDistinctCount`: A boolean field that indicates whether the DirectQuery source supports the Distinct Count function.
– `SupportsPercentile`: A boolean field that indicates whether the DirectQuery source supports the Percentile function.
– `SupportsMedian`: A boolean field that indicates whether the DirectQuery source supports the Median function.
– `SupportsMode`: A boolean field that indicates whether the DirectQuery source supports the Mode function.
– `SupportsVariance`: A boolean field that indicates whether the DirectQuery source supports the Variance function.
– `SupportsStandardDeviation`: A boolean field that indicates whether the DirectQuery source supports the Standard Deviation function.

Common Use Cases

`DirectQueryCapabilities.From` is often used when defining the capabilities of a custom data connector or when troubleshooting issues with an existing connector. By passing in a record that accurately describes the capabilities of the DirectQuery source, you can

Double.From

The Double.From function is written in the M language, which is the language used by Power Query to control the data transformation process. Understanding the M code behind the Double.From function can be helpful when working with large or complex datasets.
Syntax of the Double.From Function
The syntax of the Double.From function is as follows:

Double.From(text as text, optional culture as nullable text) as nullable number

The Double.From function takes two arguments:
- `text`: The text value to convert to a double.
- `culture`: An optional argument that specifies the culture to use when converting the text value to a double.
The function returns a nullable number value.
Understanding the M Code Behind the Double.From Function
The M code behind the Double.From function can be broken down into three parts:
1. Input validation and error handling
2. Parsing the text value to a double
3. Returning the double value
Input Validation and Error Handling
The input validation and error handling section of the code ensures that the input value is valid and that no errors occur during the conversion process. The code checks if the input value is null or blank and returns a null value if true. It also checks if the text value can be converted to a double and returns an error if it cannot.
Parsing the Text Value to a Double
The code then proceeds to parse the text value to a double. It first checks if the culture argument has been specified. If it has, the code uses that culture to parse the text value to a double. If the culture argument has not been specified, the code uses the default culture of the system.
The code uses the NumberStyles.Float flag to indicate that the text value is a floating-point value. It also uses the CultureInfo object to

Duration.Days

Understanding the Duration.Days Function

The Duration.Days function is a simple yet powerful function that calculates the number of days between two given dates. The function takes two arguments, start and end dates, and returns the number of days between them. Here's the syntax of the Duration.Days function:

```
Duration.Days(startDate as date, endDate as date) as number
```

The function takes two date values, start date and end date, and returns a number that represents the number of days between the two dates.

The M Code Behind the Duration.Days Function

Under the hood, the Duration.Days function is implemented using the M language, which is the native language of Power Query. The M language is a functional programming language that is used to define the data transformation logic in Power Query. Let's take a look at the M code behind the Duration.Days function:

```
(Duration as duration) =>
   Duration.Days(Duration)
```

The code above defines a function that takes a duration value and returns the number of days represented by the duration value. The Duration.Days function is called to calculate the number of days between the start and end dates.

Understanding Duration Values

Before we dive deeper into the M code behind the Duration.Days function, let's take a moment to understand what duration values are. In Power Query, a duration value represents a time span. A duration value is defined using the Duration.FromText function, which takes a text value as input and returns a duration value. Here's the syntax of the Duration.FromText function:

```
Duration.From
```

What is Duration.From?

Duration.From is an M function that takes a decimal number as input and returns a duration value. The duration is expressed in days, hours, minutes, and seconds. The syntax of Duration.From is as follows:

```
Duration.From(number as any) as duration
```

Here, the argument "number" is the decimal number that you want to convert to a duration.

How Does Duration.From Work?

To understand how Duration.From works, let's take an example. Suppose you have a decimal number 1.5, which represents 1 day and 12 hours. When you pass this number to Duration.From, it will return a duration value of "1 day, 12 hours". Here is the M code for this:

```
Duration.From(1.5)
```

The output will be:

```
#duration(0, 36, 0, 0)
```

This output may seem a bit cryptic, but it can be easily decoded. The first argument of the #duration function is the number of days, the second argument is the number of hours, the third argument is the number of minutes, and the fourth argument is the number of seconds. So, the output "#duration(0, 36, 0, 0)" means "0 days, 36 hours, 0 minutes, and 0 seconds".

How to Convert Duration to Other Units?

Duration.FromText

Understanding the Duration.FromText Function

The Duration.FromText function is used to convert a text value to a duration value. The function takes a single argument, which is the text value that you want to convert. The text value must be in a format that can be recognized as a duration value, such as “1 day 5 hours 30 minutes”.

The Duration.FromText function returns a duration value in the format of “hh:mm:ss” or “d.hh:mm:ss”. For example, if you pass the text value “5 hours 30 minutes” to the function, it will return a duration value of “05:30:00”. If you pass the text value “1 day 6 hours 15 minutes” to the function, it will return a duration value of “30:15:00”.

The M Code Behind the Duration.FromText Function

The M code behind the Duration.FromText function is relatively simple. The function is defined as follows:

```
Duration.FromText = (text as text) as duration =>
let
  hours = try Number.FromText(Text.BetweenDelimiters(text,””, ” hours”), 0) otherwise 0,
  minutes = try Number.FromText(Text.BetweenDelimiters(text, “”, ” minutes”), 0) otherwise 0,
  seconds = try Number.FromText(Text.BetweenDelimiters(text, “”, ” seconds”), 0) otherwise 0,
  days = try Number.FromText(Text.BetweenDelimiters(text, “”, ” days”), 0) otherwise 0,
  duration = #duration(days, hours, minutes, seconds)
in
  duration
```

Let's break down this code into its individual parts to understand how it works.

Defining the Function

The first line of the code defines the function name and its arguments:

```
Duration.Hours
```

What is the Duration.Hours function?

The Duration.Hours function is a Power Query function that returns the number of hours in a duration value. The function is used to extract the number of hours from a time duration value, which could be useful in a variety of scenarios. For example, you might use the function to calculate the number of hours worked by an employee, or the duration of a movie.

The M code behind Duration.Hours

The M code behind the Duration.Hours function is relatively simple. The function takes a single argument, which is a duration value, and returns the number of hours in the duration. Here is the M code for the function:

```
(Duration) =>
Duration.TotalHours
```

The code above defines an anonymous function that takes a single argument, which is a duration value. The function then returns the number of hours in the duration value using the TotalHours property of the duration.

Using Duration.Hours in Power Query

Now that we understand the M code behind the Duration.Hours function, let's take a look at how it can be used in Power Query. Suppose we have a table with a column of duration values, and we want to create a new column that contains the number of hours in each duration value. We can do this using the following steps:

1. Select the duration column in the table.
2. Click the "Add Column" tab and select "Custom Column".
3. In the "Custom Column" dialog box, enter a name for the new column and the following formula:

```
  Duration.Hours([DurationColumn])
```

Duration.Minutes

Understanding Duration.Minutes

Duration.Minutes is a function in Power Query that calculates the number of minutes in a duration value. A duration value is a time span that represents the difference between two dates or times. For example, if you have two dates: 1/1/2021 and 1/2/2021, the duration value would be 1 day. The Duration.Minutes function can then be used to calculate the number of minutes in that duration value.

The M Code Behind Duration.Minutes

To understand the M code behind Duration.Minutes, let's first take a look at the syntax of the function:

```
Duration.Minutes(duration as duration) as number
```

The function takes one argument, which is the duration value that you want to calculate the number of minutes for. The argument is of type "duration" and the function returns a number.

Now, let's break down the M code for the function:

```
(duration as duration) as number =>
Duration.TotalMinutes(duration)
```

The first line defines the argument for the function, which is the duration value we want to calculate the number of minutes for. The second line uses the Duration.TotalMinutes function to calculate the total number of minutes in the duration value. The result is then returned as a number.

Using Duration.Minutes

Now that we understand the M code behind Duration.Minutes, let's take a look at how we can use it in Power Query. Suppose we have a table that contains a column of duration values, and we want to calculate the total number of minutes for each row. We can do this by adding a new column to the table and using the Duration.Minutes function:

Duration.Seconds

Overview of Duration.Seconds function

The Duration.Seconds function is a built-in function in Power Query that calculates the duration between two dates or times in seconds. It takes two arguments: start and end, which are the starting and ending dates or times respectively. The function returns the duration in seconds as a decimal number.

Syntax of Duration.Seconds function

The syntax of the Duration.Seconds function is as follows:

```
Duration.Seconds(start as any, end as any) as any
```

The start and end arguments can be any data type that represents a valid date or time. The function returns a decimal number that represents the duration between the start and end arguments in seconds.

Example usage of Duration.Seconds function

Here is an example that demonstrates the usage of the Duration.Seconds function:

```
let
  startDate = #datetime(2022, 1, 1, 0, 0, 0),
  endDate = #datetime(2022, 1, 1, 0, 0, 10),
  durationInSeconds = Duration.Seconds(startDate, endDate)
in
  durationInSeconds
```

In this example, we define the start and end dates as #datetime values that represent January 1, 2022, at 12:00:00 AM and January 1, 2022, at 12:00:10 AM respectively. We then pass these values to the Duration.Seconds function, which calculates the duration between

Duration.ToRecord

What is the Duration.ToRecord function?

The Duration.ToRecord function is used to convert a duration value into a record. A duration represents a length of time, such as 2 hours and 30 minutes. A record is a collection of key-value pairs, where each key represents a field name and each value represents the data in that field.

So, the Duration.ToRecord function takes a duration value and converts it into a record with two fields: “Hours” and “Minutes”. The “Hours” field contains the number of hours in the duration, while the “Minutes” field contains the remaining minutes.

Using the Duration.ToRecord function

To use the Duration.ToRecord function, you’ll need a duration value to convert. This can be a column in a table, a variable, or a literal value. Here’s an example:

```
let
  durationValue = #duration(0, 2, 30, 0),
  recordValue = Duration.ToRecord(durationValue)
in
  recordValue
```

In this example, we're creating a duration value of 2 hours and 30 minutes using the #duration function. Then, we're passing that value to the Duration.ToRecord function to create a record. The resulting record will look like this:

```
[Hours=2, Minutes=30]
```

Understanding the M code behind Duration.ToRecord

Now that we've seen how to use the Duration.ToRecord function, let's take a look at the M code behind it. Here's the code for the Duration.TotalDays

Behind this function lies the M code, a programming language used by Power Query to transform and manipulate data. In this article, we will explore the M code behind the Duration.TotalDays function and how it works.

Understanding the Duration.TotalDays Function

The Duration.TotalDays function is used to convert a duration value into the number of days. It takes one argument, which is the duration value to be converted.

For example, if we have a duration value of 2 days and 6 hours, we can use the Duration.TotalDays function to convert it into 2.25 days. The M code behind this function performs the necessary calculation to convert the duration value into the equivalent number of days.

The M Code Behind the Duration.TotalDays Function

The M code behind the Duration.TotalDays function is a simple formula that takes the duration value and divides it by the total number of seconds in a day. This calculation gives us the number of days in decimal form.

The M code for the Duration.TotalDays function looks like this:

```
(duration as duration) =>
   duration / #duration(1,0,0,0)
```

Let's break down this code to understand what it does.

The first line defines the function and its input, which is a duration value. The second line performs the calculation to convert the duration value into the number of days.

The #duration(1,0,0,0) is a duration value that represents one day. It has a format of #duration(days, hours, minutes, seconds). In this case, we have set the days value to 1 and the other values to 0 to represent one full day.

The M code divides the duration value by the one-day duration value to get the number of days. This calculation is performed for each row in the data set, allowing us to convert multiple duration values into their equivalent number of days.

Using the Duration.TotalDays Function

To use the Duration.TotalDays function in Power Query, we need to have a duration value to convert. This value can be obtained from a column containing duration values or from a custom formula.

Duration.TotalHours

But what is the M code behind this function? How does it work? In this article, we'll explore the M code behind the Duration.TotalHours function and learn how to use it effectively.

Understanding Durations in Power Query

Before we dive into the M code behind the Duration.TotalHours function, let's first understand what durations are in Power Query.

A duration is a measure of time that is represented by a fixed number of seconds. For example, a duration of 1 hour is equal to 3,600 seconds. In Power Query, durations are represented as values of the Duration data type.

To create a duration in Power Query, you can use the Duration.FromText function. For example, the following M code creates a duration of 1 hour:

```
Duration.FromText("1:00:00")
```

The M Code Behind Duration.TotalHours

Now that we understand durations in Power Query, let's take a closer look at the M code behind the Duration.TotalHours function.

The Duration.TotalHours function is a simple function that takes a duration value as its input and returns the total number of hours in that duration as a decimal value.

Here's the M code for the Duration.TotalHours function:

```
(duration as duration) as number => duration / #duration(0, 0, 0, 1)  24
```

Let's break down this code into its individual components.

The Function Signature

The function signature specifies the input and output types of the function. In this case, the function takes a duration value as its input and returns a decimal value.

Duration.TotalMinutes

What is Duration.TotalMinutes?

Before we dive into the M code behind Duration.TotalMinutes, let's first understand what this function does. As mentioned earlier, this function converts a duration value to minutes. In other words, it takes a time-based value in the format of hours, minutes, and seconds and converts it to minutes. For example, if you have a duration value of 01:30:00 (1 hour, 30 minutes, and 0 seconds), the Duration.TotalMinutes function would return the value 90 (90 minutes).

The M Code Behind Duration.TotalMinutes

The M code behind the Duration.TotalMinutes function is relatively simple. The function takes a single argument, which is the duration value to be converted to minutes. This argument can be either a duration value or a column reference that contains duration values. Here is the M code for the Duration.TotalMinutes function:

```
(duration as duration) =>
Duration.TotalMinutes(duration)
```

As you can see, the function takes the duration value as an argument and passes it to the internal Power Query function Duration.TotalMinutes. This function then converts the duration value to minutes and returns the result.

Using Duration.TotalMinutes in Power Query

Now that we understand the M code behind the Duration.TotalMinutes function, let's take a look at how we can use it in Power Query. Suppose we have a table that contains a column of duration values, and we want to convert these values to minutes. To do this, we can use the Duration.TotalMinutes function in a new column. Here is the M code for this scenario:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WcjQzNzS2MTGwMlGyUipWsgZi0YrJzUwMlGyUllSBmJQYnJ
xYkK5VqTnFQAKG1RyBi1GKyc1MDJRLzEwNzAtMy1UJzEwVcoqVooA1owNjA1MDS0MjU0MNMw0zQzNDSwMjYzSjY2N0jQ3MNIzUzIyNzE
Duration.TotalSeconds
```

What is the Duration.TotalSeconds function?

The Duration.TotalSeconds function is used to convert a duration value to a numeric value in seconds. This function takes a duration value as its input and returns a numeric value in seconds. The duration value can be a time span, a date/time value, or a duration value.

The M Code behind the Duration.TotalSeconds Function

The M code behind the Duration.TotalSeconds function is relatively simple. It involves the use of the Duration.TotalSeconds function and the Duration.From function, both of which are built-in functions in Power Query.

Here's the M code for the Duration.TotalSeconds function:

```
(duration as duration) =>
Duration.TotalSeconds(duration)
```

As you can see, the function takes a single input parameter, which is the duration value that needs to be converted. This parameter is of the data type 'duration'.

The function then calls the built-in function Duration.TotalSeconds, passing the input parameter (duration) as an argument. The Duration.TotalSeconds function then returns a numeric value in seconds.

The Duration.TotalSeconds function is actually a simpler version of the Duration.ToText function, which is used to convert a duration value to a text value. The Duration.ToText function allows you to specify the format of the text value.

Here's the M code for the Duration.From function:

```
(seconds as number) =>
Duration.From(seconds)
```

The Duration.From function takes a single input parameter, which is the numeric value in seconds that needs to be converted to a

Duration.ToText

The M code behind the Duration.ToText function can be customized to suit specific business needs. In this article, we will explore the M code behind the Duration.ToText function and how it can be customized.

Understanding the Duration.ToText Function

The Duration.ToText function in Power Query is used to convert duration values, which are represented in seconds, to a text format that is more readable. By default, this function converts the duration value to a text format that displays the number of hours, minutes, and seconds.

For example, if the duration value is 126 seconds, the function will return the text value “00:02:06” which represents 2 minutes and 6 seconds.

The Duration.ToText function takes two arguments: the duration value and the time zone. The time zone argument is optional and is used to adjust the time zone for the duration value. If the time zone argument is not specified, the function will use the local time zone.

Customizing the Duration.ToText Function

The M code behind the Duration.ToText function can be customized to suit specific business needs. To customize this function, we need to modify the M code that is generated by Power Query when the function is used.

One common customization of the Duration.ToText function is to display the duration value in a different format. For example, we may want to display the duration value in days, hours, and minutes instead of hours, minutes, and seconds.

To achieve this, we need to modify the M code that is generated by Power Query when the function is used. We can do this by using the “Advanced Editor” in Power Query.

Once we have opened the “Advanced Editor”, we can modify the M code that is generated by the Duration.ToText function. In this case, we can modify the “Duration” type to “Duration.Days”. This will convert the duration value to days, hours, and minutes.

```
(Duration as duration) =>
let
  Days = Duration.Days(Duration),
  Hours = Duration.Hours(Duration),
  Minutes = Duration.Minutes(Duration),
  Result =
Embedded.Value
```

What is Power Query M?

Power Query M is a functional language used in Power Query. It is used to perform data transformations and create custom functions. M is a case-sensitive language that uses a combination of expressions and values to perform calculations and data manipulations. The Embedded.Value function is one of the built-in functions in Power Query M.

Understanding the Embedded.Value Function

The Embedded.Value function is used to extract values from cells in an Excel sheet. It takes two arguments: the first argument is a cell reference, and the second argument is an optional default value. If the cell contains a value, the function returns that value. If the cell is empty, the function returns the default value. Here is the syntax for the Embedded.Value function:

```
Embedded.Value(cellReference, defaultValue)
```

The cellReference argument can be a reference to a single cell or a range of cells. The defaultValue argument is optional, and if it is not specified, the function returns a blank value if the cell is empty.

How the Embedded.Value Function Works

Let's take a deeper look at how the Embedded.Value function works. Suppose you have an Excel sheet with the following data:

Customer Name	Sales
John	$100
Jane	
Bob	$200
Sarah	$300

You can use the Embedded.Value function to extract the sales data for Jane. Here's an example of how to use the function:

```
= Embedded.Value([Sales][[2]])
```

Error.Record

Understanding Error.Record

Before diving into the M code behind the Error.Record function, it’s important to understand what the function does. Essentially, Error.Record allows you to create custom error messages that can be displayed when errors occur during data transformation.

For example, let’s say you have a column of data that should only contain numeric values. If there are any non-numeric values in the column, you may want to display a custom error message to alert the user. This is where Error.Record comes in.

Using Error.Record, you could create a custom error message that looks something like this:

```
Error.Record(“Invalid Data Type”, “The column X should only contain numeric values”)
```

This error message would be displayed whenever there is a non-numeric value in the column. By providing a clear and concise error message, you can help users quickly identify and resolve data transformation issues.

The M Code Behind Error.Record

Now that we understand the purpose of Error.Record, let's take a look at the M code behind the function. The syntax for Error.Record is as follows:

```
Error.Record(errorCode as text, details as text)
```

The errorCode parameter is used to specify a unique error code for the error message. This can be any text string that you choose. The details parameter is used to provide more information about the error. This could include a description of the error, the name of the column or table where the error occurred, or any other relevant details.

When an error occurs during data transformation, Power Query will evaluate any Error.Record functions that are present in the query. If an Error.Record function matches the error that occurred, the custom error message will be displayed. If there are no matching Error.Record functions, the default error message will be displayed.

Essbase.Cubes

The Essbase.Cubes function allows you to connect to Essbase, a multidimensional database management system. With this function, you can access data from Essbase cubes and bring it into Power Query. But what's the M code behind the Essbase.Cubes function? Let's take a closer look.

Understanding the Essbase.Cubes Function

Before diving into the M code behind the Essbase.Cubes function, it's important to understand what the function does. The Essbase.Cubes function returns a table of cube metadata for a given Essbase server. The metadata includes information about the cubes available on the server, such as the name of the cube, the number of dimensions, and the size of the cube.

The function takes two arguments: the Essbase server name and the optional Essbase login credentials. Here's an example of the Essbase.Cubes function in action:

```
let
  Source = Essbase.Cubes("EssbaseServerName"),
  #"Filtered Rows" = Table.SelectRows(Source, each ([Cube Name] = "Sample"))
in
  #"Filtered Rows"
```

In this example, we're using the Essbase.Cubes function to connect to an Essbase server named "EssbaseServerName" and retrieve metadata for all cubes on that server. We then filter the results to only include the cube named "Sample".

Breaking Down the M Code

Now that we understand what the Essbase.Cubes function does, let's take a closer look at the M code behind it. Here's the M code for the Essbase.Cubes function:

```
let
  EssbaseCubes = (EssbaseServerName as text, optional EssbaseCredentials as record) =>
Excel.CurrentWorkbook
```

What is Excel.CurrentWorkbook?

Excel.CurrentWorkbook is a function in Power Query M that allows you to reference tables and named ranges within the current workbook. This function is particularly useful when you have a workbook with multiple tables and you want to extract data from them into a single query.

How to use Excel.CurrentWorkbook

To use Excel.CurrentWorkbook, you need to understand its syntax. The syntax for this function is as follows:

```
Excel.CurrentWorkbook()
```

This function returns a record containing all the tables and named ranges in the current workbook. You can reference this record to extract data from specific tables or named ranges.

Here's an example of how to use Excel.CurrentWorkbook to extract data from a table:

```
let
  Source = Excel.CurrentWorkbook(),
  Table1 = Source{[Name=”Table1"]}[Content]
in
  Table1
```

In this example, we first use Excel.CurrentWorkbook to get a record of all the tables and named ranges in the current workbook. We then reference the "Table1" table from this record and extract its content.

The M code behind Excel.CurrentWorkbook

The M code behind Excel.CurrentWorkbook is quite simple. Here's what it looks like:

```
Excel.ShapeTable
```

The Excel.ShapeTable function takes in a single argument, which is a list of shapes. The shapes can be either images or geometric shapes such as rectangles, circles, and lines. When the function is executed, it returns a table with columns for each shape property such as height, width, and shape type.

Understanding the M Code Behind the Excel.ShapeTable Function

The M code behind the Excel.ShapeTable function is what makes it possible for the function to work. The function is written in the M language, which is a functional programming language used by Power Query. The M code behind the Excel.ShapeTable function can be broken down into two parts: the function signature and the function body.

Function Signature

The function signature is the first part of the M code and defines the function name and its input parameters. In the case of the Excel.ShapeTable function, the function signature is as follows:

```
Excel.ShapeTable = (shapes as list) =>
```

The function name is Excel.ShapeTable, and it takes in a single parameter called shapes, which is of type list.

Function Body

The function body is the second part of the M code and contains the instructions that the function executes when called. In the case of the Excel.ShapeTable function, the function body is as follows:

```
let
  Source = Table.FromRecords(
    List.Transform(
      shapes,
      each Record.FromList(Shape.ToList(_), Shape.PropertyNames)
    ),
Exchange.Contents
```

Introduction to Exchange.Contents

Exchange.Contents is a Power Query M function that allows users to connect to web data sources and extract data from them. It is a versatile function that can be used to connect to a wide range of web data sources, including web pages, web services, and other web-based data sources. The function takes a URL parameter as input and returns a table of data that represents the data on the web page or other data source.

Understanding the M Code Behind Exchange.Contents

To fully understand the M code behind Exchange.Contents, we need to understand how web data sources work. Web data sources are typically accessed using web protocols such as HTTP or HTTPS. When a user enters a URL into a web browser, the browser sends a request to the web server that hosts the data source. The server then responds with a web page or other data that represents the data on the data source.

The Exchange.Contents function works in a similar way. When the function is called with a URL parameter, it sends a request to the web server that hosts the data source, and the server responds with data that represents the data on the data source. The data is then converted into a table format and returned as output.

Using Exchange.Contents to Extract Data from Web Data Sources

Exchange.Contents can be used to extract data from a wide range of web data sources. Some examples of web data sources that can be accessed using Exchange.Contents include:

– Web pages: Exchange.Contents can be used to extract data from web pages, including tables, lists, and other types of data.
– Web services: Exchange.Contents can be used to extract data from web services that provide data in JSON, XML, or other formats.
– FTP sites: Exchange.Contents can be used to extract data from FTP sites that provide data in text, CSV, or other formats.

To use Exchange.Contents to extract data from a web data source, you need to provide the URL of the data source as input. For example, the following M code can be used to extract data from a web page:

```
let
  Source = Exchange.Contents(“https://www.example.com”),
  #”Extracted Table” = Source{[Name=”table1"]}[Data]
in
```

Expression.Constant

Understanding the Expression.Constant Function

The Expression.Constant function is a built-in function in Power Query that allows users to create a constant value. The syntax for the function is as follows:

```
Expression.Constant(value as any) as function
```

The function takes a single argument, value, which can be any data type. The function then returns a function that, when called, returns the value passed as an argument.

For example, the following M code creates a constant value of 42:

```
constantValue = Expression.Constant(42)
```

The constantValue variable now contains a function that, when called, will return the value 42.

Using the Expression.Constant Function

The Expression.Constant function can be used in a variety of ways in Power Query. One common use case is to define a constant value that can be reused throughout a query. For example, if we wanted to calculate the average price of a product across multiple regions, we could use the following M code:

```
pricePerRegion = Table.Group(products, {“Region”}, {{“Average Price”, each List.Average([Price])}})
```

In this code, we are grouping the products table by the Region column and calculating the average price for each region. However, we

Expression.Evaluate

The Expression.Evaluate function in Power Query M language is one such function that allows users to evaluate a string expression as a formula. This function is a powerful tool that can be used to create dynamic formulas, perform calculations on the fly, and manipulate data in complex ways.

Understanding the Expression.Evaluate Function

The Expression.Evaluate function in Power Query M language takes a single argument, which is a string expression that contains a formula. The function then evaluates the string expression as a formula and returns the result.

Here's an example that shows how the Expression.Evaluate function works:

```
Expression.Evaluate("1 + 2 + 3")
```

The above formula will evaluate to 6, which is the sum of 1, 2, and 3. This is a simple example, but it illustrates the power of the Expression.Evaluate function.

Using Expression.Evaluate to Create Dynamic Formulas

The Expression.Evaluate function can be used to create dynamic formulas that change based on the data in a table. Here's an example:

```
Expression.Evaluate("if [Sales] > 1000 then [Sales]  0.1 else [Sales]  0.05")
```

In the above example, the formula uses an if statement to determine whether the value in the Sales column is greater than 1000. If it is, the formula multiplies the value by 0.1; otherwise, it multiplies the value by 0.05. This is a dynamic formula that changes based on the data in the Sales column.

Manipulating Data with Expression.Evaluate

The Expression.Evaluate function can also be used to manipulate data in complex ways. For example, you can use the function to split a column into multiple columns based on a delimiter.

Expression.Identifier

What is Expression.Identifier function?

The Expression.Identifier function is a built-in function in M language that is used to create unique identifiers for data tables. It takes an expression as input and returns a text value that is unique to that expression. It is commonly used in data transformation scenarios where there is a need to merge or join data tables based on a common key.

The syntax of the Expression.Identifier function is as follows:

```
Expression.Identifier(expression)
```

The expression parameter is the input value for which the unique identifier needs to be generated. It can be any valid M expression, such as a column name, a list of values, or a record.

How does Expression.Identifier function work?

The Expression.Identifier function uses a hashing algorithm to generate a unique text value for the input expression. The algorithm takes the input expression, converts it to a binary format, and then applies a set of mathematical operations to generate a hash value. The hash value is then converted to a text value using base 64 encoding. The resulting text value is guaranteed to be unique for any given input expression.

The unique identifiers generated by the Expression.Identifier function are used to join or merge data tables based on a common key. When two or more tables need to be merged, a common key column is identified, and the Expression.Identifier function is used to generate a unique identifier for each row in the key column. The unique identifiers are then used to match rows in the key column of the two tables and merge them into a single table.

Examples of using Expression.Identifier function

Here are some examples of using the Expression.Identifier function in Power Query:

Example 1: Generating unique identifiers for a column

Suppose you have a table with a column named "Name" that contains a list of names. You want to generate unique identifiers for each name in the column. Here is the M code that uses the Expression.Identifier function to achieve this:

```
File.Contents
```

In this article, we'll take a closer look at the M code behind `File.Contents`. We'll explore how the function works, how it's used in Power Query, and how you can modify the code to suit your needs.

Understanding the File.Contents Function

At its core, `File.Contents` is a simple function. It takes a single argument, which is the path to the file you want to read. When you call the function, it reads the contents of the file and returns a table with a single column containing the file's contents.

Here's an example:

```
let
  Source = File.Contents("C:UsersJohnDoeexample.txt")
in
  Source
```

In this example, we're using `File.Contents` to read the contents of a file named "example.txt" located on the "C:UsersJohnDoe" directory. The function reads the file's contents and returns a table with a single column containing the contents of the file.

The M Code Behind File.Contents

To understand how `File.Contents` works

Folder.Contents

The Folder.Contents function is written in a language called M, which is used exclusively in Power Query. In this article, we will explore the M code behind the Folder.Contents function and how it works.

Understanding the Folder.Contents Function

Before we dive into the M code behind the Folder.Contents function, let's first understand what this function does. The Folder.Contents function returns a table that lists all the files in a folder, along with their metadata such as file name, file path, size, and date modified. The syntax for the Folder.Contents function is as follows:

```
Folder.Contents(folder_path)
```

Here, `folder_path` is the path of the folder from which you want to import files.

The M Code Behind the Folder.Contents Function

Now let's take a closer look at the M code behind the Folder.Contents function. When you use the Folder.Contents function, Power Query generates the following M code:

```
let
  Source = Folder.Files(folder_path),
  #"Filtered Rows" = Table.SelectRows(Source, each ([Attributes]?[Hidden]? <> true) and ([Attributes]?[ReadOnly]? <> true)),
  #"Removed Other Columns" = Table.SelectColumns(#"Filtered Rows",{"Folder Path", "Name", "Extension", "Date accessed", "Date modified", "Date created", "Attributes", "Content"}),
  #"Expanded Content" = Table.ExpandBinaryColumn(#"Removed Other Columns", "Content", {"Content"}, {"Content"})
in
  #"Expanded Content"
```

Function.From

Introduction to the Function.From M Function

The Function.From M function is a powerful tool that allows users to create custom functions in Power Query. This function takes in two parameters, the first being a record type that defines the function signature, and the second being a function that performs the desired transformation.

Record types are used to define the input and output of the function. They specify the data types of the function parameters and return values. The function that is passed as a parameter to Function.From is used to define the transformation logic.

Basic Syntax of Function.From

The basic syntax of the Function.From M function is as follows:

```
Function.From([functionType as type], [functionImplementation as function], [optional options as nullable record])
```

The functionType parameter is a record type that defines the input and output of the function. The functionImplementation parameter is a function that performs the desired transformation. The optional options parameter is used to specify additional options that control the behavior of the function.

Example Usage of Function.From

To illustrate the usage of Function.From, let's consider the following example. Assume we have a table that contains sales data for different products. We want to create a custom function that calculates the total sales for each product.

We can define the record type for our custom function as follows:

```
let
  SalesRecord = [ProductID = 0, TotalSales = 0],
  SalesFunctionType = type function (SalesRecord as any) as any
in
  SalesFunctionType
```

Function.Invoke

In this article, we'll take a deep dive into the M code behind `Function.Invoke`. We'll explore how it works, what it can do, and how to use it effectively in your Power Query queries.

Understanding `Function.Invoke`

At its core, `Function.Invoke` is a function that takes two arguments: a function to invoke, and a record of arguments to pass to that function. Here's a simple example:

```
let
  fn = (a, b) => a + b,
  args = [a = 1, b = 2],
  result = Function.Invoke(fn, args)
in
  result
```

In this example, we define a simple function `fn` that takes two arguments and returns their sum. We also define a record of arguments `args` that sets the values of `a` and `b`.

Finally, we call `Function.Invoke` and pass in `fn` and `args`. The result is the sum of `a` and `b`, which is `3`.

Using `Function.Invoke` for Dynamic Function Calls

So why is `Function.Invoke` useful? The primary benefit is that it allows you to call functions dynamically at runtime, rather than statically defining them in your query.

This is particularly useful when you're dealing with data that has varying shapes or structures. For example, consider a dataset that contains different columns depending on the source. Using `Function.Invoke`, you can create a query that automatically adapts to these varying structures.

Here's a simple example. Let's say we have two tables, `TableA` and `TableB`, and we want to merge them together. However, the two tables have different columns:

Function.InvokeAfter

Function.InvokeAfter is one of the useful functions available in M that allows you to delay the execution of a function for a certain amount of time. In this article, we will take a closer look at the M code behind Function.InvokeAfter and explore its use cases.

Syntax of Function.InvokeAfter

The syntax of Function.InvokeAfter is as follows:

```
Function.InvokeAfter(function as function, delay as duration) as any
```

The function parameter specifies the function that you want to execute after a delay. The delay parameter specifies the duration of the delay before the function is executed. The result of the function is returned as any.

Examples of Function.InvokeAfter

Here are some examples of how Function.InvokeAfter can be used:

Example 1: Delay Execution of a Function

Suppose you have a function that takes a long time to execute, and you want to delay its execution until after all other functions have finished executing. You can use Function.InvokeAfter to delay the execution of the function as follows:

```
let
  delay = #duration(0, 0, 5, 0), // delay execution for 5 seconds
  result = Function.InvokeAfter(() => MyFunction(), delay)
in
  result
```

In this example, we define a delay of 5 seconds using the #duration function. We then use Function.InvokeAfter to delay the execution of MyFunction by 5 seconds. The result of MyFunction is stored in the result variable.

Function.IsDataSource

What is Function.IsDataSource?

Function.IsDataSource is a built-in function in Power Query M that checks whether the given input is a valid data source or not. It returns a boolean value of true if the input is a valid data source and false otherwise.

How does Function.IsDataSource work?

Function.IsDataSource works by validating the input against a set of rules. These rules are defined in the M code behind the function. Let's take a look at the M code behind Function.IsDataSource and understand how it works.

```
let
  isDataSource = (input as any) =>
    let
      dataSource = if Type.Is(input, type table) then input else null,
      validDataSource = if dataSource <> null and Record.HasFields(dataSource, { “Data”, “Columns” }) then dataSource else null
    in
      validDataSource <> null
in
  isDataSource
```

The M code for Function.IsDataSource contains a nested function called isDataSource that takes an input parameter of any data type. The first step in the function is to check whether the input is of type table. If the input is not of type table, the dataSource variable is set to null.

If the input is of type table, the dataSource variable is set to the input. The next step is to check whether the dataSource variable is null or not. If it is null, the validDataSource variable is also set to null.

If the dataSource variable is not null, the next step is to check whether it has the required fields - "Data" and "Columns". If the dataSource variable has these fields, it is considered a valid data source and the validDataSource variable is set to the dataSource variable. If the dataSource variable does not have these fields, the validDataSource variable is set to null.

Function.ScalarVector

In this article, we will explore the M code behind the Function.ScalarVector function and discuss how it can be used to transform data in Power Query.

Understanding Scalar Functions

Before we dive into the M code behind the Function.ScalarVector function, it's important to understand scalar functions. Scalar functions are functions that operate on a single value and return a single value. Examples of scalar functions include:

– Text functions, such as Text.Length and Text.Upper
– Math functions, such as Number.Abs and Number.Round
– Date and time functions, such as DateTime.AddDays and DateTime.FromText

Scalar functions are extremely useful in data transformation, as they allow users to apply a consistent operation to each value in a dataset.

Introducing the Function.ScalarVector Function

The Function.ScalarVector function is a powerful tool that allows users to apply a scalar function to each item in a vector. The syntax for the Function.ScalarVector function is as follows:

```
Function.ScalarVector(vector as list, function as function) as list
```

The function takes two arguments: a vector (represented as a list) and a scalar function. The function is then applied to each item in the vector, resulting in a new vector with the same number of items.

For example, if we have a vector containing the values {1, 2, 3} and we want to apply the Number.Square function to each item, we would use the following M code:

```
Function.ScalarVector({1, 2, 3}, Number.Square)
```

```
Geography.FromWellKnownText
```

In this article, we will dive into the M code behind the Geography.FromWellKnownText function and understand how it works.

Understanding Well-Known Text (WKT)

Before we dive into the M code behind the Geography.FromWellKnownText function, let’s first understand what Well-Known Text (WKT) is.

WKT is a text representation of a geometric object that is commonly used in GIS (Geographic Information Systems) applications. It describes the shape, size, and location of the object using a set of coordinates.

Here’s an example of a WKT representation of a point:

```
POINT (-122.34900 47.65100)
```

This represents a point with longitude -122.34900 and latitude 47.65100.

The M Code Behind Geography.FromWellKnownText

Now that we understand what WKT is, let's take a look at the M code behind the Geography.FromWellKnownText function.

The function takes a single parameter, which is a text value that represents a WKT representation of a geometric object. Here's the syntax of the function:

```
Geography.FromWellKnownText(wktText as text) as nullable geography
```

The function returns a nullable geography value, which represents the geography object that corresponds to the WKT representation.

Here's an example usage of the function:

```
let
Geography.ToWellKnownText
```

The Geography.ToWellKnownText M function is a powerful tool in Power Query that allows users to transform geographical data into a Well-Known Text (WKT) format. WKT is a text markup language that represents spatial objects in a human-readable format.
In this article, we will explore the M code behind the Geography.ToWellKnownText function and how it can be used to transform geographical data.
What is the Geography.ToWellKnownText function?
The Geography.ToWellKnownText function is a built-in function in Power Query that converts a geographical object to a Well-Known Text (WKT) string. The function takes a single argument, which is a geography value.
The geography value can be a point, line, or polygon. The function then converts the geography value to a WKT string, which can be used in various geographic information systems (GIS) applications.
How does the Geography.ToWellKnownText function work?
The Geography.ToWellKnownText function works by converting a geography value to a WKT string. The function uses the Open Geospatial Consortium (OGC) standard for WKT representation.
The WKT format represents a geography value as a string of text. The string is made up of different elements that represent the geometry of the spatial object. The elements include the type of geometry, the coordinates of the spatial object, and any other necessary information.
The Geography.ToWellKnownText function works by parsing the geography value and generating a string of text that conforms to the WKT standard.
Examples of using the Geography.ToWellKnownText function
Let’s take a look at some examples of using the Geography.ToWellKnownText function in Power Query.
Example 1: Converting a point to WKT
Suppose we have a table with a column named “Location,” which contains the latitude and longitude of different points.
To convert the points to WKT, we can use the following M code:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText(“i45WMjQwNLE0NjJQ0lFKSVEyNVHiDc5MqeozCjJSMjAyNjK3Ss
GeographyPoint.From
```

What is the GeographyPoint.From function?

The GeographyPoint.From function is used to create a point in the geography data type, which represents a single location on the Earth's surface. The function takes two arguments: latitude and longitude, both of which are decimal values. The function returns a geography data type that contains the specified latitude and longitude coordinates.

How does the GeographyPoint.From function work?

The M code behind the GeographyPoint.From function is relatively simple. Here is an example of the M code:

```
let
  lat = 47.6062,
  lon = -122.3321,
  point = GeographyPoint.From(lat, lon)
in
  point
```

In this example, we define the latitude and longitude coordinates as decimal values. We then use the GeographyPoint.From function to create a point in the geography data type. Finally, we return the point as the output of the M code.

The GeographyPoint.From function works by creating a new instance of the geography data type and setting the Latitude and Longitude properties to the specified values. The resulting geography data type can be used in other Power Query transformations, such as spatial joins or distance calculations.

What are some use cases for the GeographyPoint.From function?

The GeographyPoint.From function is useful in a variety of scenarios where latitude and longitude coordinates need to be converted into a geography data type. Here are a few examples:

- Mapping data: If you have a dataset that contains latitude and longitude coordinates, you can use the GeographyPoint.From function to create a geography data type that can be plotted on a map.
- Distance calculations: If you need to calculate the distance between two locations, you can use the GeographyPoint.From function to

Geometry.FromWellKnownText

The M code behind the Geometry.FromWellKnownText function is responsible for parsing the WKT string and creating a spatial value that can be used in Power Query. In order to fully understand the M code behind this function, we need to first understand what WKT is and how it represents geometric objects.

What is Well-Known Text?

Well-known text (WKT) is a text markup language that is used to represent geometric objects. WKT is a part of the Open Geospatial Consortium (OGC) Simple Features Specification, which defines a standard for representing spatial data. WKT is used to represent points, lines, polygons, and other geometric objects in a human-readable format.

Here is an example of a WKT string that represents a polygon:

```
POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))
```

In this example, the polygon is defined by a series of points that form a closed shape. Each point is represented by its x and y coordinates, separated by a space. The entire string is enclosed in parentheses and preceded by the keyword "POLYGON".

How Does Geometry.FromWellKnownText Work?

The Geometry.FromWellKnownText function takes a WKT string as its input and returns a spatial value that represents the geometric object. The M code behind this function is responsible for parsing the WKT string and creating a spatial value that can be used in Power Query.

Here is an example of how to use the Geometry.FromWellKnownText function in Power Query:

```
let
  wktString = “POINT (30 10)”,
  spatialValue = Geometry.FromWellKnownText(wktString)
in
  spatialValue
```

Geometry.ToWellKnownText

What is WKT?

Well-known text (WKT) is a text markup language used to represent vector geometry objects such as points, lines, and polygons. WKT is an open format that is widely used in geographic information systems (GIS) and is supported by many software packages. The WKT format is human-readable and easy to parse, making it an ideal choice for data exchange and interoperability.

Understanding the Geometry.ToWellKnownText Function

The Geometry.ToWellKnownText function is used to convert a geometry object to a well-known text (WKT) representation. The function takes a geometry object as input and returns a text string in WKT format. The syntax for the function is as follows:

```
Geometry.ToWellKnownText(geometry as any) as text
```

The `geometry` parameter is the input geometry object, which can be of any type that implements the `IGeometry` interface. The function returns a text string in WKT format.

The M Code Behind Geometry.ToWellKnownText

The M code behind the Geometry.ToWellKnownText function is relatively simple and easy to understand. The function is defined as follows:

```
let
  toWKT = (geometry as any) as text =>
    let
      wktType =
        if Type.Is(geometry, type function (type nullable any) as nullable any) then
          Type.RecordFields(Type.AsNullable(geometry)){0}[Type],
        else if Type.Is(geometry, type nullable any) then
          Type.RecordFields(Type.AsNullable(geometry)){0}[Type],
GeometryPoint.From
```

What is GeometryPoint.From?

GeometryPoint.From is a Power Query M function that is used to create a point in a two-dimensional coordinate system. The function takes two arguments, x and y, which represent the x and y coordinates of the point. The x and y arguments must be numeric values. The function returns a record that represents the point in the coordinate system.

Understanding the M code behind GeometryPoint.From

To understand the M code behind GeometryPoint.From, we need to understand the structure of the record that is returned by the function. The record has two fields, x and y, which represent the x and y coordinates of the point. The M code behind GeometryPoint.From is as follows:

```
(x as number, y as number) =>
[
  x = x,
  y = y,
  type = type nullable number meta [
    IsCoordinate = true,
    IsNullable = true
  ]
]
```

The first line of the M code defines the two arguments, x and y, that are passed to the function. The second line creates a record that has three fields, x, y, and type. The x and y fields are assigned the values of the x and y arguments, respectively. The type field is a metadata field that is used to identify the record as a coordinate point. The type field is nullable, which means that it can be null.

Using GeometryPoint.From

Now that we understand the M code behind GeometryPoint.From, let's look at some examples of how it can be used to work with spatial data in Power Query.

GoogleAnalytics.Accounts

The M code behind the GoogleAnalytics.Accounts M function is responsible for establishing a connection to the Google Analytics API, authenticating the user, and retrieving the data requested by the user. In this article, we will dive deep into the M code that powers this function and understand how it works.

Setting up the Connection

The first step in retrieving data from the Google Analytics account is to establish a connection to the Google Analytics API. This is done by using the following M code:

```
let
  Source = GoogleAnalytics.Accounts(),
  #”Changed Type” = Table.TransformColumnTypes(Source,{{“Id”, type text}})
in
  #”Changed Type”
```

The GoogleAnalytics.Accounts() function returns a table containing the list of Google Analytics accounts available to the user. The function does not require any parameters and can be called with an empty argument list. Once the list is retrieved, it is transformed to change the data type of the "Id" column to text.

Authenticating the User

Once the list of accounts is retrieved, the user needs to authenticate themselves with the Google Analytics API. This is done by using the following M code:

```
let
  Source = GoogleAnalytics.Accounts(
    [
      #”Client ID”=”xxxxxx”,
```

Graph.Nodes

One of the most useful functions in Power Query is Graph.Nodes. This function allows you to extract information about the nodes in a graph, which can be useful for analyzing network data. In this article, we'll take a look at the M code behind the Graph.Nodes function, and show you how to use it to extract information from a graph.

What is a Graph?

Before we dive into the M code behind Graph.Nodes, it's important to understand what a graph is. In computer science, a graph is a collection of nodes (also called vertices) and edges. Nodes can represent anything from people to computer systems to web pages, and edges represent the connections between them.

Graphs are often used to represent complex networks, such as social networks or computer networks. By analyzing the nodes and edges in a graph, you can gain insight into the underlying structure of the network.

The M Code Behind Graph.Nodes

The Graph.Nodes function takes a graph as its input and returns a table that contains information about the nodes in the graph. The table has two columns: Node and Properties. The Node column contains the name of each node in the graph, and the Properties column contains a record that describes the properties of each node.

Here is an example of the M code that you can use to extract information about the nodes in a graph:

```
let
  Source = Graph.FromEdges({{1, 2}, {1, 3}, {2, 3}, {3, 4}, {4, 5}}),
  Nodes = Graph.Nodes(Source)
in
  Nodes
```

In this example, we are creating a graph with five nodes and five edges. The Graph.FromEdges function takes a list of edges and returns a graph object. We then pass this graph object to the Graph.Nodes function to extract information about the nodes.

When you run this code, you will get a table that looks like this:

Guid.From

What is a GUID?

A GUID is a unique identifier that is used to represent a piece of data in a globally unique way. GUIDs are often used in computer systems to ensure that data is unique, even when it is copied or transferred between different systems. GUIDs are usually represented as a 16-byte sequence of hexadecimal digits, such as “6F9619FF-8B86-D011-B42D-00C04FC964FF”.

The Guid.From Function

The Guid.From function is a built-in function in Power Query that is used to convert a 16-byte GUID into a text string. The syntax of the Guid.From function is as follows:

```
Guid.From(binaryGUID as binary) as text
```

The function takes a single argument, which is a binary value representing the 16-byte GUID. The function returns a text value representing the GUID in the standard format, with groups of hexadecimal digits separated by hyphens.

The M Code Behind Guid.From

The M code behind the Guid.From function is relatively simple, but it is still worth examining in detail. The following code snippet shows the implementation of the Guid.From function:

```
let
  Bytes = List.Buffer(Binary.ToList(binaryGUID)),
  Guid = Text.Combine(
    {
      Text.From(Bytes{0}, “X2”), Text.From(Bytes{1}, “X2”), Text.From(Bytes{2}, “X2”), Text.From(Bytes{3}, “X2”),
      “-“,
      Text.From(Bytes{4}, “X2”), Text.From(Bytes{5}, “X2”),
      “-“,
Hdfs.Contents
```

What is Hadoop Distributed File System (HDFS)?

Hadoop Distributed File System (HDFS) is a distributed file system that is part of the Apache Hadoop ecosystem. It is designed to store and manage large amounts of data across multiple commodity hardware nodes. HDFS is used in big data processing applications where data is stored in a distributed manner across multiple nodes for parallel processing.

What is Power Query M?

Power Query M is the functional programming language used by Power Query. It is used to transform and shape data from various data sources before loading it into a destination. Power Query M is a case-sensitive language that is similar to F# and is easy to learn and use.

What is Hdfs.Contents?

Hdfs.Contents is a Power Query M function used to connect to Hadoop Distributed File System (HDFS) and retrieve a list of files and folders in a specified directory. The function takes a single argument, the Hadoop Distributed File System (HDFS) directory path, and returns a table with the following columns:

– Name: The name of the file or folder.
– Folder Path: The fully qualified HDFS path to the folder containing the file.
– Extension: The file extension (if applicable).
– IsFolder: A Boolean value indicating whether the item is a folder or a file.
– Date Accessed: The date the item was last accessed.
– Date Modified: The date the item was last modified.
– Date Created: The date the item was created.

The M Code Behind Hdfs.Contents

The M code behind Hdfs.Contents is composed of several functions that work together to establish a connection to Hadoop Distributed File System (HDFS) and retrieve a list of files and folders in a specified directory. Here is a breakdown of the M code for Hdfs.Contents:

```
let
  Source = (directoryPath as text) =>
    let
      HadoopUri = “hdfs://:“,
```

Hdfs.Files

What is Power Query?

Power Query is a data connectivity and transformation tool that allows users to connect to various data sources, transform the data, and load it into Excel or Power BI. It provides a user-friendly interface making it easy to perform complex data transformations without the need for coding. Power Query supports a wide range of data sources, including databases, Excel files, text files, and web services.

What is Hdfs.Files?

Hdfs.Files is a Power Query M function that allows users to connect to Hadoop Distributed File System (HDFS) and retrieve a list of files from a directory in HDFS. HDFS is a distributed file system that is commonly used in big data environments. The Hdfs.Files function makes it easy to connect to HDFS and retrieve files for further processing.

The M Code Behind Hdfs.Files

The M code behind Hdfs.Files is relatively simple. The function takes two parameters: the URL of the HDFS instance and the directory path to retrieve the files from. The function uses the Web.Contents function to connect to the HDFS instance and retrieve the file list. The file list is then parsed using the Xml.Tables function to extract the file names and other information.

Here is an example of the M code for the Hdfs.Files function:

```
let
  Source = (url as text, path as text) =>
    let
      hdfsUrl = url & “/webhdfs/v1” & path & “?op=LISTSTATUS”,
      fileContent = Web.Contents(hdfsUrl),
      xml = Xml.Tables(fileContent),
      files = xml{0}[#”Table1"],
      fileList = Table.FromList(files, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
      fileTable = Table.TransformColumnTypes(fileList, {{“Column1”, type text}})
    in
      fileTable
in
HdInsight.Containers
```

What is Power Query M?

Power Query M is a powerful tool that allows you to transform and clean data before importing it into Excel or Power BI. Power Query M is based on a functional programming language called “M”. M is used to define queries in Power Query and is a powerful language that allows you to manipulate and transform data in many ways.

What is HdInsight.Containers Function?

HdInsight.Containers is a Power Query M function that allows you to access data stored in Azure Blob Storage, Azure Data Lake Storage, and Hadoop Distributed File System (HDFS) through the HDInsight cluster. The function provides a simple way to import external data into Power Query and perform ETL processes on it. The function requires authentication to access the data source, which can be done through Azure Active Directory or key-based authentication.

The M Code Behind the HdInsight.Containers Function

When you use the HdInsight.Containers function, Power Query generates M code behind the scenes. The M code generated by Power Query defines the steps required to access the external data source and transform it as required. Here is an example of the M code generated by Power Query when using the HdInsight.Containers function:

```
let
  Source = HdInsight.Containers(“storageaccount.blob.core.windows.net”, “containername”, [HierarchicalNavigation=true]),
  #”blobname.csv” = Source{[Name=”blobname.csv”]}[Content],
  #”Imported CSV” = Csv.Document(#”blobname.csv”,[Delimiter=”,”, Columns=7, Encoding=1252, QuoteStyle=QuoteStyle.None]),
  #”Promoted Headers” = Table.PromoteHeaders(#”Imported CSV”, [PromoteAllScalars=true]),
  #”Changed Type” = Table.TransformColumnTypes(#”Promoted Headers”,{{“Column1”, type text}, {“Column2”, type text},
{“Column3”, type text}, {“Column4”, type text}, {“Column5”, type text}, {“Column6”, type text}, {“Column7”, type text}})
in
  #”Changed Type”
```

The M code above shows the steps required to access a CSV file stored in an Azure Blob Storage container. The code retrieves the HdInsight.Contents

What is the HdInsight.Contents Function?

The HdInsight.Contents function is a Power Query function that allows users to extract data from Azure HDInsight clusters. It takes two parameters – the URL of the HDInsight cluster and the query to execute. The function then returns a table of data that can be further transformed or loaded into other tools.

The M Code Behind the HdInsight.Contents Function

To understand the M code behind the HdInsight.Contents function, we need to understand how it works. When the function is called, it sends an HTTP request to the HDInsight cluster with the specified URL and query. The response from the cluster is in JSON format, which is then parsed by Power Query to create a table of data.

The M code for the HdInsight.Contents function is as follows:

```
let
  Source = (url as text, query as text) =>
    let
      options = [
        RelativePath = “templeton/v1/hive”,
        Headers = [#”Content-Type”=”application/json”]
      ],
      body = “{“”execute””:””” & query & “”””}”,
      result = Json.Document(Web.Contents(url, options, [Content=Text.ToBinary(body)]))
    in
      result
in
  Source
```

This code defines a function called "Source" that takes two parameters - the URL and query. It then sets some options for the HTTP

HdInsight.Files

In this article, we'll take a closer look at the M code behind the HdInsight.Files function and explore how it works.

Understanding Hadoop Distributed File System (HDFS)

Before we dive into the M code behind the HdInsight.Files function, let's first discuss what Hadoop Distributed File System (HDFS) is and how it works.

HDFS is a distributed file system that is designed to store and manage large amounts of data across a network of machines. It is a core component of the Apache Hadoop framework, which is widely used for big data processing and analytics.

In HDFS, data is stored in blocks across a network of machines. Each block is replicated across multiple machines for fault tolerance, ensuring that data can be easily recovered in case of machine failure. HDFS also includes a NameNode, which tracks the location of each block of data and manages access to the file system.

How HdInsight.Files Works

Now that we have a basic understanding of HDFS, let's explore how the HdInsight.Files function works.

The HdInsight.Files function is an M function in Power Query that allows users to connect to and retrieve data from HDFS clusters. It works by sending requests to the HDFS NameNode to retrieve information about files and directories in the file system.

When you use the HdInsight.Files function in Power Query, you'll need to provide the following parameters:

– `url`: The URL of the HDFS NameNode.

– `path`: The path to the file or directory you want to retrieve.

– `recursive`: A boolean value that determines whether to retrieve files and directories recursively.

Once you've provided these parameters, the HdInsight.Files function will send a request to the HDFS NameNode to retrieve information about the file or directory you specified. It will then return a table in Power Query that contains metadata about the file or directory, such as its name, size, and modification time.

The M Code Behind HdInsight.Files

Now that we understand how the HdInsight.Files function works, let's take a closer look at the M code behind it.

Here is an example of the M code for the HdInsight.Files function:

```
let
  Source = Hadoop.Files("hdfs://namenode:8020/", [HierarchicalNavigation=true]),
Html.Table
```

To better understand how the Html.Table function works, it is important to dive into the M code that powers it. In this article, we will explore the M code behind the Html.Table function and provide examples of how it can be used to extract data from HTML tables.

What is the Html.Table function?

The Html.Table function is a built-in function in Power Query that allows users to extract tables from HTML pages and convert them into usable data structures. This function takes a single input parameter, which is the HTML content of the webpage, and outputs a table containing the data from the HTML table.

How does the Html.Table function work?

The Html.Table function works by parsing the HTML content of a webpage and identifying the table elements within the HTML. It then extracts the data from these tables and converts them into a structured table format that can be used in data analysis.

The M code behind the Html.Table function is responsible for performing this parsing and extraction process. The code uses a combination of built-in functions and custom functions to extract the data from the HTML tables and transform it into a usable format.

Exploring the M code behind the Html.Table function

The M code behind the Html.Table function can be broken down into several distinct steps, including:

Step 1: Retrieving the HTML content of the webpage

The first step in the M code behind the Html.Table function is to retrieve the HTML content of the webpage. This is typically done using the Web.Page function, which retrieves the HTML content of a webpage and returns it as text.

Step 2: Parsing the HTML content

Once the HTML content has been retrieved, the M code behind the Html.Table function must parse the HTML and identify the table elements within it. This is typically done using the Xml.Tables function, which converts the HTML content into an XML format and identifies all of the table elements within it.

Step 3: Extracting table data

Once the table elements have been identified, the M code behind the Html.Table function must extract the data from each table and convert it into a usable format. This is typically done using a combination of built-in functions and custom functions that are designed to extract specific types of data from HTML tables.

Step 4: Combining table data

Once the data has been extracted from each table, the M code behind the Html.Table function must combine it into a single table structure. This is typically done using the Table.Combine function, which combines multiple tables into a single table structure.

Identity.From

What is Identity.From?

The Identity.From function is a simple yet powerful function in the M language. It allows you to create a new column in your query that is an exact copy of an existing column. This is useful when you want to apply additional transformations to the data in the copied column, or when you want to rename the column.

The syntax for Identity.From is as follows:

```
= Table.AddColumn(Source, NewColumnName, each [ExistingColumnName])
```

Here, Source is the name of the table or query you are working with, NewColumnName is the name you want to give to the new column, and ExistingColumnName is the name of the column you want to copy.

How to Use Identity.From

Using Identity.From is very straightforward. Simply follow these steps:

1. Open Power Query and create a new query.
2. Load the data you want to work with.
3. Select the column you want to copy.
4. Click the "Add Column" tab on the ribbon.
5. Click "Custom Column".
6. In the "Custom Column" dialog box, enter the following formula:

```
= Table.AddColumn(#”Previous Step”, “New Column Name”, each [Existing Column Name])
```

7. Replace "Previous Step" with the name of the previous step in your query, "New Column Name" with the name you want to give to the new column, and "Existing Column Name" with the name of the column you want to copy.

Identity.IsMemberOf

Understanding the Function Syntax

Before we dive into the M code, let’s examine the function syntax. The Identity.IsMemberOf function takes two parameters – the first is the user to be checked, and the second is the group to check against.

```
Identity.IsMemberOf(user as text, group as text) as logical
```

The function returns a logical value - either true or false - depending on whether the user is a member of the specified group.

The M Code Behind the Function

The M code behind the Identity.IsMemberOf function is relatively straightforward. The function first retrieves the current user's security groups using the following code:

```
groups =
Binary.Decompress(Web.Contents(“https://graph.microsoft.com/v1.0/me/transitiveMemberOf/$/microsoft.graph.group?$count=true”
), Compression.Deflate)
```

This code retrieves the current user's transitive security groups from the Microsoft Graph API and stores them in a variable called "groups."

Next, the function checks whether the specified group is included in the list of security groups. This is done using the following code:

```
isMember = List.Contains(groups, group)
```

```
IdentityProvider.Default
```

In this article, we will dive deep into the M code behind the IdentityProvider.Default function and explore its various components and how they work together.

Understanding IdentityProvider.Default

The IdentityProvider.Default function is a built-in function in Power Query that is used to provide authentication services to users who need to access data sources through Power Query queries. This function is called by default when a user creates a new query or when an existing query is refreshed.

The IdentityProvider.Default function takes several input parameters, including the credentials being used to authenticate the user and the type of authentication being used (e.g. OAuth, Windows, etc.). The function then returns a token that can be used to authenticate the user and access the data source.

The M Code Behind IdentityProvider.Default

The M code behind the IdentityProvider.Default function is a complex series of expressions and variables that work together to authenticate users and provide access to data sources. Here is a breakdown of the main components of the M code:

Step 1: Determine the Authentication Type

The first step in the M code is to determine the type of authentication being used. This is done by evaluating the authentication method specified in the query options. If no authentication method is specified, then the function defaults to Windows authentication.

Step 2: Retrieve the Credentials

The next step is to retrieve the credentials being used to authenticate the user. This is done by calling the CredentialRetrieval function and passing in the authentication type and any additional parameters required for the specific authentication method.

Step 3: Generate the Authentication Token

Once the credentials have been retrieved, the function generates an authentication token by calling the GenerateToken function and passing in the credentials and any additional parameters required for the specific authentication method. The authentication token is then returned to the user.

Step 4: Store the Authentication Token

The final step in the M code is to store the authentication token in the query settings. This is done by calling the SetQueryCredentials function and passing in the token and any additional parameters required for the specific authentication method.

The IdentityProvider.Default function is a critical component of Power Query that enables users to authenticate and access data sources. Understanding the M code behind this function is essential for developers and users who want to customize their

Implementation

What is the M Language?

The M language is a functional language used to create custom data transformations in Power Query. It is similar to other programming languages such as SQL, Python, and R, but has its own syntax and structure. M is designed to be easy to learn and use, even for those without a programming background.

M is a functional language, which means that it is based on the concept of functions. Functions are used to transform data in Power Query, and can be combined and nested to create complex data transformations. M functions are similar to Excel functions, but are designed to work with large datasets and handle complex data transformations.

The M Code Behind Power Query M Function Implementation

The M code behind the Power Query M function implementation is what enables users to create powerful data transformations in Power Query. The code is written in the M language and can be viewed and edited in the Advanced Editor window in Power Query.

The M code behind a Power Query M function implementation typically consists of three main parts:

1. Input parameters: These are the variables that are passed into the function as arguments. Input parameters are defined at the beginning of the function and are used to specify the data that the function will work with.
2. Code block: This is the main body of the function, where the data transformation logic is defined. The code block contains a series of M expressions that transform the input data into the desired output.
3. Output statement: This is the final line of the function, where the output of the function is specified. The output statement defines the format and structure of the output data, which can be a table, text, or other data type.

Understanding M Expressions

M expressions are the building blocks of the Power Query M function implementation. They are used to transform data and can be combined and nested to create complex data transformations. M expressions consist of a series of steps that are executed in a specific order.

Each step in an M expression performs a specific data transformation, such as filtering data, renaming columns, or merging tables. The steps are executed in the order that they are written, and each step takes the output of the previous step as its input.

M expressions are written using a specific syntax, which consists of keywords and operators. Some of the most commonly used keywords in M expressions include let, in, if, and else. Operators are used to perform mathematical and logical operations on data, such as addition, subtraction, and comparison.

Creating Custom M Functions

Informix.Database

The Informix.Database connector enables you to connect to an Informix database server and retrieve data from it. The connector uses the Power Query M function, which is a functional programming language that is used to define data transformations and manipulations. In this article, we will explore the M code behind the Informix.Database connector and how it works.

Connecting to an Informix Database

To connect to an Informix database using Power Query, you need to have the following information:

– Database server name or IP address
– Database name
– Port number
– Authentication method (Windows or database authentication)
– Username and password (if using database authentication)

Once you have this information, you can use the following M code to connect to the database:

```
let
  serverName = “serverName”,
  dbName = “dbName”,
  port = 1234,
  authMethod = DatabaseCredential,
  username = “username”,
  password = “password”,
  connectionString = “DRIVER={IBM INFORMIX ODBC
DRIVER};SERVER=”+serverName+”;DATABASE=”+dbName+”;HOST=”+serverName+”;SERVICE=”+port+”;PROTOCOL=onsoctcp;CLIEN
T_LOCALE=en_US.utf8;DB_LOCALE=en_US.utf8;”,
  source = Odbc.DataSource(connectionString, [HierarchicalNavigation=true, Credential=authMethod, UID=username,
PWD=password])
in
  source
```

Int16.From

Understanding Int16.From

Before we dive into the M code, let’s first understand what Int16.From does. This function takes a value as input and converts it to a 16-bit integer. The resulting integer can range from -32,768 to 32,767.

Here’s an example of how Int16.From is used:

```
Int16.From(1234)
```

In this example, the value 1234 is converted to a 16-bit integer.

The M Code Behind Int16.From

Now that we have a basic understanding of what Int16.From does, let's explore the M code behind it. The M code for Int16.From is quite simple:

```
(Int16.From as any) (value as any) =>
let
result = Int16.From(value)
in
result
```

Let's break down what each line of this code does:

- `(Int16.From as any)`: This line defines the function name and input parameter type. In this case, the function name is Int16.From and the input parameter type is any.
- `(value as any)`: This line defines the input parameter name as value and the type as any.
- `result = Int16.From(value)`: This line performs the actual conversion of the input value to a 16-bit integer and stores the result in a

Int32.From

One of the functions available in the M language is the Int32.From function. This function is used to convert a text string to a 32-bit integer value. In this article, we will explore the M code behind the Int32.From function and how it works.

Understanding the Int32.From Function

The Int32.From function is a conversion function that takes a text string as input and returns a 32-bit integer value. The syntax for the function is as follows:

```
Int32.From(text as text) as number
```

The 'text' argument is the text string that is to be converted, and the 'number' is the 32-bit integer value that is returned. The function will return an error if the text string is not a valid integer or if it is outside the range of a 32-bit integer.

The M Code Behind the Int32.From Function

The M code behind the Int32.From function is quite simple. The function is defined as follows:

```
let
  Int32.From = (text as text) as number =>
    if Text.StartsWith(text, "-") then
      -1  List.Sum(List.Transform(
        List.Skip(Text.ToList(text), 1),
        each Number.From(Text.Start(_, 1))  10 ^ (List.Count(Text.ToList(text)) – _)
      ))
    else
      List.Sum(List.Transform(
        Text.ToList(text),
        each Number.From(Text.Start(_, 1))  10 ^ (List.Count(Text.ToList(text)) – _)
```

Int64.From

Understanding the Int64.From Function

Before we dive into the M code behind the Int64.From function, let’s take a quick overview of what this function does. The Int64.From function converts a text string to a 64-bit integer value. This can be useful when working with large numbers that cannot be represented by the standard 32-bit integer data type.

Here’s an example of using the Int64.From function:

```
let
  textValue = “9223372036854775807”,
  intValue = Int64.From(textValue)
in
  intValue
```

In this example, we have a text string representing the maximum value of a 64-bit integer. We use the Int64.From function to convert this text string to a 64-bit integer value.

The M Code Behind the Int64.From Function

The M code behind the Int64.From function is relatively simple. Here's the code:

```
(Int64.From as (textValue) =>
  Number.From(Text.Replace(textValue, “,”, “”)))
```

Let's break down this code into its individual components.

The Int64.From Function Definition

The first part of the code defines the Int64.From function and its argument:

Int8.From

Understanding Int8.From

The Int8.From function in Power Query is used to convert text values to 8-bit integers. The function takes a single argument, which is the text value to be converted. Here's the basic syntax of the function:

```
Int8.From(text as text) as nullable number
```

The function returns a nullable number, which means that it can return a number or a null value. If the text value cannot be converted to an 8-bit integer, the function will return a null value.

The M Code Behind Int8.From

To understand how the Int8.From function works, we need to take a look at the M code behind it. The M code is the code that Power Query uses to perform data transformations.

Here's the M code behind the Int8.From function:

```
(text) => Number.From(Text.Remove(text, {" "}), "en-US")
```

Let's break down this code to understand what it does.

Breaking Down the M Code

The M code for the Int8.From function is a lambda function. A lambda function is a small anonymous function that can be used to create more complex functions. The lambda function takes a single argument, which is the text value to be converted.

The first part of the M code is:

```
Text.Remove(text, {" "})
```

ItemExpression.From

What is the ItemExpression.From Function?

The ItemExpression.From function is a Power Query M function that is used to extract values from a list or record. It takes two arguments: the first argument is the list or record to extract values from, while the second argument specifies the item to extract.

Understanding the M Code

To better understand the M code behind the ItemExpression.From function, let us consider an example. Suppose we have a table with a column named ‘Fruits’, and each row in this column contains a record with two fields: ‘Name’ and ‘Quantity’. We can use the ItemExpression.From function to extract the quantity of each fruit as follows:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText(“i45WCgZRCVJzsUQyKpWitWJVjNyclQykyzxKkKJFILyosB”,
BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Fruits = _t]),
  #”Changed Type” = Table.TransformColumnTypes(Source,{{“Fruits”, type any}}),
  #”Expanded Fruits” = Table.ExpandRecordColumn(#”Changed Type”, “Fruits”, {“Name”, “Quantity”}, {“Fruits.Name”,
“Fruits.Quantity”}),
  #”Extracted Quantity” = Table.AddColumn(#”Expanded Fruits”, “Quantity”, each ItemExpression.From([Fruits.Quantity],”Quantity”))
in
  #”Extracted Quantity”
```

In the above M code, we first create a source table from the data, and then transform the 'Fruits' column to contain records with two fields. We then expand the 'Fruits' column to create separate columns for the 'Name' and 'Quantity' fields. Finally, we use the ItemExpression.From function to extract the quantity of each fruit and create a new column named 'Quantity'.

The second argument in the ItemExpression.From function specifies the item to extract from the list or record. In the above example, we specify "Quantity" as the item to extract.

Best Practices for Using the ItemExpression.From Function

ItemExpression.Item

Understanding the ItemExpression.Item Function

The ItemExpression.Item function in Power Query is used to extract a value from a list or a record based on the index or the name of the item. This function takes two parameters: the first parameter is the list or record, and the second parameter is the index or the name of the item.

Here is the syntax of the ItemExpression.Item function:

```
ItemExpression.Item(listOrRecord as list or record, indexOrName as any) as any
```

The function returns the value at the specified index or name. If the index or name is not found, the function returns a null value.

Using the ItemExpression.Item Function

To understand the M code behind the ItemExpression.Item function, let's consider an example. Suppose we have a list of products with their corresponding prices, and we want to extract the price of a specific product based on its name. Here is the list:

Product	Price
Apple	1.5
Banana	2
Orange	3
Mango	4

To extract the price of a specific product, we can use the ItemExpression.Item function as follows:

```
let
  products = #table({“Product”, “Price”}, {{“Apple”, 1.5}, {“Banana”, 2}, {“Orange”, 3}, {“Mango”, 4}}),
  productName = “Apple”,
  price = products{[Product=productName]}[Price]
```

Json.Document

What is the Json.Document Function?

The Json.Document function is used to convert a JSON string into a table of values. This function takes a single parameter, which is the JSON string to be parsed. The output of this function is a table with columns and rows that correspond to the JSON data.

The M Code Behind the Json.Document Function

The M code behind the Json.Document function is relatively simple. The code uses the built-in M functions Text.FromBinary and Json.Document to convert the JSON string into a table of values.

```
Json.Document = (json as text) =>
  let
    binaryJson = Text.ToBinary(json),
    result = Json.Document(binaryJson)
  in
    result
```

The first step in the M code is to convert the JSON string into a binary format using the Text.ToBinary function. This step is necessary because the Json.Document function requires binary input.

The second step in the M code is to call the Json.Document function with the binary JSON input. The output of this function is a table with columns and rows that correspond to the JSON data.

Working with the Json.Document Function

Once you have converted your JSON data into a table of values using the Json.Document function, you can use a variety of Power Query functions to manipulate and transform the data. For example, you can use the ExpandRecordColumn function to expand columns that contain nested JSON records.

```
let
Json.FromValue
```

Understanding JSON

Before we dive into the M code behind Json.FromValue, let's first understand what JSON is and why it's important. JSON, or JavaScript Object Notation, is a lightweight text-based format for representing data. It's easy for humans to read and write, and it's also easy for machines to parse and generate. JSON is commonly used for web-based data exchange and storage, and it's also used for configuration files and other data that needs to be easily readable and writable.

JSON is essentially a collection of key-value pairs, where the keys are strings and the values can be any valid JSON data type, including strings, numbers, booleans, nulls, arrays, and objects. Here's an example of a simple JSON document:

```
{
  "name": "John Doe",
  "age": 30,
  "isMarried": false,
  "hobbies": ["reading", "cooking", "traveling"],
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "state": "NY",
    "zip": "10001"
  }
}
```

As you can see, the document contains five key-value pairs, where the values include a string, a number, a boolean, an array, and an object.

The Json.FromValue Function

The Json.FromValue function in Power Query M converts a value into a JSON document. The value can be any valid Power Query M

Lines.FromBinary

What is Lines.FromBinary?

Lines.FromBinary is a Power Query M function that converts binary data into lines of text. It is particularly useful when working with text files that have been encoded in binary format. By converting the binary data into text lines, it becomes easier to read and manipulate the data in Power BI.

The M Code Behind Lines.FromBinary

The M code behind Lines.FromBinary is relatively simple. It consists of a single line of code that uses the Binary.Buffer and Text.FromBinary functions to convert the binary data into text lines. Here is the code:

```
let
  Source = Binary.Buffer(BinaryData),
  Result = Text.FromBinary(Source)
in
  Result
```

The code starts with the 'let' statement, which creates a variable called 'Source' and assigns it the value of the binary data. The Binary.Buffer function is used to ensure that the binary data is loaded into memory before the conversion process begins. This can help to improve the performance of the function.

Next, the 'Result' variable is created and assigned the value of the Text.FromBinary function. This function takes the 'Source' variable as its input and converts it into text lines.

Finally, the 'in' statement is used to return the 'Result' variable as the output of the function. This output can then be used in other parts of the Power Query Editor to manipulate the data as required.

Using Lines.FromBinary

To use Lines.FromBinary, you first need to have binary data that you want to convert. This data can be stored in a variety of formats, including files, databases, and APIs.

Once you have the binary data, you can use Power Query Editor to create a new query. In the query editor, select the 'From Binary' option

Lines.FromText

Understanding Lines.FromText

Lines.FromText is a Power Query M function that splits text into lines. When you apply Lines.FromText to a text value, it returns a list of text values, where each value represents a line in the original text. Here's the basic syntax for the Lines.FromText function:

Lines.FromText(text as text, optional delimiter as nullable text)

The first argument, text, is the text value you want to split into lines. The second argument, delimiter, is an optional argument that specifies the character or string that separates the lines in the original text.

The M Code Behind Lines.FromText

To understand the M code behind Lines.FromText, let's take a look at a simple example. Suppose you have the following text value:

"apple#(lf)banana#(lf)orange#(lf)"

This text value represents a list of fruits, where each fruit is on a separate line. To split this text into lines, you can apply Lines.FromText as follows:

Lines.FromText("apple#(lf)banana#(lf)orange#(lf)")

When you apply this function, it returns a list of text values:

Lines.ToBinary

In this article, we will explore the M code behind the Lines.ToBinary function and learn how to use it effectively in our Power Query workflows.

Understanding the Lines.ToBinary Function

The Lines.ToBinary function is a built-in function in Power Query that converts a text string into a binary value. The function takes one argument, which is a text string, and returns a binary value.

The syntax for the Lines.ToBinary function is as follows:

```
Lines.ToBinary(text as text) as binary
```

In this syntax, the text argument is the text string that we want to convert to binary. The function returns a binary value.

Let's look at an example to understand how the Lines.ToBinary function works. Suppose we have a text string "Hello, world!". We can apply the Lines.ToBinary function to this text string using the following code:

```
Lines.ToBinary(“Hello, world!”)
```

This will return the binary value:

```
01101000 01100101 01101100 01101100 01101111 00101100 00100000 01110111 01101111 01110010 01101100 01100100 00100001
```

The binary value is a sequence of 0s and 1s that represent the characters in the text string. Each character in the text string is converted

Lines.ToText

What is Lines.ToText?

The Lines.ToText function is a built-in M function in Power Query that converts a list of text strings into a single text string, separated by line breaks. The syntax for the function is:

```
Lines.ToText(list as list, optional delimiter as nullable text) as text
```

The function takes a list of text strings as its first argument and an optional delimiter as its second argument. If no delimiter is specified, line breaks are used by default.

Understanding the M Code Behind Lines.ToText

To better understand the M code behind the Lines.ToText function, let's take a look at an example. Suppose we have the following table in Power Query:

Column1
Hello
World
Power
Query

We can use the Lines.ToText function to convert the table into a single text string as follows:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMjIwMjJR0lFIyM1NTE2NjcwVdQyNLBQsFqTmJ1ZHhCQlpiTWJycoVIWnFqQUqWkpJSUyfFIXqk6sLk5MSwQA", BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type text) meta [Serialized.Text = true]) in type table [Column1 = _t]),
List.Accumulate
```

Understanding List.Accumulate

Before we dive into the M code behind List.Accumulate, it is important to understand how the function works. List.Accumulate takes three arguments:

– The first argument is a list of values that the function will process.

– The second argument is an initial state value that will be used as the starting point for the accumulation.

– The third argument is a function that will be applied to each element in the list, along with the current accumulated state value.

The function will process each element in the list, applying the custom function to the current element and the accumulated state value. The result of this operation will become the new accumulated state value, which will be used as the input for the next iteration of the function. This process is repeated until all elements in the list have been processed, at which point the final accumulated state value is returned as the result.

The M Code Behind List.Accumulate

The M code behind List.Accumulate is relatively simple, but it can be a bit daunting for those who are new to Power Query or programming in general. Here is an example of the M code for a simple List.Accumulate function:

```
let
   Source = {1..10},
   AccumulateFunction = (state, current) => state + current,
   Result = List.Accumulate(Source, 0, AccumulateFunction)
in
   Result
```

In this example, we are using List.Accumulate to find the sum of a list of numbers from 1 to 10. Here is what each line of the code does:

- The first line sets the Source variable to a list of values from 1 to 10.

- The second line defines the AccumulateFunction variable, which is a custom function that takes two arguments (state and current) and returns the result of adding them together.

List.AllTrue

In this article, we will take a closer look at the M code behind the Power Query M function List.AllTrue. This function is used to determine if all the values in a list are true. Let’s dive in!

Syntax

Before we dive into the M code, let’s look at the syntax of the List.AllTrue function. The function takes a single argument, which is a list of values. It returns a Boolean value that indicates whether all the values in the list are true.

Here’s the syntax of the List.AllTrue function:

```
List.AllTrue(list as list) as logical
```

M Code Explanation

Now that we know the syntax of the List.AllTrue function, let's take a closer look at the M code behind it.

The M code for the List.AllTrue function is actually quite simple. Here's the code:

```
(list) => List.AllTrue(list)
```

The code consists of a lambda expression that takes a single argument, which is a list of values. The lambda expression then calls the List.AllTrue function, passing in the list of values as an argument. The List.AllTrue function returns a Boolean value that indicates whether all the values in the list are true. The lambda expression then returns this Boolean value as the result of the function.

How List.AllTrue Works

Now that we know what the M code behind the List.AllTrue function looks like, let's take a closer look at how the function works.

The List.AllTrue function works by iterating through all the values in the input list. If any of the values are false, the function returns false. If all the values in the list are true, the function returns true.

Here's an example of how the List.AllTrue function works:

List.Alternate

Overview of List.Alternate Function
The List.Alternate function has the following syntax:

List.Alternate(list1 as list, list2 as list) as list

The function takes two lists as inputs and returns a new list that contains the elements from both lists, alternating one from each list. The two lists must be of the same length, or an error will occur.
For example, suppose we have the following two lists:

list1 = {1, 2, 3}
list2 = {4, 5, 6}

The List.Alternate function applied to these lists will return the following result:

{1, 4, 2, 5, 3, 6}

Understanding the M Code Behind List.Alternate Function
To understand how the List.Alternate function works, we need to take a closer look at the M code behind it. In Power Query, all functions are written in M, a functional programming language. The M code for the List.Alternate function is as follows:

List.AnyTrue

Understanding the List.AnyTrue Function

Before we dive into the M code behind the List.AnyTrue function, let’s first understand what this function does. The List.AnyTrue function takes a list of logical values and returns true if any of the values in the list are true. If all the values are false, it returns false.

Here is an example of using the List.AnyTrue function in Power Query:

```
let
   myList = {true, false, true},
   result = List.AnyTrue(myList)
in
   result
```

In this example, we have a list of three logical values: true, false, and true. We then pass this list to the List.AnyTrue function, which returns true because at least one of the values in the list is true.

The M Code Behind List.AnyTrue

Now that we understand what the List.AnyTrue function does, let's take a look at the M code behind this function. The code for the List.AnyTrue function is as follows:

```
(List as list) as logical =>
   List.Contains(List.Transform(List, each if _ then 1 else 0), 1)
```

Let's break this code down and understand what each part does:

`(List as list) as logical` - This is the function signature, which specifies that the List.AnyTrue function takes a list as input and returns a logical value.

List.Average

What is the List.Average Function?

The List.Average function is a Power Query M function that is used to calculate the average of a list of numbers. It takes a list of numbers as input and returns the average value of those numbers. The syntax of the List.Average function is as follows:

```
List.Average(list as list) as number
```

Here, `list` is the input list of numbers, and the function returns a `number` that represents the average value of the list.

How Does the List.Average Function Work?

The List.Average function works by first summing all the numbers in the input list and then dividing the sum by the number of items in the list. Here is the M code behind the List.Average function:

```
(list) => List.Sum(list) / List.Count(list)
```

As you can see, the function takes the input list as a parameter, sums the list using the List.Sum function, and divides the sum by the number of items in the list using the List.Count function.

Examples of Using the List.Average Function

Here are some examples of using the List.Average function to calculate the average of a list of numbers:

Example 1

Suppose we have the following list of numbers:

```
{1, 2, 3, 4, 5}
```

List.Buffer

Introduction to List.Buffer

List.Buffer is a M function in Power Query that can significantly improve the performance of your data transformations. This function is used to create a buffer for a list, which means that the list will be loaded into memory and held there until the buffer is released.

One of the main advantages of using List.Buffer is that it can help reduce the number of queries that are sent to the data source. This is because List.Buffer creates a cache for the data, allowing you to perform multiple transformations on the data without having to re-query the data source.

How to Use List.Buffer

To use List.Buffer, simply call the function and pass in a list as the argument. For example, let's say you have a table called "SalesData" that contains sales data, and you want to create a buffer for the "SalesAmount" column. You can do this by creating a new column and using the List.Buffer function, as shown below:

```
= Table.AddColumn(SalesData, "BufferedSalesAmount", each List.Buffer([SalesAmount]))
```

In the above example, we are adding a new column called "BufferedSalesAmount" to the SalesData table. We are using the List.Buffer function to create a buffer for the "SalesAmount" column. This means that the "SalesAmount" column will be loaded into memory and held there until the buffer is released.

Benefits of Using List.Buffer

List.Buffer has several benefits that make it a popular function among Power Query users. Some of the main benefits include:

Improved Performance

As mentioned earlier, List.Buffer can significantly improve the performance of your data transformations. By creating a buffer for a list, you can reduce the number of queries that are sent to the data source and perform multiple transformations on the data without having to re-query the data source.

Reduced Memory Consumption

List.Buffer can also help reduce the amount of memory that is used by your data transformations. This is because the function only loads the data that is needed into memory, rather than loading the entire dataset into memory.

List.Combine

Introduction to the List.Combine Function

The List.Combine function is a part of the Power Query M language, which is used for data transformation and analysis. This function allows you to combine multiple lists into one, which can be useful when working with large datasets.

One of the key advantages of the List.Combine function is that it can handle lists of different lengths and types. This means that you can easily combine lists with different structures and formats, without having to worry about compatibility issues.

Understanding the M Code Behind List.Combine

To understand the M code behind the List.Combine function, let's take a look at a simple example. Suppose we have two lists, List1 and List2, which contain the following data:

List1 = {1, 2, 3}
List2 = {"A", "B", "C"}

To combine these lists into a single list, we can use the List.Combine function as follows:

List.Combine({List1, List2})

The output of this function would be a new list with the following data:

{1, 2, 3, "A", "B", "C"}

The M code behind the List.Combine function is relatively simple. It takes a list of lists as its input and returns a single list that contains

List.ConformToPageReader

In this article, we will take a closer look at the List.ConformToPageReader M function, which is used to transform a list of records into a table that is suitable for display in a Power BI report.

Understanding the List.ConformToPageReader Function

The List.ConformToPageReader function is used to convert a list of records into a table that can be displayed in a Power BI report. This function takes two arguments: the list of records to be converted, and a function that provides information about the schema of the resulting table.

The first argument to the List.ConformToPageReader function is a list of records. Each record in the list represents a row in the resulting table. The records can have different fields, but they must have the same number of fields and the fields must be of the same data type.

The second argument to the function is a schema function. This function takes no arguments and returns a record that describes the schema of the resulting table. The record must have one field for each column in the table, and the name and data type of each field must match the corresponding field in the records that make up the list.

Example Usage of List.ConformToPageReader

To illustrate the usage of the List.ConformToPageReader function, consider the following example. Suppose we have a list of records that represent sales data for different products and regions. Each record has three fields: Product, Region, and SalesAmount.

```
[
  [Product = “Product A”, Region = “North”, SalesAmount = 1000],
  [Product = “Product A”, Region = “South”, SalesAmount = 2000],
  [Product = “Product B”, Region = “North”, SalesAmount = 1500],
  [Product = “Product B”, Region = “South”, SalesAmount = 2500]
]
```

To convert this list of records into a table, we can use the List.ConformToPageReader function as follows:

List.Contains

Understanding the List.Contains Function

The List.Contains function in Power Query is used to determine if a specified value is present in a list. It takes two arguments: the list to be searched, and the value to be found. If the value is found in the list, the function returns true; otherwise, it returns false.

Here is an example of how the List.Contains function is used:

```
let
  myList = {1, 2, 3, 4, 5},
  result = List.Contains(myList, 3)
in
  result
```

In this example, we define a list called myList that contains the values 1 through 5. We then use the List.Contains function to search for the value 3 in myList. The result of the function will be true, since the value 3 is present in the list.

The M Code Behind List.Contains

The actual M code behind the List.Contains function is quite simple. Here is the code:

```
List.Contains = (list as list, value as any) as logical =>
  if List.IsEmpty(list) then
    false
  else if list{0} = value then
    true
  else
    List.Contains(List.RemoveFirstN(list, 1), value)
```

List.ContainsAll

In this article, we'll take a closer look at List.ContainsAll and explore its M code. We'll also provide some practical examples to help you understand how the function works and how you can use it in your own data transformations.

What is List.ContainsAll?

List.ContainsAll is a Power Query M function that takes two lists as arguments and returns a Boolean value. The function checks whether all the elements of the first list appear in the second list. If they do, the function returns true; otherwise, it returns false.

Here's the syntax for List.ContainsAll:

```
List.ContainsAll(list1 as list, list2 as list) as logical
```

As you can see, the function takes two arguments: list1 and list2. List1 is the list of elements to look for, and list2 is the list to search in. The function returns a Boolean value that indicates whether all the elements in list1 are present in list2.

How List.ContainsAll Works

To understand how List.ContainsAll works, let's take a look at some examples. Suppose we have two lists:

```
list1 = {“apple”, “banana”, “orange”}
list2 = {“pear”, “apple”, “mango”, “banana”, “kiwi”}
```

If we apply List.ContainsAll to these two lists, we get the following result:

```
List.ContainsAll(list1, list2)
```

List.ContainsAny

What is the List.ContainsAny Function?

The List.ContainsAny function is a built-in function in Power Query's M language. Its primary purpose is to check whether a list contains any of the items in another list. The function takes two arguments: the first argument is the list to check, and the second argument is the list of items to look for.

The syntax for List.ContainsAny is as follows:

```
List.ContainsAny(list as list, values as list) as logical
```

The function returns a logical value (true or false) depending on whether any of the items in the second list are found in the first list.

How Does List.ContainsAny Work?

List.ContainsAny works by iterating over each item in the second list and checking whether it is present in the first list. If any of the items in the second list are found in the first list, the function returns true. Otherwise, it returns false.

Here is an example of how List.ContainsAny works:

```
let
   list1 = {“apple”, “banana”, “orange”},
   list2 = {“banana”, “pear”},
   result = List.ContainsAny(list1, list2)
in
   result
```

In this example, the function checks whether any items in list2 ("banana" and "pear") are found in list1 ("apple", "banana", "orange"). Since "banana" is found in list1, the function will return true.

List.Count

What is the M language?

Before we dive into the M code behind List.Count, let’s first discuss what the M language is. M is the formula language used by Power Query, which allows users to create custom functions and formulas to manipulate data. The M language is a functional language, which means that it focuses on the evaluation of expressions rather than the execution of commands.

List.Count function overview

The List.Count function is used to return the number of items in a list. It takes one argument, which is the list to count. Here is an example of using List.Count to count the number of items in a list:

```
let
  myList = {1, 2, 3},
  count = List.Count(myList)
in
  count
```

In this example, we define a list called `myList` with three items and then call the List.Count function on it to get the count of items. The result of this expression is `3`.

The M code behind List.Count

Now that we've seen how List.Count works, let's take a closer look at the M code behind it. Here is the M code for the List.Count function:

```
(List as list) as number =>
  List.Count(List)
```

List.Covariance

What is Power Query M Language?

Power Query M language is a functional language used to query and transform data. It is used in Microsoft Excel, Power BI, and other data analysis tools. The language is designed to be simple and easy to learn, with a syntax similar to Excel formulas. The language is used to create custom functions, automate data transformation, and extract data from various sources.

Understanding Covariance

Before we dive into the List.Covariance function, let's take a moment to understand covariance. Covariance is a measure of the relationship between two variables. It is calculated as the average of the product of the deviations of each variable from its mean. A positive covariance indicates that the two variables tend to move together, while a negative covariance indicates that they tend to move in opposite directions.

Syntax of List.Covariance Function

The List.Covariance function in Power Query M language takes two lists of numbers as input and returns the covariance between them. The syntax of the function is as follows:

```
List.Covariance(list1 as list, list2 as list, optional biased as nullable logical) as number
```

- list1: The first list of numbers.
- list2: The second list of numbers.
- biased: An optional parameter that indicates whether to use a biased or unbiased estimator of covariance. If this parameter is omitted, the function uses a biased estimator by default.

How to Use List.Covariance Function

To use the List.Covariance function, you first need to create two lists of numbers. These lists can be created using other Power Query functions or by manually entering the data. Once you have the two lists, you can call the List.Covariance function and pass the two lists as arguments.

Here is an example of using the List.Covariance function to calculate the covariance between two lists of numbers:

List.Dates

Overview of List.Dates

Before we get into the M code, let’s briefly review what List.Dates does. As mentioned, it generates a list of dates within a specified range. Here’s the syntax:

```
List.Dates(startDate as date, numberOfDays as number, optional stepSize as number) as list
```

`startDate`: The starting date for the list.
`numberOfDays`: The number of days to include in the list.
`stepSize` (optional): The number of days between each date in the list. Default is 1.

Here's an example usage of List.Dates:

```
List.Dates(#date(2021,1,1), 7)
```

This would generate a list of 7 dates starting from January 1, 2021: `{#date(2021,1,1), #date(2021,1,2), #date(2021,1,3), #date(2021,1,4), #date(2021,1,5), #date(2021,1,6), #date(2021,1,7)}`.

The M code behind List.Dates

So how does List.Dates actually generate this list of dates? Let's take a closer look at the M code behind the function.

Here's the basic structure of the M code for List.Dates:

```
(startDate as date, numberOfDays as number, optional stepSize as number) =>
  let
    dates = List.Generate(
List.DateTimes
```

What is List.DateTimes?

List.DateTimes is a built-in M function in Power Query that generates a list of datetime values. It takes three arguments:

1. Start: The start datetime value
2. Count: The number of datetime values to generate
3. Interval: The time interval between each datetime value

For example, the following code generates a list of 10 datetime values starting from January 1, 2021, with an interval of 1 hour:

```
List.DateTimes(#datetime(2021, 1, 1, 0, 0, 0), 10, #duration(0, 1, 0, 0))
```

The output of this code will be a list of datetime values in the following format:

```
{
 #datetime(2021, 1, 1, 0, 0, 0),
 #datetime(2021, 1, 1, 1, 0, 0),
 #datetime(2021, 1, 1, 2, 0, 0),
 ...
 #datetime(2021, 1, 1, 9, 0, 0)
}
```

List.DateTimes is a very useful function for generating datetime ranges that can be used for various data analysis and reporting tasks.

The M Code Behind List.DateTimes

The M code behind List.DateTimes is fairly simple and can be broken down into three steps:

1. Calculate the duration between each datetime value

List.DateTimeZones

In this article, we will explore the M code behind the List.DateTimeZones function and how it can be used to transform and load data.

What is List.DateTimeZones?

List.DateTimeZones is a Power Query M function that returns a list of all time zones. A time zone is a region of the Earth where the same standard time is used. Time zones are important in many applications, including financial and travel-related applications.

The List.DateTimeZones function returns a list of time zones in the format of a table with two columns: “Name” and “Timezone”. The “Name” column contains the name of the time zone, while the “Timezone” column contains the UTC offset of the time zone.

The M Code Behind List.DateTimeZones

The M code behind the List.DateTimeZones function is straightforward. It uses the TimeZoneInfo class in .NET to retrieve all available time zones. The code can be viewed by opening the Advanced Editor in Power Query M and searching for the List.DateTimeZones function.

```
let
  Source = #table(
    {“Name”, “Timezone”},
    List.Transform(
      TimeZoneInfo.GetSystemTimeZones(),
      each {
        .DisplayName,
        Duration.TotalHours(.BaseUtcOffset)
      }
    )
  )
in
  Source
```

List.Difference

One of the functions available in M is List.Difference, which allows you to compare two lists and return the items that are unique to each list. In this article, we will explore the M code behind the List.Difference function and how it can be used in Power Query.

Understanding the List.Difference Function

The List.Difference function takes two lists as inputs and returns a new list that contains the items that are unique to each list. In other words, it returns the items that are in one list but not in the other.

Here is the syntax for the List.Difference function:

```
List.Difference(list1 as list, list2 as list) as list
```

The first argument, list1, is the list that you want to compare to the second list, list2. The second argument, list2, is the list that you want to compare to list1. The function then returns a new list that contains the items that are unique to each list.

Example Usage

Let's take a look at an example of how the List.Difference function can be used in Power Query. Suppose we have two lists, list1 and list2, as shown below:

```
list1 = {“apple”, “banana”, “orange”, “pear”}
list2 = {“banana”, “kiwi”, “orange”, “grape”}
```

We can use the List.Difference function to compare these two lists and return the items that are unique to each list. Here is the M code to do this:

```
let
List.Distinct
```

Overview of List.Distinct

The List.Distinct function is part of the M language, which is the programming language used in Power Query. The function takes a list as its input and returns a new list with only the unique values from the original list. For example, if you have a column with the following values:

| Column A |
|———-|
| Blue |
| Red |
| Green |
| Blue |

Using List.Distinct on this column would return a new column with the values:

| Column A |
|———-|
| Blue |
| Red |
| Green |

The M Code Behind List.Distinct

The M code for the List.Distinct function is relatively simple. It consists of a single line of code that specifies the input list and the output list. Here is the basic syntax for List.Distinct:

```
List.Distinct(list as list) as list
```

The "list as list" parameter specifies the input list, while the "as list" at the end of the function specifies that the output should be a list. To use the List.Distinct function, you would replace "list" with the name of the column or list you want to retrieve unique values from. For example, if you have a table called "Sales" with a column called "Region", you could use List.Distinct to retrieve a list of unique

List.Durations

One of the commonly used M functions in Power Query is List.Durations. This function is used to calculate the duration between two dates and times. In this article, we will explore the M code behind the List.Durations function.

Understanding List.Durations Function

Before we dive into the M code behind the List.Durations function, let's first understand how it works. The List.Durations function takes two lists of dates and times as input. It then calculates the duration between each pair of dates and times in the lists and returns a list of durations.

Here is the syntax of the List.Durations function:

```
List.Durations(startList as list, endList as list, optional durationUnits as nullable any) as list
```

The startList and endList are the input lists of dates and times. The durationUnits parameter is optional and specifies the units of the duration returned by the function. If the durationUnits parameter is not specified, the function returns the duration in days.

Now that we have a basic understanding of the List.Durations function let's look at the M code behind it.

The M Code Behind the List.Durations Function

The M code behind the List.Durations function is a combination of several M functions that work together to calculate the duration between two given dates and times.

Here is an example of the M code behind the List.Durations function:

```
let
  startList = {#datetime(2021, 1, 1, 0, 0, 0), #datetime(2021, 1, 2, 0, 0, 0), #datetime(2021, 1, 3, 0, 0, 0)},
  endList = {#datetime(2021, 1, 4, 0, 0, 0), #datetime(2021, 1, 5, 0, 0, 0), #datetime(2021, 1, 6, 0, 0, 0)},
  durationList = List.Transform(List.Zip({startList, endList}), each Duration.From([{1},{0}] – [{0},{1}]))
in
  durationList
```

List.FindText

What is the List.FindText Function?

The List.FindText function is used to search for a specific text string within a list of values. It returns the index of the first item in the list that contains the specified text. This function is particularly useful when working with large datasets where manual searching is not practical.

Syntax

The syntax for the List.FindText function is as follows:

```
List.FindText(list as list, searchText as text, optional occurrence as nullable number) as nullable number
```

- list - The list of values to search for the specified text.
- searchText - The text to search for within the list of values.
- occurrence - (optional) The occurrence of the specified text to search for. Default value is 1.

M Code Explanation

The List.FindText function is comprised of several steps that work together to search for the specified text within a list of values. Let's break down the M code behind this function:

```
let
  Source = (list as list, searchText as text, optional occurrence as nullable number) =>
  let
    ListLength = List.Count(list),
    OccurrenceCount = List.Generate(
      () => [Count = 0, Index = -1],
      each [Count] < ListLength,
      each [Count = [Count] + 1, Index = List.FindText(List.Skip(list, [Index] + 1), searchText) + [Index] + 1],
List.First
```

In this article, we will explore the M code behind the List.First function and how it can be used to extract the first element of a list in Power Query.

Understanding Lists in Power Query

Before we dive into the List.First function, let's first understand what lists are in Power Query. A list is an ordered collection of values, where each value can be of a different type. A list is enclosed in curly braces {} and each value is separated by a comma. Here's an example of a list in Power Query:

```
{1, "Apple", true, #date(2022,1,1)}
```

As you can see, this list contains four values, each of a different type - an integer, a string, a boolean, and a date.

The List.First Function

Now that we understand what lists are in Power Query, let's take a closer look at the List.First function. The List.First function is used to extract the first element of a list. It takes a list as an input and returns the first element of that list.

Here's the syntax of the List.First function:

```
List.First(list as list) as any
```

As you can see, the List.First function takes a list as an input and returns the first element of that list. The function returns an "any" type, which means that it can return any type of value.

How the List.First Function Works

Now that we understand the syntax of the List.First function, let's take a deeper look at how it works. The List.First function works by taking the input list and returning the first element of that list.

Here's an example of how the List.First function works:

List.FirstN

But how exactly does List.FirstN work? In this article, we'll explore the M code behind this powerful function and provide some examples of how it can be used.

Understanding List.FirstN

Before we dive into the M code behind List.FirstN, let's first take a closer look at the function itself. List.FirstN takes two inputs: the list you want to extract items from, and the number of items you want to extract. Here's an example of how it's used:

```
let
  myList = {1, 2, 3, 4, 5},
  firstThree = List.FirstN(myList, 3)
in
  firstThree
```

In this example, we're creating a list called "myList" that contains the numbers 1 through 5. We then use List.FirstN to extract the first three items from this list. The result is a new list containing only the numbers 1, 2, and 3.

The M Code Behind List.FirstN

Now that we understand how List.FirstN works, let's take a look at the M code behind it. Here's the basic structure of the function:

```
List.FirstN(list as list, count as number) as list =>
  let
    result = List.FirstN(list, count)
  in
    result
```

List.Generate

Understanding List.Generate

The List.Generate function is used to generate a list based on a set of rules. It takes four arguments:

– The initial value of the list

– A function to test whether the list should continue to be generated

– A function to generate the next element in the list

– An optional function to transform the final list

Here’s an example of how the List.Generate function works:

```
List.Generate(
  () => 1, // initial value
  each _ <= 5, // continue generating as long as the value is less than or equal to 5
  each _  2 // generate the next value by multiplying the previous value by 2
)
```

This code generates the following list: `[1, 2, 4, 8, 16]` .

The M Code Behind List.Generate

Now let's take a look at the M code that powers the List.Generate function.

```
(listGenerator as function, optional resultSelector as function) =>
let
   step = (state, accumulated) =>
      let
         next = listGenerator(state),
         newAccumulated = accumulated & {next}
```

List.InsertRange

What is the List.InsertRange Function?

The List.InsertRange function in Power Query is used to insert a range of values into a list at a specified index. It takes three arguments:

– The list to insert the range into

– The index at which to insert the range

– The range of values to insert

Here is an example of how to use the List.InsertRange function in Power Query:

```
let
  myList = {1, 2, 3, 4, 5},
  myRange = {10, 11, 12},
  insertIndex = 2,
  outputList = List.InsertRange(myList, insertIndex, myRange)
in
  outputList
```

In this example, we have a list called `myList` that contains the values 1 through 5. We also have a range of values called `myRange` that contains the values 10 through 12. We want to insert `myRange` into `myList` at index 2. The `List.InsertRange` function is used to accomplish this. The resulting list will be `{1, 2, 10, 11, 12, 3, 4, 5}`.

Understanding the M Code Behind List.InsertRange

The M code behind the List.InsertRange function is relatively simple. Here is the basic syntax of the function:

```
List.InsertRange(list as list, index as number, values as list) as list
```

List.Intersect

List.Intersect is particularly useful when you need to compare data from two different sources, such as a database and an Excel spreadsheet. In this article, we'll take a closer look at the M code behind List.Intersect, and explore how it works.

What is List.Intersect?

List.Intersect is a function in Power Query that takes two lists as input and returns a new list containing only the elements that are common to both lists. For example, if we have two lists:

ListA = {1, 2, 3, 4, 5}
ListB = {4, 5, 6, 7, 8}

We can use List.Intersect to find the common elements:

```
List.Intersect(ListA, ListB)
```

This will return a new list containing the values 4 and 5.

How Does List.Intersect Work?

List.Intersect works by comparing each element in the first list with each element in the second list. If an element is found in both lists, it is added to the new list that is returned by the function.

The M code behind List.Intersect is relatively simple. Here's what it looks like:

```
(List1, List2) =>
 let
   Intersection = List.Distinct(List.Intersect(List1, List2))
List.IsDistinct
```

In this article, we will explore the M code behind the List.IsDistinct function and how it works.

Understanding Lists in Power Query

Before we dive into the List.IsDistinct function, we must first understand what a list is in Power Query. In Power Query, a list is a collection of values of the same data type. Lists can be created manually by typing in values or by using other functions to generate them.

For example, the following M code creates a list of numbers from 1 to 10:

```
let
  myList = {1,2,3,4,5,6,7,8,9,10}
in
  myList
```

Lists can also be created from columns in a table. For example, the following M code creates a list of unique values from a column in a table:

```
let
  Source = Excel.CurrentWorkbook(){[Name=”Table1"]}[Content],
  myList = List.Distinct(Source[Column1])
in
  myList
```

The List.IsDistinct Function

The List.IsDistinct function is used to check if a list has distinct values or not. It takes a list as its input and returns a Boolean value (true

List.IsEmpty

Understanding the List.IsEmpty Function

Before we dive into the technicalities, let us first comprehend the `List.IsEmpty` function's basic concept. It is a simple yet powerful function that returns true if a list or table is empty and false if it is not. The following example demonstrates the function's use:

```
= List.IsEmpty({})
```

Here, the function returns true because the curly brackets {} define an empty list. Similarly, the following function returns false:

```
= List.IsEmpty({1,2,3})
```

Here, the function returns false because the list contains elements. The `List.IsEmpty` function can also be used with tables and records, as illustrated in the following examples:

```
= List.IsEmpty(#table({},{}))
```

The above function returns true because the table contains no rows or columns. On the other hand, the following function returns false:

```
= List.IsEmpty([Name = “John”, Age = 25])
```

List.Last

Introduction to List.Last Function

The List.Last function is used to return the last item in a list. It takes a list as an argument and returns the last item in the list. The syntax for the List.Last function is as follows:

```
List.Last(list as list) as any
```

The argument "list as list" specifies the list you want to return the last item from. The function returns the last item in the list as "any".

The M Code Behind List.Last

To understand the M code behind List.Last, we need to understand some basic concepts of M language. M is a functional language that is used to define custom functions. It is used in Power Query to manipulate data and perform various transformations.

The M code behind List.Last is as follows:

```
(list as list) =>
  let
    numberOfItems = List.Count(list),
    result = List.Buffer(list){numberOfItems-1}
  in
    result
```

Let's break down this code to understand how it works.

The Arguments

The function takes only one argument, which is "list" as list. This specifies the list that we want to return the last item from.

The Variables

List.LastN

What is List.LastN?

List.LastN is a function that allows users to extract the last N items from a list in Power Query. The function takes two arguments: the list to extract from and the number of items to extract. For example, if we have a list of numbers {1, 2, 3, 4, 5} and we want to extract the last two items, we would use the function List.LastN({1, 2, 3, 4, 5}, 2), which would return the list {4, 5}.

How List.LastN Works

To understand how List.LastN works, we need to look at the M code behind the function. The M code for List.LastN is:

```
(list as list, count as number) as list =>
  let
    lastIndex = List.Count(list),
    firstIndex = lastIndex – count + 1
  in
    List.Range(list, firstIndex, count)
```

Let's break down the code line by line:

```
(list as list, count as number) as list =>
```

This line defines the function and its arguments. The function takes two arguments: a list (which must be a list of values) and a count (which must be a number). The function returns a list.

```
let
```

List.MatchesAll

Syntax

The syntax for the List.MatchesAll function is as follows:

List.MatchesAll(list1 as list, list2 as list) as logical

The first argument, list1, is the list that we want to check for matches. The second argument, list2, is the list that we want to compare against. The function returns a logical value, which is TRUE if all the elements in list1 match the corresponding elements in list2, and FALSE otherwise.

Example

Let's take a look at an example to illustrate how the List.MatchesAll function works. Suppose we have two lists:

list1 = {“apple”, “banana”, “orange”}
list2 = {“apple”, “banana”, “orange”}

We can use the List.MatchesAll function to check whether all the elements in list1 match the corresponding elements in list2:

List.MatchesAll(list1, list2)

This will return TRUE, since all the elements in list1 match the corresponding elements in list2.

Practical Applications

The List.MatchesAll function can be useful in a variety of scenarios. For example, suppose we have a dataset containing information

List.MatchesAny

Understanding List.MatchesAny

List.MatchesAny is a function in Power Query M that takes two arguments: a value and a list of values. It returns a Boolean value indicating whether the value is contained in the list. Here is the syntax of the List.MatchesAny function:

```
List.MatchesAny(list as list, expression as any, optional equationCriteria as any) as logical
```

The first argument, list, is the list of values that you want to check. The second argument, expression, is the value that you want to check for. The third argument, equationCriteria, is optional and allows you to specify a custom comparison function.

The List.MatchesAny function is case-sensitive by default. This means that if you are checking for a string value, the case of the value must match exactly with the case of the value in the list. However, you can change this behavior by providing a custom comparison function using the equationCriteria argument.

The M Code Behind List.MatchesAny

Now that we have a basic understanding of the List.MatchesAny function, let's take a look at the M code behind it. The M code for List.MatchesAny is relatively simple and can be broken down into two parts:

```
List.MatchesAny = (list as list, expression as any, optional equationCriteria as any) as logical =>
   List.Contains(List.Transform(list, each equationCriteria(_, expression)), true)
```

The first part of the code defines the List.MatchesAny function and its arguments. The second part of the code is the actual implementation of the function.

The List.MatchesAny function is essentially a wrapper function that calls the List.Contains function with a transformed list. The List.Transform function is used to apply the equationCriteria function to each element in the list, which returns a list of Boolean values indicating whether each element matches the expression. The List.Contains function is then called to check whether any of the values

List.Max

In this article, we will explore the M code behind the List.Max function, and how you can use it to manipulate data in Power Query.

Understanding the List.Max Function

The List.Max function is used to find the maximum value in a list of numbers. The syntax for the function is as follows:

```
List.Max(list as list) as any
```

The function takes a list of numbers as its argument, and returns the maximum value in that list. If the list is empty, the function returns null.

The M Code Behind List.Max

The M code behind the List.Max function is relatively simple. Here is the M code for the function:

```
let
  List.Max = (list as list) =>
    List.MaxOrNull(list),
  List.MaxOrNull = (list as list) =>
    List.Sort(list, Order.Descending){0}
in
  List.Max
```

The code consists of two functions: List.Max and List.MaxOrNull.

The List.Max function calls the List.MaxOrNull function to find the maximum value in the list. If the list is empty, List.Max returns null.

The List.MaxOrNull function sorts the list in descending order, and returns the first value in the sorted list. This value is the maximum value in the original list.

List.MaxN

What is List.MaxN?

List.MaxN is a Power Query M function that takes two arguments: a list and a number n. It returns the maximum n values from the list. For example, if we have a list {1, 5, 2, 4, 3} and we want to get the top three values, we can use List.MaxN(list, 3) which will return {5, 4, 3}.

The M code behind List.MaxN

The M code behind List.MaxN is quite simple. It first sorts the list in descending order and then takes the first n values from the sorted list. Here is the M code:

```
(list as list, n as number) =>
let
  sortedList = List.Sort(list, (a, b) => b – a),
  result = List.FirstN(sortedList, n)
in
  result
```

Let's break down this code step by step:

1. It takes two arguments: list and n.

```
  (list as list, n as number) =>
```

2. It sorts the list in descending order using List.Sort function and a lambda function. In this lambda function, we subtract b from a to sort the list in descending order.

List.Median

The M code behind the List.Median function is relatively simple and easy to understand. Here, we'll take a closer look at how the function works and the M code that powers it.

Understanding the Median Function

Before we delve into the M code, let's first review what the median is and how it works. The median is the middle value in a list of numbers. To calculate the median, we first need to sort the list in ascending order. If the list has an odd number of values, the median is the middle value. If the list has an even number of values, the median is the average of the two middle values.

For example, consider the list of numbers [5, 2, 8, 1, 6]. To calculate the median, we first sort the list in ascending order: [1, 2, 5, 6, 8]. The list has an odd number of values, so the median is the middle value, which is 5.

Now, let's see how this process is performed using the List.Median function in Power Query.

The M code Behind List.Median

The List.Median function takes a list of numbers as its argument and returns the median value. The M code behind the function is as follows:

```
(List as list) =>
let
  count = List.Count(List),
  mid = Number.RoundUp(count / 2),
  sorted = List.Sort(List),
  median = if count mod 2 = 0 then
        (sorted{mid} + sorted{mid + 1}) / 2
      else
        sorted{mid}
in
  median
```

List.Min

What is the List.Min Function?

Before we get into the M code behind the List.Min function, let’s first review what this function actually does. The List.Min function is used to find the smallest value in a list of numbers or values. Here’s an example of how the List.Min function can be used in Power Query:

```
let
   myList = {4, 6, 2, 8, 3},
   minVal = List.Min(myList)
in
   minVal
```

In this example, we define a list of numbers called "myList" and then use the List.Min function to find the smallest value in that list. The result of this query would be 2, since that is the smallest value in the list.

The M Code Behind List.Min

So how does the List.Min function actually work? Let's take a look at the M code behind this function to find out. Here's the M code for the List.Min function:

```
(List) => List.Combine(List.Select(List, each _ <> null and not Number.IsNaN(_)), (x, y) => if x < y then x else y)
```

Let's break this code down piece by piece to see how it works.

The Input Parameter

The List.Min function takes a single input parameter, which is a list of values. In the M code, this parameter is represented by the "List" variable.

List.MinN

What is List.MinN Function?

List.MinN is a function in Power Query M language that is used to find the smallest n number of items in a list. The function takes two arguments: the list to be processed and the number of smallest items to be returned.

The syntax of the function is as follows:

List.MinN(list as list, optional count as number) as list

Here, the list argument is the list to be processed, and the count argument is the number of smallest items to be returned. If the count argument is not provided, the function returns the smallest item in the list.

How List.MinN Function Works?

The List.MinN function works by first sorting the list in ascending order and then selecting the smallest n items from the list. If the count argument is not provided, the function returns only the smallest item in the list.

Let's take an example to understand how List.MinN function works. Consider the following list:

{5, 2, 6, 3, 1, 8, 9}

If we want to find the smallest three items from this list, we can use the List.MinN function as follows:

List.MinN({5, 2, 6, 3, 1, 8, 9}, 3)

The function will first sort the list in ascending order:

List.Mode

What is the List.Mode Function?

The List.Mode function is used to find the most frequently occurring value in a list of values. It takes a list as its input and returns the most common value in that list. If there are multiple values that occur with the same frequency, List.Mode returns the first value that it encounters.

How Does List.Mode Work?

The List.Mode function works by first sorting the list in ascending order, then counting the number of occurrences of each value in the list. Once it has counted all of the occurrences, it returns the value with the highest count.

Here is an example of how the List.Mode function works:

Suppose we have the following list of values:

{1, 2, 3, 4, 3, 2, 1, 4, 5, 3}

The List.Mode function would first sort this list in ascending order:

{1, 1, 2, 2, 3, 3, 3, 4, 4, 5}

Next, it would count the number of occurrences of each value in the list:

{2, 2, 2, 3, 2}

Finally, it would return the value with the highest count, which is 3.

List.Modes

In this article, we will take a deep dive into the M code behind one of the most commonly used M functions, List.Modes. We will explore its syntax, arguments, and usage scenarios.

Syntax of List.Modes

The List.Modes function in Power Query is used to return a list of the most frequently occurring values in a given list. The syntax of List.Modes is as follows:

```
List.Modes(list as list, optional equationCriteria as any)
```

The first argument, 'list', is a required argument that specifies the list of values for which we want to find the most frequent values. The second argument, 'optional equationCriteria', is an optional argument that specifies the criteria for comparing the values in the list.

Arguments of List.Modes

Let's explore the arguments of List.Modes in more detail.

Argument 1: list

The first argument of List.Modes is a required argument that specifies the list of values for which we want to find the most frequent values. The 'list' argument can be a column of a table, a list of values, or an expression that returns a list.

Argument 2: optional equationCriteria

The second argument of List.Modes is an optional argument that specifies the criteria for comparing the values in the list. The 'equationCriteria' argument can be any value that returns a true or false result when compared with the values in the list.

For example, suppose we have a list of numbers and we want to find the most frequent values in the list. We can use the following formula:

```
List.Modes({1, 2, 3, 4, 4, 5, 5, 5, 6})
```

List.NonNullCount

Understanding the List.NonNullCount function

The List.NonNullCount function is a simple but powerful tool that can be used to quickly and easily count the number of non-null values in a list. Here's the M code for this function:

```
List.NonNullCount(list as list) as number
```

As you can see, this function takes a single argument - a list - and returns a number that represents the count of non-null values in that list. Here's an example of how this function can be used:

```
let
  myList = {1, 2, null, 4, null, 6},
  count = List.NonNullCount(myList)
in
  count
```

In this example, we define a list called `myList` that contains six values, two of which are null. We then use the List.NonNullCount function to count the number of non-null values in this list, which is four. Finally, we assign this count to a variable called `count` and return it.

Using List.NonNullCount in practice

Now that we understand the basics of how the List.NonNullCount function works, let's explore some real-world applications for this function.

Cleaning up messy data

One of the most common use cases for the List.NonNullCount function is cleaning up messy data. When working with large datasets,

In this article, we will explore the M code behind the List.Numbers function and how it can be used to generate sequential numbers in Power Query.

What is the List.Numbers Function?

The List.Numbers function is a Power Query function that is used to generate a list of sequential numbers. It takes three arguments: the starting number, the count, and the optional increment value. The starting number specifies the first number in the sequence, the count specifies how many numbers should be generated, and the increment value specifies the difference between each number in the sequence.

For example, the following List.Numbers function generates a list of ten sequential numbers starting from 1:

```
List.Numbers(1, 10)
```

This will generate the following output:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

The M Code behind the List.Numbers Function

The List.Numbers function is implemented in M code, which is the language used by Power Query to define custom functions. The M code behind the List.Numbers function is as follows:

```
let
  ListOfNumbers = List.Generate(
    () => [Current = start, Counter = 0],
```

List.Percentile

Understanding Percentiles

Before we dive into the M code behind the List.Percentile function, let's first understand what percentiles are. A percentile is a measure used in statistics to indicate the value below which a given percentage of observations in a group of observations fall. For example, the 90th percentile is the value below which 90% of the observations fall.

Syntax of List.Percentile function

The syntax for the List.Percentile function is as follows:

```
List.Percentile(list as list, percentile as number, optional precision as nullable number) as any
```

The List.Percentile function takes in three arguments:

1. `list`: This is the list of values for which we want to calculate the percentile.
2. `percentile`: This is the percentile value that we want to calculate. It should be between 0 and 1.
3. `precision`: This is an optional argument that allows us to specify the number of decimal places to round the result to.

M Code Behind the List.Percentile Function

The List.Percentile function is implemented in M code, which is the language used by Power Query. Let's take a look at the M code behind the List.Percentile function:

```
let
  sortedList = List.Sort(list),
  index = percentile  List.Count(sortedList) – 1,
  fraction = Number.Mod(index, 1),
  wholeNumber = Number.RoundDown(index),
  lowerValue = List.Position(sortedList, wholeNumber),
  upperValue = if lowerValue + 1 < List.Count(sortedList) then List.Position(sortedList, lowerValue + 1) else lowerValue,
List.PositionOf
```

Understanding the List.PositionOf function

The List.PositionOf function is used to find the position of an item in a list. This function takes two arguments: the list to search and the item to find. The syntax for this function is as follows:

```
List.PositionOf(list as list, item as any) as nullable number
```

The function returns the position of the item in the list as a nullable number. If the item is not found in the list, the function returns null.

The M code behind List.PositionOf

The M code behind the List.PositionOf function is relatively simple. The function first checks if the list is null. If the list is null, the function returns null. If the list is not null, the function uses the List.Generate function to loop through the list and find the position of the item.

Here is the M code behind the List.PositionOf function:

```
(list as list, item as any) =>
  if list = null then null
  else List.Generate(
    () => [i = 0, found = false],
    each [i] <= List.Count(list) - 1 and not [found],
    each [i = [i] + 1, found = list{i} = item],
    each [i]
  )
```

The List.Generate function is used to generate a list of values based on a set of rules. In this case, the function generates a list of values

List.PositionOfAny

Syntax and Arguments

The syntax for List.PositionOfAny is as follows:

```
List.PositionOfAny(list as list, values as list, optional occurrence as nullable number) as list
```

The function takes in three arguments:

1. list (required): the list of text values to search through
2. values (required): the list of characters or strings to search for
3. occurrence (optional): the occurrence of the value to search for (default is 1)

Examples

Let's take a look at some examples of List.PositionOfAny in action.

Example 1: Finding the Position of a Single Character

Suppose we have a dataset that contains a column of email addresses. We want to create a new column that contains the position of the "@" symbol within each email address.

We can use List.PositionOfAny to do this as follows:

```
let
  Source = Excel.CurrentWorkbook(){[Name="EmailAddresses"]}[Content],
  PositionOfAtSymbol = Table.AddColumn(Source, "PositionOfAtSymbol", each List.PositionOfAny({[EmailAddress]}, {"@"}))
in
  PositionOfAtSymbol
```

In this example, we are creating a new column called "PositionOfAtSymbol" that uses List.PositionOfAny to find the position of the "@"

List.Positions

In this article, we will take a closer look at the M code behind the List.Positions function and how it can be used to simplify your data analysis.

What is the List.Positions function?

The List.Positions function is a built-in function in Power Query that allows you to find all the occurrences of a specific value in a list. The function takes two arguments: the list to search and the value to find.

The List.Positions function returns a list of positions where the value is found in the list. If the value is not found in the list, the function returns an empty list.

How to use the List.Positions function

Using the List.Positions function is straightforward. Let's say you have a list of numbers and you want to find all the positions where the number 5 appears. Here's how you can use the List.Positions function to achieve that:

```
let
  numbers = {1, 3, 5, 2, 5, 4, 5},
  positions = List.Positions(numbers, 5)
in
  positions
```

This will return the following list of positions:

```
{2, 5, 7}
```

The M code behind the List.Positions function

The M code behind the List.Positions function is quite simple. Here's the code:

List.Product

Understanding List.Product Function

Before we dive into the M code, let’s understand the List.Product function in Power Query. The List.Product function takes two or more lists as inputs. It then multiplies the items in each of the lists and returns the product of all the items.

For example, let’s say we have two lists: “ListA” with 2, 4, and 6 as its items, and “ListB” with 3, 5, and 7 as its items. The List.Product function will multiply 2 x 3, 4 x 5, and 6 x 7, and then return the product of these three results: 2 x 3 x 4 x 5 x 6 x 7 = 2520.

The M Code Behind List.Product Function

Now that we understand what the List.Product function does, let’s take a closer look at the M code behind this function. The M code behind the List.Product function is:

```
(List1 as list, List2 as list, optional Lists as list) =>
List.Accumulate(
  List.Combine({List1, List2} & Lists),
  1,
  (state, current) => state  current
)
```

Let's break down this code to understand how it works.

Inputs

The List.Product function takes two or more lists as inputs. These inputs are represented by the parameters "List1", "List2", and "optional Lists". The "optional Lists" parameter is optional, which means that we can pass any number of additional lists to the function.

List.Combine Function

The List.Combine function is used to combine all the lists passed to the function. This function takes a list of lists as its input and returns a single list that contains all the items in the input lists. In the M code for the List.Product function, the List.Combine function is used to combine the two input lists ("List1" and "List2") and any additional lists passed to the function ("Lists").

List.Random

In this article, we will dive into the M code behind the List.Random function in Power Query. We will explore how this function works and how it can be used to randomly select items from a list.

What is the List.Random function?

The List.Random function is a built-in function in Power Query that is used to randomly select one or more items from a list. The function takes two arguments: the list to select from and the number of items to select.

The basic syntax for the List.Random function is as follows:

```
List.Random(list as list, count as number) as list
```

Here, `list` is the list of items to select from, and `count` is the number of items to select. The function returns a list of randomly selected items from the input list.

How does the List.Random function work?

The List.Random function works by generating a random number between 0 and the length of the input list. It then selects the item at the index corresponding to the random number, and adds it to the output list. This process is repeated for the specified number of items, until the output list contains the desired number of randomly selected items.

Here is an example of how the List.Random function works:

```
let
  myList = {1, 2, 3, 4, 5},
  selectedItems = List.Random(myList, 3)
in
  selectedItems
```

List.Range

What is the List.Range Function?

The List.Range function is used to return a specified range of elements from a list. The function takes three arguments:

– list: The list from which to extract the range of elements.

– offset: The position of the first element to return from the list.

– count: The number of elements to return from the list.

The function returns a list that contains the specified range of elements.

Understanding the M Code Behind the List.Range Function

To better understand how the List.Range function works, let's take a look at its M code:

```
(List as list, offset as number, count as number) as list =>
  let
    start = List.PositionOf(List.Skip(List.Range(List, offset), offset – 1), List.FirstN({1}, 1){0} + count – 1),
    end = List.PositionOf(List.Range(List.Skip(List, offset), start – offset + 1), List.FirstN({1}, 1){0} + count),
    result = List.Range(List.Skip(List, offset), start – offset + 1 + end – start)
  in
    result
```

The code may seem complex, but let's break it down.

The first line defines the function's arguments and return type. The function takes a list, an offset, and a count, and returns a list.

The second line creates a variable called "start." This variable is used to determine the starting position of the range of elements to return from the list. To do this, the List.Range function is called on the list, starting at the specified offset. The List.Skip function is used to skip the first elements of the list that are not included in the range. The List.PositionOf function is then used to find the position of the last element in the range.

The third line creates a variable called "end." This variable is used to determine the ending position of the range of elements to return from the list. To do this, the List.Range function is called on the list again, starting at the specified offset. The List.Skip function is used to

List.RemoveFirstN

Understanding the List.RemoveFirstN Function

The List.RemoveFirstN function is a built-in function in Power Query that removes the first n elements from a list. The syntax for the List.RemoveFirstN function is:

```
List.RemoveFirstN(list as list, count as number) as list
```

Here, 'list' is the list from which the first n elements need to be removed, and 'count' is the number of elements to be removed. The function returns the remaining elements as a list.

For example, consider the following list:

```
list = {1, 2, 3, 4, 5}
```

If we want to remove the first 2 elements from this list, we can use the List.RemoveFirstN function as follows:

```
List.RemoveFirstN(list, 2)
```

This will return the following list:

```
{3, 4, 5}
```

List.RemoveItems

In this article, we'll dive deep into the M code behind the List.RemoveItems function, exploring its syntax, arguments, and practical applications.

Syntax and Arguments

The syntax for the List.RemoveItems function is as follows:

```
List.RemoveItems(list as list, values as list, optional equationCriteria as nullable function) as list
```

The function takes in three arguments:

- `list` (required): The list from which to remove items.
- `values` (required): The list of items to remove from the `list` argument.
- `equationCriteria` (optional): A function that determines how to compare the items in the `list` and `values` arguments.

Practical Applications

The List.RemoveItems function can be used to remove specific items from a list. For example, suppose you have a list of sales data that includes sales from different regions and countries. You might want to remove all the sales data from a specific region or country to focus on other data. In this case, you can use the List.RemoveItems function to remove the unwanted data from the list.

Here's an example of how to use the List.RemoveItems function to remove all sales data from the United States:

```
let
  salesData = {{“United States”, 1000}, {“United Kingdom”, 1500}, {“Canada”, 800}},
  removeCountries = {“United States”},
  filteredSales = List.RemoveItems(salesData, removeCountries)
in
  filteredSales
```

List.RemoveLastN

In this article, we will explore the M code behind the List.RemoveLastN function and how it can be used to manipulate data in Power Query.

Understanding List.RemoveLastN

The List.RemoveLastN function is used to remove the last n items from a list. The function takes two arguments: the list and the number of items to remove. The syntax of the function is as follows:

```
List.RemoveLastN(list as list, count as number) as list
```

The function returns a list that contains all the items of the input list except the last n items. If the count argument is greater than or equal to the number of items in the list, the function returns an empty list.

The M Code Behind List.RemoveLastN

The M code behind the List.RemoveLastN function is relatively simple. It consists of a single expression that uses the List.Take function to extract the first n items from the list and the List.Count function to calculate the number of items in the list.

```
list => List.Take(list, List.Count(list) – count)
```

The code above takes the list as input and returns a new list that contains all the items of the input list except the last n items. The List.Count function is used to determine the number of items in the list, and this value is subtracted from the count argument to determine the number of items to extract using the List.Take function.

Using List.RemoveLastN in Power Query

The List.RemoveLastN function can be used in a wide range of scenarios in Power Query. Here are a few examples:

Example 1: Removing the last row from a table

Suppose you have a table that contains data for multiple years, and you want to remove the last row, which contains data for the current

List.RemoveMatchingItems

What is List.RemoveMatchingItems?

List.RemoveMatchingItems is a Power Query M function that allows you to remove all instances of a particular value from a list. This can be useful when you're working with large data sets and need to quickly eliminate certain values. The syntax for List.RemoveMatchingItems is as follows:

List.RemoveMatchingItems(list as list, match as any, optional equationCriteria as function) as list

The first argument, list, is the list that you want to remove items from. The second argument, match, is the value that you want to remove. Finally, the optional third argument, equationCriteria, is a function that can be used to specify a custom comparison function.

How does List.RemoveMatchingItems work?

The List.RemoveMatchingItems function works by iterating over each item in the list and checking whether it matches the specified value. If a match is found, that item is removed from the list. This process continues until all instances of the specified value have been removed.

Examples of List.RemoveMatchingItems in action

Let's take a look at some examples of List.RemoveMatchingItems in action. In these examples, we'll assume that we have a list of values that contains a mix of numbers and text.

Example 1: Removing a single value from a list

Suppose we have the following list:

{1, “apple”, 3, “banana”, 5, “orange”, 7}

If we want to remove all instances of the value "apple" from this list, we can use the following code:

List.RemoveNulls

Understanding the M Code Behind List.RemoveNulls

Before we dive into the M code behind `List.RemoveNulls`, let's first take a look at what this function does. The `List.RemoveNulls` function takes a list as its input and returns a new list with all null values removed. Here is the syntax for the `List.RemoveNulls` function:

```
List.RemoveNulls(list as list) as list
```

Now let's take a closer look at the M code behind this function. The M code for `List.RemoveNulls` is fairly straightforward. Here is the code:

```
(list) => List.Select(list, each _ <> null)
```

Let's break down this code. The `List.Select` function is used to filter out null values from the input list. The second argument of the `List.Select` function is a logical expression that determines whether each item in the list should be included in the output list. In this case, the logical expression is `each _ <> null`, which means that any item in the list that is not null will be included in the output list.

Use Cases for List.RemoveNulls

Now that we understand the M code behind `List.RemoveNulls`, let's explore some use cases for this function.

Cleaning Up Data

One of the most common use cases for `List.RemoveNulls` is to clean up data. When working with large datasets, it is not uncommon to encounter null values in your data. These null values can cause issues when analyzing or visualizing your data. By using `List.RemoveNulls`, you can quickly and easily remove null values from your data, making it easier to work with.

Creating Custom Functions

Another use case for `List.RemoveNulls` is to create custom functions that perform more complex data cleaning tasks. For example,

List.RemoveRange

Syntax of List.RemoveRange Function

The syntax of the List.RemoveRange function is as follows:

```
List.RemoveRange(list as list, offset as number, count as number) as list
```

The List.RemoveRange function takes three arguments:

- list: A list from which items need to be removed.
- offset: A number that specifies the starting index of the range.
- count: A number that specifies the number of items to remove.

How List.RemoveRange Function Works

The List.RemoveRange function works by creating a new list that excludes the range of items specified by the offset and count parameters. The function does not modify the original list and returns a new list containing the remaining items.

The following example illustrates how the List.RemoveRange function works:

```
let
  myList = {1, 2, 3, 4, 5, 6, 7},
  myRange = List.RemoveRange(myList, 2, 3)
in
  myRange
```

In this example, we have a list of numbers from 1 to 7. We want to remove the range of items starting from index 2 and including the next 3 items. The List.RemoveRange function is used to create a new list that excludes the specified range. The resulting list contains the remaining items, which are 1, 2, and 6.

List.Repeat

Understanding the Function Syntax

Before we dive into the M code, it’s important to understand the syntax of the List.Repeat function. The function takes two arguments: the value that you want to repeat, and the number of times that you want to repeat it. Here’s the syntax:

List.Repeat(value as any, count as number) as list

The function returns a list of the repeated values. For example, if you want to repeat the value "hello" three times, the function call would look like this:

List.Repeat(“hello”, 3)

And the output would be:

{“hello”, “hello”, “hello”}

Breaking Down the M Code

Now that we understand the basic syntax of the function, let's take a closer look at the M code behind it. Here's what the code looks like:

```
let
  repeat = List.Generate(
List.ReplaceMatchingItems
```

One of the most powerful functions in the M language is List.ReplaceMatchingItems. This function allows users to replace all occurrences of a particular value in a list with a new value. In this article, we will take a closer look at the M code behind this function and explore some of its use cases.

Understanding List.ReplaceMatchingItems

Before we dive into the M code behind List.ReplaceMatchingItems, let's first take a moment to understand what the function does. List.ReplaceMatchingItems is used to replace all occurrences of a particular value in a list with a new value. The function takes three arguments:

1. The list to be modified.
2. The value to be replaced.
3. The new value that will replace all occurrences of the original value.

Here's an example of how the function can be used:

```
let
  myList = {1, 2, 3, 4, 5, 3, 6, 7},
  replacedList = List.ReplaceMatchingItems(myList, 3, 9)
in
  replacedList
```

This code will replace all occurrences of the value 3 in the list {1, 2, 3, 4, 5, 3, 6, 7} with the value 9. The resulting list will be {1, 2, 9, 4, 5, 9, 6, 7}.

The M Code Behind List.ReplaceMatchingItems

Now that we understand what List.ReplaceMatchingItems does, let's take a closer look at the M code behind the function. Here's the M code for List.ReplaceMatchingItems:

List.ReplaceRange

Understanding the List.ReplaceRange Function

The List.ReplaceRange function is used to replace a range of values within a list with a new set of values. The syntax for the List.ReplaceRange function is as follows:

```
List.ReplaceRange(list as list, offset as number, count as number, newItems as any) as list
```

Here, the `list` parameter is the input list that we want to modify. The `offset` parameter is the position within the list where we want to start replacing values. The `count` parameter is the number of values that we want to replace. Finally, the `newItems` parameter is the new set of values that we want to replace the old values with.

Using the List.ReplaceRange Function

Now that we understand the syntax of the List.ReplaceRange function, let's look at an example of how it can be used to modify data within Power Query.

Suppose we have a table containing sales data for a company. The table contains columns for the month, product, and sales amount. We want to replace the sales amount for the month of January with a new value of 1000. We can use the List.ReplaceRange function to achieve this.

First, we need to select the sales amount column and convert it to a list. We can do this using the following M code:

```
= Table.Column([SalesData], “SalesAmount”)
```

Here, `[SalesData]` is the name of the table that contains our sales data, and `"SalesAmount"` is the name of the column that contains the sales amount data.

Next, we can use the List.ReplaceRange function to replace the sales amount for the month of January. We can do this using the following M code:

List.ReplaceValue

Understanding the List.ReplaceValue Function

Before we dive into the code behind the List.ReplaceValue M function, let’s first understand what the function is all about. List.ReplaceValue is a function that enables you to replace values in a list with new values. The syntax for the function is as follows:

```
List.ReplaceValue(list as list, old_value as any, new_value as any, optional occurrence as nullable number) as list
```

The List.ReplaceValue function takes four arguments. The first argument is the list that you want to modify. The second argument is the old value that you want to replace. The third argument is the new value that you want to replace the old value with. The fourth argument is optional, and it specifies the number of occurrences of the old value that you want to replace. If you omit the fourth argument, all occurrences of the old value in the list will be replaced.

The M Code Behind the List.ReplaceValue Function

Now that we have a basic understanding of the List.ReplaceValue function, let's take a closer look at the M code that powers the function. The M code for List.ReplaceValue is as follows:

```
(list as list, old_value as any, new_value as any, optional occurrence as nullable number) =>
let
  replaceValue = (value) => if value = old_value then new_value else value,
  replaceAll = List.Transform(list, replaceValue),
  replaceOccurrences = if occurrence <> null then List.PositionalReplace(replaceAll, old_value, new_value, occurrence) else replaceAll
in
  replaceOccurrences
```

Let's break down the code step-by-step to understand what's happening. The first line of the code defines the function and its

List.Reverse

What is the List.Reverse function in Power Query?

The List.Reverse function is used to reverse the order of items in a list. It takes a list as its input and returns a new list with the items in reverse order. This function is commonly used to sort data in descending order or to reverse the order of columns in a table.

The M code behind the List.Reverse function

The M code behind the List.Reverse function is relatively simple. The function takes a list as its input and uses the List.Buffer function to create a new list with the same items. The List.Buffer function is used to improve performance by storing the list in memory as a buffer. After creating the new list, the M code uses a for loop to iterate through each item in reverse order. The loop starts with the last item in the list and moves backwards to the first item. The loop uses the List.InsertRange function to insert each item into the new list in reverse order.

The M code for the List.Reverse function looks like this:

```
(List as list) =>
let
  Source = List.Buffer(List),
  Result = List.Generate(
    () => List.Count(Source) – 1,
    each _ >= 0,
    each _ – 1,
    each Source{_}
  )
in
  Result
```

The code creates a new list called "Source" using the List.Buffer function. The "Result" variable uses the List.Generate function to loop through each item in the source list in reverse order. The List.Count function is used to determine the number of items in the list, and List.Select

Overview of the List.Select Function

The List.Select function is used to filter a list based on a condition. It takes two arguments: the first argument is the list to filter, and the second argument is the condition to filter on. The condition is specified using a function that takes a single argument, which is the item from the list being evaluated.

For example, consider the following list:

{“apple”, “banana”, “cherry”, “date”, “elderberry”}

Suppose we want to filter this list to include only items that start with the letter "b". We can do this using the List.Select function as follows:

List.Select({“apple”, “banana”, “cherry”, “date”, “elderberry”}, each Text.StartsWith(_, “b”))

The second argument to List.Select is a function that takes a single argument (represented by the underscore) and returns a Boolean value indicating whether the item should be included in the filtered list. In this case, we use the Text.StartsWith function to test whether the item starts with the letter "b".

The resulting list will be:

{“banana”}

The M Code Behind List.Select

List.Single

Overview of the List.Single Function

The List.Single function is used to extract a single value from a list of values. It takes a list as input and returns the first item in the list. If the list contains more than one item, it throws an error. Here is the syntax of the List.Single function:

```
List.Single(list as list) as any
```

The List.Single function takes a single argument, which is the list of values. The function returns the first item in the list as an any data type.

Understanding the M Code Behind List.Single

The M code behind the List.Single function is relatively simple. Here is the code:

```
list{0}
```

The code above extracts the first item in the list by using the indexing operator { } and specifying the index value 0. In M, arrays and lists are zero-indexed, meaning that the first item in the list has an index of 0, the second item has an index of 1, and so on.

The List.Single function is a wrapper function that encapsulates this M code and provides some additional functionality, such as error handling if the list contains more than one item.

Using the List.Single Function

Let's see an example of how to use the List.Single function in Power Query. Suppose we have a table that contains a column called "Fruit" that contains a list of fruit names. We want to extract the first fruit name from each row of the table. Here is how we can do it using the List.Single function:

1. Load the table into Power Query.
2. Select the "Fruit" column.

List.SingleOrDefault

What is the List.SingleOrDefault function?

The List.SingleOrDefault function is used to retrieve the single element from a list that matches a specific condition. If there is more than one element that matches the condition, an error is returned. If there are no elements that match the condition, a null value is returned.

The syntax for the List.SingleOrDefault function is as follows:

List.SingleOrDefault(list as list, optional condition as function)

- `list`: The list of values to search through.
- `condition`: An optional function that determines which element to return. If no function is provided, the function returns the first element in the list.

How does the List.SingleOrDefault function work?

The List.SingleOrDefault function works by iterating through each element in the list and checking if it matches the specified condition. If there is only one element that matches the condition, that element is returned. If there are no elements that match the condition, the function returns a null value. If there are multiple elements that match the condition, an error is returned.

Let's look at an example to illustrate how the List.SingleOrDefault function works. Suppose we have a list of sales data with the following columns: Product, Salesperson, and SalesAmount. We want to retrieve the sales amount for a specific product and salesperson. We can use the List.SingleOrDefault function to achieve this as follows:

```
let
  Source = #table({“Product”, “Salesperson”, “SalesAmount”}, {
    {“Product A”, “John”, 100},
    {“Product B”, “John”, 200},
    {“Product A”, “Sally”, 150},
```

List.Skip

What is List.Skip?

The List.Skip function is used to skip a specified number of elements from the beginning of a list and return the remaining elements. It takes two arguments: the first argument is the list, and the second argument is the number of elements to skip. The syntax for the List.Skip function is:

```
List.Skip(list as list, count as number) as list
```

Here, "list" is the list to be skipped, and "count" is the number of elements to skip.

How to Use List.Skip?

To use the List.Skip function in Power Query, you need to follow these steps:

1. Open Microsoft Excel or Microsoft Power BI.
2. Click on the "Data" tab.
3. Click on "From Table/Range" to import your data.
4. Click on "Transform Data" to open the Power Query Editor.
5. In the "Power Query Editor," select the column you want to skip elements from.
6. Click on the "Add Column" tab.
7. Click on "Custom Column" to open the "Add Custom Column" dialog box.
8. In the "Add Custom Column" dialog box, enter the formula for the List.Skip function.
9. Click "OK" to add the new column.

Here is an example of how to use the List.Skip function to skip the first two elements of a list:

```
let
  myList = {1, 2, 3, 4, 5},
  skippedList = List.Skip(myList, 2)
```

List.Sort

Understanding List.Sort Function

List.Sort is a simple function that can be used to sort lists of values in ascending or descending order. The syntax of the List.Sort function is as follows:

```
List.Sort(list as list, optional comparisonCriteria as any, optional sortOrder as any) as list
```

Here, the `list` represents the list of values that we want to sort. The `comparisonCriteria` is an optional parameter that we can use to specify the type of comparison to be used while sorting the list. The `sortOrder` is another optional parameter that can be used to specify whether the list should be sorted in ascending or descending order.

Breaking Down the M Code Behind List.Sort

Now let's take a closer look at the M code behind List.Sort. The following is the M code for the List.Sort function:

```
let
  Source = List.Sort(list, comparisonCriteria, sortOrder)
in
  Source
```

This M code is fairly straightforward. It defines a variable called `Source`, which uses the List.Sort function to sort the list. The List.Sort function takes the three parameters that we discussed earlier: `list`, `comparisonCriteria`, and `sortOrder`.

However, the real magic happens inside the List.Sort function. When we call the List.Sort function, it actually calls a hidden function called the comparer function, which is responsible for comparing the values in the list and sorting them.

The Comparer Function

The comparer function is an internal function that is used by List.Sort to compare the values in the list and sort them. The comparer

List.Split

Understanding the List.Split Function

The List.Split function has two parameters: the list to be split and the delimiter (or splitting function). The function returns a list of lists, where each sublist contains the elements between the delimiters.

Here is the basic syntax for the List.Split function:

List.Split(list as list, delimiter as any) as list

The `list` parameter is the input list to be split, and the `delimiter` parameter is either a value or a function that specifies the splitting criteria.

If `delimiter` is a value, it is used as the delimiter to split the list. For example, if we wanted to split a list of names by the comma delimiter, we would use the following code:

List.Split({“John, Smith”, “Jane, Doe”, “Bob, Johnson”}, “,”)

This would return the following list of lists:

```
{
  {“John”, ” Smith”},
  {“Jane”, ” Doe”},
  {“Bob”, ” Johnson”}
}
```

List.StandardDeviation

What is Standard Deviation?

Before we dive into the M code behind the List.StandardDeviation function, it is important to understand what standard deviation is. Standard deviation is a measure of the variability or dispersion of a set of data values. It measures how spread out the values are from the mean or average value. The higher the standard deviation, the more spread out the values are, and the lower the standard deviation, the more tightly clustered the values are.

The List.StandardDeviation Function

The List.StandardDeviation function is used to calculate the standard deviation of a list of numbers. The syntax of the function is as follows:

```
List.StandardDeviation(list as list, optional options as nullable record) as nullable number
```

The List.StandardDeviation function takes two arguments. The first argument is the list of numbers for which we want to calculate the standard deviation. The second argument is an optional record that contains options for calculating the standard deviation. The function returns the standard deviation of the list of numbers as a nullable number.

The M Code Behind the List.StandardDeviation Function

The M code behind the List.StandardDeviation function is a bit complex, but it can be broken down into several steps. Let's take a look at the M code behind the function:

```
let
  Source = List.Average(list),
  SumOfSquares = List.Sum(List.Transform(list, each _ _)),
  Count = List.Count(list),
  StandardDeviation = if Count = 0 then null else if Count = 1 then 0 else
    let
List.Sum
```

In this article, we will be taking a closer look at the M code behind the List.Sum function and how it works. We will also explore some practical examples of how you can use this function in your own data analysis projects.

Understanding the List.Sum Function

The List.Sum function is a part of the M language, which is the programming language used in Power Query. This function takes a list as its input and returns the sum of all the values in that list. The syntax for the List.Sum function is as follows:

```
List.Sum(list as list) as number
```

The argument for this function is a list, which can be either a column or a list of values. The function returns a number that represents the sum of all the values in the list.

The M Code Behind List.Sum

The M code behind the List.Sum function is relatively simple. When you use the List.Sum function in Power Query, the M code that is generated looks something like this:

```
= List.Sum({1, 2, 3, 4})
```

This code creates a list of values (1, 2, 3, and 4) and then passes that list to the List.Sum function. The function then adds up the values in the list and returns a result of 10.

Of course, you can also use the List.Sum function with a column of data. In this case, the M code would look something like this:

```
= Table.AddColumn(#"PreviousStep", "ColumnSum", each List.Sum([Column]))
```

List.Times

What is the List.Times function?

The List.Times function is used to repeat a value or a list of values a specified number of times. It takes two arguments: the value or list to repeat and the number of times to repeat it. For example, List.Times(5,3) will return the list {5,5,5}. List.Times can also be used to repeat a list of values. For example, List.Times({1,2,3},2) will return the list {1,2,3,1,2,3}.

The M code behind the List.Times function

The M code behind the List.Times function is relatively simple. It uses a for loop to repeat the specified value or values the specified number of times. Here is the M code for the List.Times function:

```
(List as any, Count as number) =>
let
  Source = List.Generate(() => 0, each _ < Count, each _ + 1, each List)
in
  Source{Count}
```

Let's break down this code line by line.

Line 1

```
(List as any, Count as number) =>
```

This line defines the function and its arguments. The List argument can be of any data type, while the Count argument must be a number.

Line 2

List.Transform

Understanding the List.Transform Function

The List.Transform function takes two arguments: a list and a function. The function applies a transformation to each element of the list and returns a new list with the transformed elements. The syntax for the List.Transform function is as follows:

```
List.Transform(list as list, transform as function) as list
```

Here, `list` is the list to be transformed, and `transform` is the function that will be applied to each element of the list. The `as list` at the end of the function definition is used to specify the output type of the function, which is also a list.

Examples of Using the List.Transform Function

Let's take a look at some examples of using the List.Transform function in Power Query.

Example 1: Converting All Text to Uppercase

Suppose we have a list of names that we want to convert to uppercase. We can use the List.Transform function to apply the `Text.Upper` function to each element of the list as follows:

```
let
  names = {“john”, “jane”, “jim”},
  uppercaseNames = List.Transform(names, each Text.Upper(_))
in
  uppercaseNames
```

In this example, we define a list called `names` that contains three elements: john, jane, and jim. We then use the List.Transform function to apply the `Text.Upper` function to each element of the `names` list, which converts all text to uppercase. The resulting list is stored in the `uppercaseNames` variable.

List.TransformMany

Understanding the List.TransformMany Function

The List.TransformMany function is a powerful function in Power Query that can be used to transform data in a variety of ways. At a high level, the function takes a list of values and applies a transformation function to each item in the list. The transformation function can return a list of values, and the List.TransformMany function will combine all of the lists into a single list of transformed values.

Here is a basic example of how the List.TransformMany function works:

```
let
  inputList = {1,2,3},
  outputList = List.TransformMany(inputList, each { _ 2, _ 3 })
in
  outputList
```

In this example, we start with a list of integers (1, 2, and 3). We then use the List.TransformMany function to apply a transformation function to each item in the list. The transformation function is defined using the "each" keyword, which is a shorthand way of defining a function that takes a single argument. In this case, the argument is represented by the underscore (_) character, which is a placeholder for the current item in the list being transformed.

The transformation function itself returns a list of two values: the current item in the list multiplied by 2, and the current item in the list multiplied by 3. The List.TransformMany function then combines all of the lists together into a single list of transformed values, which in this case is (2, 3, 4, 6, 6, 9).

Advanced Examples of List.TransformMany

While the basic example above demonstrates the basic functionality of the List.TransformMany function, there are many more advanced ways that it can be used in Power Query. Here are a few examples:

Applying Multiple Transformation Functions

In some cases, you may want to apply multiple transformation functions to each item in a list. You can achieve this by nesting multiple List.TransformMany functions together. Here's an example:

List.Union

Syntax of List.Union

The List.Union function takes one or more lists as input and combines them into a single list. The syntax of List.Union function is as follows:

List.Union(list1 as list, list2 as list, ..., optional comparer as any) as list

The function takes one or more lists as input, separated by commas. You can combine two or more lists by specifying their names. The optional comparer parameter is a function that compares elements in the list. If you omit the comparer parameter, List.Union uses the default equality comparer to compare elements in the list.

Usage of List.Union

You can use List.Union to combine two or more lists into a single list. The function returns a new list that contains all the distinct elements from the input lists. For example, suppose you have two lists, A and B, with the following elements:

A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

You can use the List.Union function to combine the two lists and get a new list, C, with all the distinct elements from A and B:

C = List.Union(A, B)

The resulting list, C, will have the following elements:

List.Zip

What is List.Zip Function?

List.Zip is a M function in Power Query that takes two lists as input and zips them together into a single list of records. The two lists must have the same number of elements. The resulting list of records will have a record for each pair of elements in the input lists. The first element of the first list will be paired with the first element of the second list, the second element of the first list will be paired with the second element of the second list, and so on.

How to Use List.Zip Function?

To use the List.Zip function, you need to pass two lists as arguments. Here is an example:

```
let
   List1 = {1, 2, 3},
   List2 = {“A”, “B”, “C”},
   ZipList = List.Zip({List1, List2})
in
   ZipList
```

In this example, we have two lists: List1 with elements 1, 2, and 3, and List2 with elements "A", "B", and "C". We then call the List.Zip function and pass it the two lists as arguments using the curly braces syntax { }. The result of this function call is a single list of records:

```
{[Column1=1,Column2=”A”],[Column1=2,Column2=”B”],[Column1=3,Column2=”C”]}
```

As you can see, the resulting list of records has three elements, one for each pair of elements in the input lists.

The M Code Behind List.Zip Function

The M code behind the List.Zip function is relatively simple. Here is what it looks like:

Logical.From

What is Logical.From?

The Logical.From function is a relatively simple M function that converts a boolean value to a text value. For example, if the input value is true, the output value will be “true”. Similarly, if the input value is false, the output value will be “false”.

The M Code Behind Logical.From

The M code behind Logical.From is relatively simple. Here’s the code in its entirety:

```
let
   Logical.From = (value as logical) as text =>
      if value then “true” else “false”
in
   Logical.From
```

The code defines a function called Logical.From that takes a single input parameter, "value". The function then uses an "if" statement to check whether the input value is true or false. If the input value is true, the function returns the text value "true". If the input value is false, the function returns the text value "false".

How to Use Logical.From

Using Logical.From is relatively simple. To use the function in your Power Query code, you simply need to call the function and pass in a boolean value. Here's an example:

```
let
   Source = Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText(“i45WMjIwMjEA”, BinaryEncoding.Base64)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Column1 = _t])),
   #”Changed Type” = Table.TransformColumnTypes(Source,{{“Column1”, type logical}}),
   #”Added Custom” = Table.AddColumn(#”Changed Type”, “Logical Value”, each Logical.From([Column1]))
```

Logical.FromText

The M code behind the Logical.FromText function is relatively simple and can be broken down into a few key components. In this article, we will take a look at the M code behind this function and explore how it works.

Understanding Logical.FromText

Before we dive into the M code behind the Logical.FromText function, it is important to first understand what this function does. As mentioned earlier, this function allows users to convert text values into logical values. Specifically, it converts text values that represent boolean or logical values into their corresponding true/false values.

For example, suppose we have a column in our data that contains text values “Yes” and “No”. We can use the Logical.FromText function to convert these text values into their corresponding true/false values, which can then be used for analysis.

The M Code Behind Logical.FromText

The M code behind the Logical.FromText function is relatively simple and can be broken down into a few key components. The code is as follows:

```
(Text as text) as logical =>
  if Text = “true” or Text = “1” then true
  else if Text = “false” or Text = “0” then false
  else null
```

Let's take a closer look at each of these components.

Text as text

The first part of the code, `Text as text`, specifies that the input to the function should be treated as text.

as logical

The second part of the code, `as logical`, specifies that the output of the function should be a logical value.

if/else statements

The bulk of the code consists of a series of if/else statements that determine the output of the function based on the input.

The first if statement checks if the input text is equal to "true" or "1". If it is, the function returns true.

Logical.ToText

In this article, we will delve into the M code behind the Logical.ToText function and explore its features and capabilities.

Understanding the Logical.ToText Function

The Logical.ToText function is used to convert a logical value to text. The function takes two arguments: the value to be converted and an optional parameter that specifies the format of the output.

The syntax for the Logical.ToText function is as follows:

```
Logical.ToText(logical_value as any, optional format_text as nullable text)
```

The first argument, logical_value, is the value to be converted. This argument can be a logical value, such as true or false, or a column that contains logical values.

The second argument, format_text, is an optional parameter that specifies the format of the output. If this argument is not specified, the function returns the value "true" or "false" as text.

The M Code Behind the Logical.ToText Function

The Logical.ToText function is implemented in the M language, which is the language used by Power Query to define transformations and calculations. The M code behind the Logical.ToText function is relatively simple and can be modified to create custom transformations.

The M code for the Logical.ToText function is as follows:

```
let
  logical_to_text = (value as any, optional format_text as nullable text) =>
    if value = true then
      if format_text = null then “true” else format_text
    else if value = false then
      if format_text = null then “false” else format_text
```

MissingField option

In this article, we will dive deep into the M code behind the Power Query M function MissingField option and understand how it works in real-world scenarios.

Introduction to MissingField option in Power Query

The MissingField option is a useful feature in Power Query that allows users to handle missing values in their data. When working with data, it is common to come across missing values, either due to human error, system limitations, or other factors. These missing values can cause inconsistencies and inaccuracies in data analysis, and hence, it is important to handle them effectively.

The MissingField option in Power Query allows users to specify how they want to handle the missing values. The option can be used to replace the missing values with a default value, remove the rows with missing values, or apply a custom transformation to the missing values.

How to use MissingField option in Power Query

To use the MissingField option in Power Query, users need to follow the below steps:

1. Load the data into Power Query
2. Select the column(s) with missing values
3. Click on the Transform tab
4. Click on the MissingField option under the Any Column section
5. Select the appropriate option from the drop-down menu

Users can select one of the following options:

– Replace with default value: Users can replace the missing values with a default value of their choice. For example, they can replace the missing values with 0, NA, or any other value.

– Remove rows: Users can remove the rows with missing values from their data. This option is useful when the missing values are few and do not impact the overall analysis significantly.

– Custom transformation: Users can apply a custom transformation to the missing values. For example, they can replace the missing values with the average of the non-missing values in the same column.

The M code behind the MissingField option

The MissingField option in Power Query is powered by the M language, which is a functional programming language used in Power Query. The M language provides users with a wide range of functions and options to manipulate and transform data.

The M code behind the MissingField option is based on a simple logic that checks for the missing values in the selected column(s) and

MySQL.Database

In this article, we will explore the M code behind the Power Query M function MySQL.Database. We will begin by discussing what MySQL is and why it is popular among developers and data analysts. Then, we will delve into the M code behind the MySQL.Database function and provide examples of how it can be used to extract and transform data from MySQL databases.

What is MySQL and why is it popular?

MySQL is an open-source relational database management system (RDBMS) that is widely used by developers and data analysts. It was created in 1995 by Swedish developers Michael Widenius and David Axmark and is now owned by Oracle Corporation.

MySQL is popular for several reasons. Firstly, it is free and open-source, which means that anyone can use it without paying licensing fees. Secondly, it is cross-platform, which means that it can run on a variety of operating systems, including Windows, Linux, and MacOS. Thirdly, it is highly scalable and can handle large volumes of data.

MySQL is also known for its ease of use and flexibility. It supports a wide range of data types, including integers, floats, strings, and dates, and can be easily integrated with other tools and programming languages.

The M code behind the MySQL.Database function

The MySQL.Database function is a Power Query M function that is used to establish a connection to a MySQL database and retrieve data from it. The function takes several parameters, including the server name, database name, username, and password.

Here is an example of the M code behind the MySQL.Database function:

```
let
  Source = MySQL.Database(“localhost”, “mydatabase”, [Username=”myusername”, Password=”mypassword”])
in
  Source
```

In this example, we are connecting to a MySQL database hosted on the local machine, with the database name "mydatabase" and the username and password "myusername" and "mypassword", respectively.

Once the connection has been established, we can use the Source variable to retrieve data from the database. For example, we can use the following code to retrieve all the rows from a table called "customers":

Name

Understanding the M Function

The M function is a functional language used in Power Query to create custom transformations. It is a case-sensitive language that is used to create expressions that manipulate data. The M function is used extensively in Power Query to create custom columns, merge queries, and perform complex transformations.

The M function is a combination of built-in functions, operators, and expressions. These functions can be used to create complex expressions that perform advanced transformations on data. The M function is a powerful feature of Power Query that allows you to create custom transformations that fit your specific needs.

The M Code Behind the Power Query M Function Name

The M code behind the Power Query M function name is a complex expression that is used to perform specific transformations on data. The M function is used to create this expression, which is then executed by Power Query to transform the data.

The M code behind the Power Query M function name is made up of a series of functions, operators, and expressions. These functions are used to manipulate data, filter data, and transform data. The M code can be complex and difficult to understand, but with practice, you can become proficient in using it.

Examples of M Functions

There are many M functions that can be used in Power Query to transform data. Here are some examples of common M functions:

– Text.Replace: This function is used to replace text in a string.

– Table.SelectColumns: This function is used to select specific columns from a table.

– List.Sort: This function is used to sort a list in ascending or descending order.

– Date.Year: This function is used to extract the year from a date.

These are just a few examples of the many M functions available in Power Query. Each function has its own specific purpose and can be combined with other functions to create complex expressions that perform specific transformations on data.

Best Practices for Using the M Function

When using the M function in Power Query, there are some best practices that you should follow to ensure that your transformations are efficient and effective:

1. Use appropriate data types: It is important to use the correct data types when working with data in Power Query. Using the wrong data type can result in errors or incorrect results.

2. Break down complex expressions: Complex expressions can be difficult to understand and debug. It is best to break down complex

Syntax of the Number.Abs Function

The syntax of the Number.Abs function is as follows:

```
Number.Abs(number as nullable number) as nullable number
```

The function takes a single argument, which is the number whose absolute value is to be calculated. The argument can be any nullable number, including integers, decimals, and fractions. The function returns the absolute value of the number as a nullable number.

How the Number.Abs Function Works

The Number.Abs function works by taking the input number and returning its absolute value. The absolute value of a number is its magnitude without considering its sign. For example, the absolute value of -5 is 5, and the absolute value of 5 is also 5.

The M code behind the Number.Abs function is quite simple. It uses a conditional statement to check whether the input number is positive or negative. If the number is positive, the function returns the same number. If the number is negative, the function returns the negative of the number, which is its absolute value.

Here is the M code behind the Number.Abs function:

```
(number as nullable number) =>
   if number >= 0 then number else -number
```

The code defines a lambda function that takes a single argument, which is the input number. The function then checks whether the input number is greater than or equal to zero. If the number is greater than or equal to zero, the function returns the same number. If the number is less than zero, the function returns the negative of the number, which is its absolute value.

Examples of Using the Number.Abs Function

Here are some examples of using the Number.Abs function in Power Query M:

Number.Acos

What is the Number.Acos Function?

The Number.Acos function is a Power Query M function that returns the arccosine of a given number. The arccosine of a number is the angle whose cosine is the given number. The Number.Acos function takes a single argument, which is the number whose arccosine is to be calculated. The function returns a decimal number, which represents the arccosine of the input number in radians.

The M Code Behind the Number.Acos Function

The M code behind the Number.Acos function is relatively simple. The following code snippet shows the M code for the Number.Acos function:

```
let
  Number.Acos = (number as number) as number =>
    if number >= -1 and number <= 1 then
      Number.Acos = Math.Acos(number)
    else
      error “Invalid input”
in
  Number.Acos
```

The code starts with the let keyword, which is used to define a variable. In this case, the variable is Number.Acos, which is the name of the function. The function takes a single argument, which is a number. The function returns a number, which is the arccosine of the input number.

The first line of the function checks whether the input number is between -1 and 1. If the input number is outside this range, the function returns an error message. Otherwise, the function calculates the arccosine of the input number using the built-in Math.Acos function.

How to Use the Number.Acos Function

Using the Number.Acos function in Power Query is very straightforward. To use the function, you need to follow these steps:

1. Open Power Query and create a new query.

Number.Asin

Understanding Number.Asin Function

The Number.Asin function in Power Query M returns the arcsine of a number in radians. The syntax for the function is as follows:

```
Number.Asin(number as any) as number
```

The `number` argument can be any valid numeric expression. The function returns the arcsine of the number in radians. If the number is outside the range of -1 to 1, the function returns an error.

The M Code Behind Number.Asin Function

The M code that powers the Number.Asin function is relatively simple. It takes the input number, converts it to a decimal value and then applies the Math.Asin function to it. Here is the M code that powers the Number.Asin function:

```
(number as any) =>
  if number >= -1 and number <= 1 then
    Math.Asin(number)
  else
    error “Invalid argument to Number.Asin”
```

The code starts by checking if the input number is within the valid range of -1 to 1 using the if statement. If the number is within the valid range, the code applies the Math.Asin function to it and returns the result. If the number is outside the valid range, the code returns an error message.

Using Number.Asin Function in Data Analysis

The Number.Asin function can be used in a variety of data analysis scenarios. For example, it can be used to calculate the sine of an angle given its opposite and hypotenuse lengths. Here is an example of how the Number.Asin function can be used in Power Query to

Number.Atan

Overview of the Atan Function

The atan function returns the arctangent of a given number. In mathematical terms, this function calculates the angle whose tangent is equal to the given number. The result is returned in radians.

The syntax for the atan function in Power Query M is as follows:

```
Number.Atan(number as nullable number) as nullable number
```

This function takes a single argument, which is the number whose arctangent is to be calculated. The argument must be a nullable number, which means that it can be either a valid number or a null value.

The M Code Behind Number.Atan

The M code behind the Number.Atan function is relatively simple. The function takes the given number argument, converts it to a decimal value, and then passes it to the .NET Framework's Math.Atan method to calculate the arctangent.

Here is the M code for the Number.Atan function:

```
let
  Number.Atan = (number as nullable number) as nullable number =>
    if number <> null then
      Math.Atan(Number.From(number))
    else
      null
in
  Number.Atan
```

Number.Atan2

What is the Number.Atan2 Function?

The Number.Atan2 function is a mathematical function that calculates the arctangent of the quotient of two numbers. It takes two arguments: x and y. The function returns the angle in radians between the positive x-axis and the point (x, y) on the Cartesian plane.

The Number.Atan2 function is commonly used in engineering and scientific applications to calculate the angle between two points. For example, it can be used to calculate the direction of a vector, or the angle between two lines.

The M Code Behind the Number.Atan2 Function

The M code behind the Number.Atan2 function is relatively simple. The function takes two arguments, x and y, and calculates the arctangent of the quotient of the two numbers using the Excel ATAN2 function.

Here is the M code for the Number.Atan2 function:

```
let
  Number.Atan2 = (x as number, y as number) as number =>
    Excel.WorkbookFunction.Atan2(x, y)
in
  Number.Atan2
```

The code defines a function called Number.Atan2 that takes two arguments, x and y. The function uses the Excel.WorkbookFunction.Atan2 function to calculate the arctangent of the quotient of x and y, and returns the result as a number.

Using the Number.Atan2 Function in Power Query

To use the Number.Atan2 function in Power Query, you first need to load the function into Power Query. This can be done by creating a new query and pasting the M code for the function into the Advanced Editor.

Once the function is loaded, you can use it in your queries like any other function. For example, here is a query that calculates the angle between two points:

Number.BitwiseAnd

What is Number.BitwiseAnd?

Before we dive into the M code behind Number.BitwiseAnd, let’s first understand what this function does. The Number.BitwiseAnd function is used to perform a bitwise AND operation on two numbers. In other words, it compares the binary representation of two numbers and only returns the bits that are set in both numbers.

For example, if we perform a bitwise AND operation on the numbers 5 and 3, we get the result 1. This is because the binary representation of 5 is 101 and the binary representation of 3 is 011. When we perform a bitwise AND operation on these two numbers, we get the binary representation 001, which is equivalent to the decimal number 1.

The M Code Behind Number.BitwiseAnd

Now that we know what Number.BitwiseAnd does, let’s take a look at the M code behind this function. The M code for Number.BitwiseAnd is fairly simple, and it looks like this:

```
(Number as number, BitwiseValue as number) => Number band BitwiseValue
```

Let's break down this code to understand what it's doing. The first line defines the two parameters that the function takes: Number and BitwiseValue. Number is the number that we want to perform the bitwise AND operation on, and BitwiseValue is the value that we want to compare it to.

The second line is where the actual bitwise AND operation is performed. The 'band' keyword is used to perform a bitwise AND operation on the two numbers, and the result is returned as the output of the function.

How to Use Number.BitwiseAnd

Now that we know how Number.BitwiseAnd works and what the M code behind it looks like, let's take a look at how we can use this function in Power Query.

To use Number.BitwiseAnd in Power Query, we first need to create a new query. We can do this by clicking on the 'New Source' button in the Home tab of the Power Query Editor. Once we've created our new query, we can start to write our M code.

To use Number.BitwiseAnd, we simply need to call the function and pass in the two numbers that we want to compare. For example, if we wanted to compare the numbers 5 and 3, we would write the following code:

Number.BitwiseNot

Understanding the Bitwise NOT Operator

Before we dive into the M code behind the Number.BitwiseNot function, let's first understand what the bitwise NOT operator does. The bitwise NOT operator is represented by the tilde (~) symbol and is used to invert the bits of a number. In other words, it converts all 0s to 1s and all 1s to 0s.

For example, if we apply the bitwise NOT operator to the number 5, which is represented in binary as 00000101, we get the result 11111010, which is equivalent to the decimal value -6. This is because the first bit in a binary number represents the sign, and a 1 in this position indicates a negative number.

The Number.BitwiseNot Function

Now that we understand the bitwise NOT operator, let's take a look at the M code behind the Number.BitwiseNot function. The syntax for this function is as follows:

```
Number.BitwiseNot(number as nullable number) as nullable number
```

This function takes a single argument, which is the number that we want to perform the bitwise NOT operation on. The argument can be any nullable numeric value, including integers, decimals, and null values. The function returns a nullable numeric value that represents the result of the operation.

To use the Number.BitwiseNot function in Power Query, we simply call the function and pass in the number that we want to operate on. For example, if we want to perform a bitwise NOT operation on the number 5, we would use the following code:

```
Number.BitwiseNot(5)
```

This would return the result -6, as we saw earlier.

Applications of the Number.BitwiseNot Function

Number.BitwiseOr

But what exactly is a bitwise OR operation? How does it work? And what is the M code behind the Number.BitwiseOr function? In this article, we'll explore the answers to these questions and more.

Understanding Bitwise OR Operations

Before we dive into the M code behind the Number.BitwiseOr function, it's important to understand what a bitwise OR operation is and how it works.

At its most basic level, a bitwise OR operation compares two binary numbers bit by bit and returns a new binary number that has a 1 in each bit position where at least one of the corresponding bits in the original numbers was a 1.

For example, let's say we want to perform a bitwise OR operation on the numbers 5 and 3. In binary form, these numbers are represented as follows:

– 5: 101

– 3: 011

To perform the bitwise OR operation, we compare each bit from left to right and return a new binary number that has a 1 in each position where at least one of the bits in the original numbers was a 1. In this case, the result would be:

– Result: 111

The M Code Behind Number.BitwiseOr

Now that we understand what a bitwise OR operation is and how it works, let's take a look at the M code behind the Number.BitwiseOr function.

The M code for Number.BitwiseOr is surprisingly simple. Here it is:

```
(x as number, y as number) => x bitwise or y
```

This code defines the Number.BitwiseOr function and specifies that it takes two arguments, x and y, both of which are numbers. The function then performs a bitwise OR operation on x and y using the "bitwise or" operator.

It's worth noting that the "bitwise or" operator is not a standard mathematical operator like addition or multiplication. Instead, it's a logical operator that works on individual bits within a binary number.

Number.BitwiseShiftLeft

What is bitwise shifting?

Bitwise shifting is a process of moving the bits of a binary number left or right. In Power Query, we work with decimal numbers, but they are converted to binary numbers behind the scenes to perform bitwise operations. The binary representation of a decimal number is a sequence of 1s and 0s, where each digit is a bit.

For example, the number 5 in binary is 101. If we shift the bits of 5 to the left by 2 positions, we get 10100, which is the binary representation of the number 20. Bitwise shifting is used in various computer science applications, such as encryption, compression, and data storage.

Syntax of Number.BitwiseShiftLeft

The syntax of Number.BitwiseShiftLeft function is as follows:

```
Number.BitwiseShiftLeft(number, count)
```

The function takes two arguments: number and count. Number is the decimal number that we want to shift the bits of, and count is the number of positions we want to shift the bits to the left. The function returns the result as a new number.

Here is an example of using Number.BitwiseShiftLeft function:

```
Number.BitwiseShiftLeft(5, 2)
```

The result of this function is 20, which is the decimal equivalent of the binary number 10100.

M code behind Number.BitwiseShiftLeft

Let's take a closer look at the M code behind the Number.BitwiseShiftLeft function. When we use this function in Power Query, the M code that is generated looks like this:

Number.BitwiseShiftRight

What is a Bitwise Right Shift?

Before we dive into the M code behind the Number.BitwiseShiftRight function, it is important to understand what a bitwise right shift is. A bitwise right shift is an operation that shifts the bits in a binary number to the right by a certain number of positions. The bits that are shifted out of the binary number are discarded, and the bits that are shifted in from the left are set to zero.

For example, let's say we have the binary number 10101110. If we perform a bitwise right shift by 2 positions, the result would be 00101011. The two leftmost bits are shifted out of the binary number, and the two rightmost bits are set to zero.

The M Code Behind Number.BitwiseShiftRight

Now that we understand what a bitwise right shift is, let's take a look at the M code behind the Number.BitwiseShiftRight function. The syntax for this function is as follows:

```
Number.BitwiseShiftRight(number as nullable any, count as number) as nullable any
```

The function takes two arguments: the number to be shifted and the number of positions to shift it to the right. The function returns the shifted number.

Here's an example of how to use the Number.BitwiseShiftRight function in Power Query M:

```
let
  Source = #table({“Number”},{{10}}),
  Shifted = Table.TransformColumns(Source, {“Number”, each Number.BitwiseShiftRight(_, 2), type number})
in
  Shifted
```

In this example, we create a table with a single column called "Number" that contains the value 10. We then use the Number.BitwiseXor

Understanding the Number.BitwiseXor Function

The Number.BitwiseXor function is used to perform a bitwise XOR operation between two numbers. It returns a number that represents the result of the operation. XOR stands for “exclusive or” and it is a logical operation that takes two binary values as input and returns a binary value as output. The operation returns a 1 in each bit position where the corresponding bits of either operand are 1, but not both.

The syntax for the Number.BitwiseXor function is:

```
Number.BitwiseXor(number1, number2)
```

Where `number1` and `number2` are the numbers to be XORed. Both arguments should be integers or numeric expressions that evaluate to integers.

Examples of Number.BitwiseXor Function

Let's look at some examples to understand how the Number.BitwiseXor function works.

Example 1:

```
Number.BitwiseXor(5, 3)
```

Output: 6

Explanation: The binary representation of 5 is 101 and the binary representation of 3 is 011. The bitwise XOR operation between these two numbers is 110, which is equal to 6 in decimal.

Example 2:

```
Number.BitwiseXor(15, 10)
```

Number.Combinations

In this article, we will explore the M code behind the Number.Combinations function and provide examples of how it can be used to solve real-world data problems.

Understanding Number.Combinations

The Number.Combinations function is used to calculate the number of possible combinations of a given set of items. It takes two arguments:

– The first argument is a list of items that will be used to generate combinations.

– The second argument is an integer that specifies the number of items to include in each combination.

The function returns a table that contains all possible combinations of the specified number of items from the input list.

For example, suppose we have a list of four items: A, B, C, and D. We want to generate all possible combinations of three items from this list. We would use the following M code:

```
Number.Combinations({“A”, “B”, “C”, “D”}, 3)
```

This would return a table with the following rows:

A B C
A B D
A C D
B C D

Note that the order of the items within each combination does not matter.

Using Number.Combinations in Practice

The Number.Combinations function can be used to solve a variety of real-world data problems. Here are some examples:

Number.Cos

What is M code?

M code is the language used by Power Query to perform data transformations. It is a functional language that allows you to create complex queries that manipulate data in various ways. M code is generated automatically by the Power Query interface, but you can also edit it manually if you need to perform more advanced transformations.

How does the Number.Cos function work?

The Number.Cos function is a built-in function in Power Query that returns the cosine of a given angle in radians. The syntax of the function is as follows:

```
Number.Cos(number)
```

The function takes a single argument, which is the angle in radians. The function returns the cosine of the angle.

For example, if you want to calculate the cosine of an angle of 45 degrees, you would first convert the angle to radians using the Radians function, like this:

```
=Number.Cos(Radians(45))
```

This would return a value of approximately 0.707.

The M code behind the Number.Cos function

When you use the Number.Cos function in Power Query, the M code generated by the interface looks like this:

```
= (number) => Number.Cos(number)
```

Number.Cosh

What is the Number.Cosh Function?

The Number.Cosh function is used to calculate the hyperbolic cosine of a given number. The hyperbolic cosine is a mathematical function that is similar to the cosine function, but is used for hyperbolic functions. The Number.Cosh function takes a single argument, which is the number for which you want to calculate the hyperbolic cosine.

Understanding the M Code Behind the Number.Cosh Function

To understand the M code behind the Number.Cosh function, we must first understand the M language. M is the programming language used in Power Query to manipulate data and create queries. The M code behind the Number.Cosh function is as follows:

```
Number.Cosh = (x) =>
  let
    e = exp(x),
    en = exp(-x)
  in
    (e + en) / 2
```

Let's break down the code to better understand what is happening. The first line defines the function with a single argument, which is denoted as "x". The "=>" symbol is used to indicate that the code following it is the function's body.

The next few lines are used to calculate the hyperbolic cosine. The "exp" function is used to calculate the value of "e", which is equal to the exponential of "x". The "exp" function is also used to calculate the value of "en", which is equal to the exponential of negative "x".

The final line is used to calculate the hyperbolic cosine of "x". The hyperbolic cosine is calculated by adding "e" and "en" together, and then dividing the result by 2.

Using the Number.Cosh Function in Power Query

Now that we understand the M code behind the Number.Cosh function, let's take a look at how it can be used in Power Query. The Number.Cosh function can be used in a formula to calculate the hyperbolic cosine of a given number. For example, to calculate the hyperbolic cosine of the number 2, you would use the following formula:

Number.Exp

One of the functions you can use in M is Number.Exp. In this article, we will explore the M code behind this function and learn how to use it.

What is Number.Exp?

The Number.Exp function in Power Query is used to calculate the exponential value of a given number. The exponential value of a number is calculated by raising the mathematical constant e (approx. 2.71828) to the power of the number.

For example, if we want to find the exponential value of 2 using the Number.Exp function, we would use the following formula:

```
Number.Exp(2)
```

This would return the value of 7.38906.

Understanding the M Code Behind Number.Exp

Now that we know what the Number.Exp function does let's take a closer look at the M code behind it.

The M code for the Number.Exp function is as follows:

```
(number as number) as number =>
  let
    e = 2.718281828459045,
    result = Number.Power(e, number)
  in
    result
```

Here is a breakdown of what this code does:

- The first line defines the input parameters for the function. In this case, we only have one input parameter, which is the number we

Number.Factorial

In this article, we will explore the M code behind the Number.Factorial function, its syntax, parameters, and usage. So, let’s dive in!

Syntax

The syntax of the Number.Factorial function is as follows:

```
Number.Factorial(number)
```

Here, number is the input for which the factorial needs to be calculated. It must be a non-negative integer.

Parameters

The Number.Factorial function has only one parameter, which is the input number.

Usage

The Number.Factorial function is used to calculate the factorial of a given number. A factorial of a number is the product of all positive integers from 1 to that number. For example, the factorial of 5 (written as 5!) is calculated as follows:

```
5! = 5 x 4 x 3 x 2 x 1 = 120
```

Here, the factorial of 5 is calculated by multiplying all positive integers from 1 to 5.

The Number.Factorial function can be used in Power Query to calculate the factorial of a column of integers. Let's see an example. Suppose we have a table named 'Numbers' with a column named 'Num' that contains integers. We want to calculate the factorial of each number in the 'Num' column and create a new column named 'Factorial' with the calculated values. To achieve this, we can use the following M code:

```
let
Number.From
```

In this article, we will explore the M code behind the Number.From function and how it can be used to convert values into numbers.

Understanding the Number.From Function

The Number.From function is used to convert a value into a number. It takes a single parameter which can be a text, date, datetime, or time value. The syntax for the Number.From function is as follows:

```
Number.From(value as any) as nullable number
```

The value parameter is the value that needs to be converted into a number. The nullable number is the result that is returned by the function.

The Number.From function works by converting the input value into a number. If the input value cannot be converted into a number, the function returns a null value.

The M Code Behind the Number.From Function

The M code behind the Number.From function can be broken down into three main parts: the function definition, the parameter type checking, and the value conversion.

Function Definition

The function definition is the first part of the M code behind the Number.From function. It defines the name of the function, the parameter, and the output value.

```
Number.From = (value) =>
```

The above code defines the Number.From function with a single parameter called value. The arrow operator (=>) is used to indicate the start of the function body.

Parameter Type Checking

Number.FromText

What is Number.FromText?

Number.FromText is a function in Power Query's M language that converts text to a number. It can be extremely useful when working with data that contains numeric values stored as text. The function takes a single parameter, which is the text that needs to be converted to a number.

The M Code Behind Number.FromText

To understand the M code behind Number.FromText, let's first take a look at the function's syntax:

```
Number.FromText(text as text, optional culture as nullable text) as nullable number
```

The function takes two parameters: text and culture. The text parameter is the text that needs to be converted to a number. The culture parameter is an optional parameter that specifies the culture to use when converting the text to a number.

The M code behind Number.FromText is relatively simple. Here's what it looks like:

```
(text as text, optional culture as nullable text) as nullable number =>
   Number.FromText(text, culture)
```

The code defines a function that takes two parameters: text and culture. The function then calls the Number.FromText function, passing in the text and culture parameters.

Using Number.FromText in Power Query

Now that we understand the M code behind Number.FromText, let's take a look at how it can be used in Power Query.

Suppose we have a table with a column called "Revenue" that contains revenue data stored as text. To convert this data to numbers, we can use the Number.FromText function in Power Query.

First, we need to add a new column to our table by clicking the "Add Column" tab and selecting "Custom Column". In the "Custom

Number.IntegerDivide

Understanding the Number.IntegerDivide Function

Before we dive into the M code behind the Number.IntegerDivide function, it’s important to understand how this function works. The function takes two arguments: numerator and denominator. It then divides the numerator by the denominator and returns the integer portion of the result. For example, if the numerator is 10 and denominator is 3, Number.IntegerDivide will return 3.

Here is the syntax of the Number.IntegerDivide function:

```
Number.IntegerDivide(numerator, denominator)
```

The M Code Behind Number.IntegerDivide

The M code behind the Number.IntegerDivide function is relatively simple. The M code for this function is as follows:

```
let
  result = numerator div denominator
in
  result
```

As you can see, the code is quite straightforward. The "let" statement defines a variable called "result" and sets its value to the result of dividing the numerator by the denominator using the "div" operator. The "div" operator is a built-in M function that performs integer division. Finally, the "in" statement returns the value of the "result" variable.

Using Number.IntegerDivide in Power Query

Now that you understand the M code behind the Number.IntegerDivide function, let's see how you can use this function in Power Query. To use Number.IntegerDivide, simply open the Power Query Editor and create a new column. Then, enter the following code in the formula bar:

```
Number.IsEven
```

Here, we will dive deeper into the M code behind the Number.IsEven function. We will examine how the function works and how it can be used in various scenarios.

The M Code Behind the Number.IsEven Function

The M code behind the Number.IsEven function is relatively straightforward. The function takes a single argument, which is the number to be checked. The code then checks whether the number is divisible by 2 without any remainder. If the number is divisible by 2, the function returns true. If not, it returns false.

Here is the M code for the Number.IsEven function:

```
Number.IsEven = (number) =>
  number mod 2 = 0
```

As you can see, the code uses the mod operator to check whether the number is divisible by 2. The mod operator returns the remainder of a division operation. If the remainder is 0, the number is divisible by 2.

Using the Number.IsEven Function

The Number.IsEven function is commonly used in Power Query M to manipulate data based on whether a value is even or odd. For example, you can use the function to filter out odd numbers from a dataset, or to perform certain operations only on even numbers.

Here is an example of using the Number.IsEven function to filter out odd numbers from a dataset:

```
let
  Source = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
  Filtered = List.Select(Source, each Number.IsEven(_) = true)
in
  Filtered
```

Number.IsNaN

What is Number.IsNaN Function?

The Number.IsNaN function in Power Query M language is used to check if a number is NaN. NaN is a value representing undefined or unrepresentable value in floating-point calculations. It is commonly used to represent invalid or missing data. The Number.IsNaN function returns true if the value is NaN, otherwise, it returns false.

The M Code Behind the Number.IsNaN Function

The M code behind the Number.IsNaN function is quite simple. Here is the M code for the Number.IsNaN function:

```
Number.IsNaN = (number) => number <> number;
```

The M code defines the Number.IsNaN function as a lambda function that takes a single argument, which is the number to be checked. The function then returns true if the number is not equal to itself, which is true only when the number is NaN.

How to Use Number.IsNaN Function in Power Query

Here is an example of how to use the Number.IsNaN function in Power Query:

Suppose we have a table with a column named "Amount" that contains some numeric values. We want to create a new column named "IsNaN" that indicates whether each value in the "Amount" column is NaN or not.

1. Select the "Amount" column in the Power Query Editor.
2. Go to the "Add Column" tab and select "Custom Column".
3. In the "Custom Column" dialog, enter "IsNaN" as the column name and enter the following formula:

```
Number.IsNaN([Amount])
```

4. Click "OK" to create the new column.

The formula uses the Number.IsNaN function to check whether each value in the "Amount" column is NaN or not. The function returns

Number.IsOdd

The M code behind the Number.IsOdd function is simple yet effective. Let’s take a closer look at how it works.

How the Number.IsOdd Function Works

The Number.IsOdd function takes a single argument, which is the number you want to check. It then returns a Boolean value indicating whether the number is odd or even.

Here’s the M code behind the function:

```
Number.IsOdd = (number) =>
   if Number.Mod(number, 2) = 1 then true else false
```

Let's break down what this code does.

The first line defines the function and its argument. In this case, the argument is called "number."

The second line checks whether the number is odd or even. It does this by using the Number.Mod function, which returns the remainder of dividing two numbers. If the remainder is 1, then the number is odd. If the remainder is 0, then the number is even.

The third line returns a Boolean value indicating whether the number is odd or even. If the remainder is 1, then the function returns true. If the remainder is 0, then the function returns false.

How to Use the Number.IsOdd Function

Now that we know how the Number.IsOdd function works, let's see how we can use it in Power Query.

Suppose we have a table with a column of numbers called "Values." We want to add a column that indicates whether each value is odd or even.

Here's the M code to do that:

```
let
   Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText(“i45WMjIwMjQ2NtbVU0lSK1Flz0xRijKz8lPMzA28ElyUUouVl5Kz
Number.Ln
```

Understanding the Number.Ln Function

The Number.Ln function in Power Query is used to calculate the natural logarithm of a given number. The natural logarithm is the logarithm to the base e, where e is the mathematical constant approximately equal to 2.71828. The natural logarithm is commonly used in various mathematical and statistical calculations, particularly in data analysis.

The syntax for the Number.Ln function is as follows:

```
Number.Ln(number as nullable number) as nullable number
```

The function takes in a single argument, which is the number for which the natural logarithm is to be calculated. The argument can be a nullable number, which means it can be a number or null.

The M Code Behind the Number.Ln Function

The M code behind the Number.Ln function is relatively simple. Here's the code:

```
let
  numberLn = (number as nullable number) =>
    if number = null then null else Number.Log(number,
2.718281828459045235360287471352662497757247093699959574966967627724076630353 5)
in
  numberLn
```

Let's break down the code:

- The first line declares a variable called "numberLn", which is a function that takes in a single argument called "number". The "nullable number" data type is used to indicate that the argument can either be a number or null.

Number.Log

What is the Number.Log Function?

The Number.Log function in Power Query is used to calculate the logarithm of a number to a specified base. The syntax of the function is as follows:

```
Number.Log(number as nullable number, base as nullable number) as nullable number
```

The function takes two arguments:

- number: the number for which you want to calculate the logarithm.
- base: the base of the logarithm. If the base is not specified, it defaults to the natural logarithm (e).

The function returns the logarithm of the number to the specified base.

The M Code Behind Number.Log

The M code behind the Number.Log function is relatively simple. Here is the M code for the function:

```
let
  NumberLog = (number as nullable number, base as nullable number) as nullable number =>
    if number = null or base = null then null
    else if number <= 0 or base <= 0 or base = 1 then error "Invalid input"
    else Number.Log(number, base)
in
  NumberLog
```

Let's break down the code and see what each part does.

The let Expression

Number.Log10

(blank)

Number.Mod

What is Number.Mod?

Before we dive into the M code behind Number.Mod, let’s quickly review what this function does. Number.Mod takes two arguments, divisor and dividend, and returns the remainder after dividing dividend by divisor. For example, Number.Mod(10, 3) would return 1, since 3 goes into 10 three times with a remainder of 1.

The M code behind Number.Mod

In Power Query, all functions are written in the M language. The M code behind the Number.Mod function is relatively simple. Here is the code:

```
let
  Number.Mod = (dividend, divisor) =>
    let
      quotient = Number.IntegerDivide(dividend, divisor),
      remainder = dividend – divisor  quotient
    in
      remainder
in
  Number.Mod
```

Let's take a closer look at what's happening here. The Number.Mod function is defined using the "let" keyword, which allows us to define variables and functions. The function takes two arguments, dividend and divisor. Inside the function, we define two more variables - quotient and remainder.

The quotient is calculated using the Number.IntegerDivide function, which returns the result of dividing dividend by divisor and rounding down to the nearest integer. This gives us the number of times that divisor goes into dividend.

The remainder is then calculated by subtracting the product of divisor and quotient from dividend. This gives us the remaining amount after all the full divisors have been subtracted from the dividend.

Number.Permutations

In this article, we will explore the M code behind the Number.Permutations function, and show how it can be used to calculate the number of permutations for a given set of items.

What are Permutations?

Permutations are a way of arranging a set of items in a particular order. For example, consider the set {1, 2, 3}. There are six possible permutations of this set:

– {1, 2, 3}
– {1, 3, 2}
– {2, 1, 3}
– {2, 3, 1}
– {3, 1, 2}
– {3, 2, 1}

Note that each permutation is a unique arrangement of the items in the set. Also, note that the number of possible permutations for a set of n items is given by n factorial (n!).

The Number.Permutations Function

The Number.Permutations function is a built-in function in Power Query that allows users to calculate the number of permutations for a given set of items. The syntax of the function is as follows:

```
Number.Permutations(n, k)
```

where n is the total number of items in the set, and k is the number of items to be arranged in each permutation.

For example, to calculate the number of permutations of a set of 5 items, where each permutation is a combination of 3 items, we can use the following formula:

```
Number.Permutations(5, 3)
```

Number.Power

Understanding the Number.Power Function

The Number.Power function is used to raise a number to a specified power. The syntax for the function is as follows:

```
Number.Power(number, power)
```

The number argument is the base number that will be raised to the power. The power argument is the exponent to which the base number will be raised.

For example, if we want to raise the number 2 to the power of 3, we would use the following formula:

```
Number.Power(2, 3)
```

This would return the value 8, which is 2 raised to the power of 3.

The M Code Behind the Number.Power Function

Under the hood, the Number.Power function is implemented in M code. M is the language used by Power Query to perform data transformations and manipulations.

The M code for the Number.Power function is as follows:

```
let
  Source = (number as number, power as number) =>
    let
      result = Number.Power(number, power)
    in
Number.Random
```

In this article, we'll dive into the M code that powers Number.Random, exploring its syntax and functionality.

Syntax of the Number.Random Function

The syntax of the Number.Random function is straightforward. It takes no arguments and returns a decimal number between 0 and 1. Here's the basic syntax:

```
Number.Random()
```

That's it! When you call the Number.Random function, it generates a random decimal number.

The M Code Behind Number.Random

Behind the scenes, the M code that powers Number.Random is a bit more complex. Here's what it looks like:

```
let
  Source = #table({“”}, {{1}}),
  RandomNumber = Number.RandomBetween(0, 100000) / 100000
in
  RandomNumber
```

Let's break this down line by line.

Line 1: Creating a Table

The first line of the code creates a table with a single column and a single row. This table is essentially just a placeholder that allows us to generate a random number.

Number.RandomBetween

What is M Code?

M code is the language used in Power Query to create custom functions and queries. It is a functional language that is used to manipulate data. M code is similar to Excel formulas, but it is more powerful and can handle larger data sets.

How to Use the Number.RandomBetween Function

The Number.RandomBetween function is used to generate a random number between two specified numbers. The syntax for the function is as follows:

```
Number.RandomBetween(lower as number, upper as number) as number
```

The function takes two arguments, the lower bound and the upper bound, and returns a random number between those two numbers. For example, if you want to generate a random number between 1 and 10, you would use the following formula:

```
Number.RandomBetween(1, 10)
```

The M Code Behind Number.RandomBetween

The M code behind the Number.RandomBetween function is quite simple. The function is defined as follows:

```
let
  Number.RandomBetween = (lower as number, upper as number) =>
    lower + Number.Random()  (upper – lower),
  result = Number.RandomBetween
in
Number.Round
```

What is Number.Round?

Number.Round is a function that is used to round numbers to a specified number of decimal places. The function takes two arguments: the number to be rounded, and the number of decimal places to round to. The function returns the rounded number as a decimal.

The M Code Behind Number.Round

The M code behind the Number.Round function is quite simple. Here is the code:

```
Number.Round(number as nullable number, digits as number) as nullable number =>
  if number is null then
    null
  else
    Number.RoundUp(number  Number.Power(10, digits)) / Number.Power(10, digits)
```

Let's break this code down:

- The function takes in two arguments: 'number' and 'digits'. 'Number' is the number to be rounded, and 'digits' is the number of decimal places to round to.
- The function then checks if the number is null. If the number is null, the function returns null.
- If the number is not null, the function multiplies the number by 10 to the power of the number of decimal places to round to.
- The function then rounds up the result of step 3.
- Finally, the function divides the result of step 4 by 10 to the power of the number of decimal places to round to.

Examples

Let's take a look at some examples to see how the Number.Round function works.

Example 1

Suppose we have the number 3.14159 and we want to round it to 2 decimal places. Here is the M code we would use:

```
Number.RoundAwayFromZero
```

Syntax

The syntax of the Number.RoundAwayFromZero function is as follows:

```
Number.RoundAwayFromZero(number as nullable any) as nullable any
```

The function takes one argument, which is the number to be rounded. The argument must be a numeric value or a value that can be converted to a numeric value. The function returns the rounded number as a nullable any value.

M Code

The M code behind the Number.RoundAwayFromZero function is as follows:

```
let
  Number.RoundAwayFromZero = (number as nullable any) as nullable any =>
    if number = null then null else
    if number >= 0 then Number.RoundUp(number) else Number.RoundDown(number)
in
  Number.RoundAwayFromZero
```

The code defines a function named Number.RoundAwayFromZero that takes one argument, number. The function first checks if the argument is null and returns null if it is. If the argument is not null, the function checks if the argument is greater than or equal to 0. If it is, the function uses the Number.RoundUp function to round the number up to the nearest integer. If the argument is less than 0, the function uses the Number.RoundDown function to round the number down to the nearest integer.

Examples

Here are some examples of how the Number.RoundAwayFromZero function can be used:

Number.RoundDown

What is the Number.RoundDown Function?

The Number.RoundDown function is a Power Query M function used to round a number down to the nearest integer. It takes two arguments: the number you want to round and the number of decimal places you want to round down to. Here is an example of the Number.RoundDown function:

```
Number.RoundDown(3.14159, 2)
```

In this example, the function rounds the number 3.14159 down to 2 decimal places, resulting in the number 3.14.

The M Code Behind the Number.RoundDown Function

The M code behind the Number.RoundDown function is fairly simple. Here is the M code for the Number.RoundDown function:

```
let
  roundDown = (number as nullable number, decimals as number) =>
    if number <> null then Number.RoundDown(number, decimals) else null
in
  roundDown
```

The M code defines a function called 'roundDown' that takes two arguments: 'number' and 'decimals'. The function then checks if the 'number' argument is not null. If the 'number' argument is not null, the function calls the Number.RoundDown function with the 'number' and 'decimals' arguments. If the 'number' argument is null, the function returns null.

How to Use the Number.RoundDown Function

To use the Number.RoundDown function in Power Query, follow these steps:

1. Open Power Query and select the data you want to transform.

Number.RoundTowardZero

What is the Number.RoundTowardZero function?

The Number.RoundTowardZero function is used to round a number towards zero to the nearest integer. This function takes a single argument, which is the number that you want to round. If the number is positive, the function will round down to the nearest integer. If the number is negative, the function will round up to the nearest integer. The syntax for the Number.RoundTowardZero function is as follows:

Number.RoundTowardZero(number as nullable number) as nullable number

How does the Number.RoundTowardZero function work?

The M code behind the Number.RoundTowardZero function uses a simple logic to round a number towards zero. If the number is positive, the function subtracts 0.5 from the number and then rounds down to the nearest integer. If the number is negative, the function adds 0.5 to the number and then rounds up to the nearest integer.

Let's look at an example to understand this better. Suppose we have a number 2.8 that we want to round towards zero. The Number.RoundTowardZero function will first subtract 0.5 from 2.8, which gives us 2.3. The function will then round this number down to the nearest integer, which is 2. So, the result of the Number.RoundTowardZero function for the number 2.8 is 2.

Now, let's consider another example where we have a number -2.8 that we want to round towards zero. The Number.RoundTowardZero function will first add 0.5 to -2.8, which gives us -2.3. The function will then round this number up to the nearest integer, which is -2. So, the result of the Number.RoundTowardZero function for the number -2.8 is -2.

How to use the Number.RoundTowardZero function?

To use the Number.RoundTowardZero function in Power Query, you need to follow these steps:

1. Open Power Query Editor by clicking on the "Transform data" button in Power BI Desktop.
2. Select the column that contains the numbers that you want to round towards zero.
3. Click on the "Add Column" tab in the ribbon and select "Custom Column".
4. In the "Custom Column" dialog box, enter a name for the new column and the formula for the Number.RoundTowardZero function. For example, if you want to round the numbers in column "Amount" towards zero, you can use the following formula:

Number.RoundUp

In this article, we will explore the M code behind the Number.RoundUp function and its various applications.

Understanding the Syntax of the Number.RoundUp Function

The Number.RoundUp function takes two arguments – the first is the value to be rounded up, and the second is the number of decimal places to round up to. The syntax of the function is as follows:

```
Number.RoundUp(value as any, optional digits as nullable number) as nullable any
```

Here, the argument 'value' can be any numeric expression in Power Query, while the argument 'digits' is an optional argument that specifies the number of decimal places to round up to. If the 'digits' argument is not specified, the function defaults to round up to the nearest whole number.

How the Number.RoundUp Function Works

The Number.RoundUp function rounds up a given numeric value to the specified number of decimal places. To understand how the function works, let's take an example.

Suppose we have a numeric value of 1.2345, and we want to round it up to two decimal places. We can do this by using the Number.RoundUp function as follows:

```
Number.RoundUp(1.2345, 2)
```

The function will return the value 1.24, which is the result of rounding up 1.2345 to two decimal places.

Applications of the Number.RoundUp Function

The Number.RoundUp function can be used in many scenarios where rounding up numeric values is necessary. Some of the common applications of this function are:

Financial Calculations

Number.Sign

Understanding the Number.Sign Function

The Number.Sign function is a simple but powerful function that is used to determine the sign of a number. The function takes a single argument, which is the number that you want to determine the sign of. The function then returns -1 if the number is negative, 0 if the number is zero, and 1 if the number is positive.

The syntax for the Number.Sign function is as follows:

```
Number.Sign(number)
```

Here, number is the number that you want to determine the sign of.

The M Code Behind the Number.Sign Function

The M code behind the Number.Sign function is relatively simple. The function is defined as follows:

```
Number.Sign = (number as number) as number =>
  if number < 0 then
    -1
  else if number > 0 then
    1
  else
    0
```

Here, we define the Number.Sign function and specify that it takes a single argument, which is a number. We then use an if-else statement to determine the sign of the number. If the number is less than 0, we return -1. If the number is greater than 0, we return 1. Finally, if the number is equal to 0, we return 0.

Number.Sin

The Number.Sin function is used to calculate the sine of a given angle in radians. It takes an argument, which is the angle in radians, and returns the sine value. The sine function is a mathematical function that is used to calculate the ratio of the length of the opposite side to the length of the hypotenuse of a right-angled triangle.

Understanding the M Code Behind the Number.Sin Function

The M code behind the Number.Sin function is quite simple. It uses the built-in Math function in M to calculate the sine of the given angle. Here is the M code for the Number.Sin function:

```
let
  Number.Sin = (angle) =>
    Math.Sin(angle)
in
  Number.Sin
```

In the above code, the 'let' keyword is used to define a new function called Number.Sin. The function takes a single argument, angle, which represents the angle in radians. The '=>' symbol is used to define the body of the function, which is the Math.Sin function. The 'in' keyword is used to specify the output of the function, which is Number.Sin.

Using the Number.Sin Function in Power Query

Now that we know the M code behind the Number.Sin function, let's see how we can use it in a Power Query transformation.

Suppose we have a table with a column that contains angles in radians. We want to calculate the sine of each angle and add it as a new column. Here's how we can do that using the Number.Sin function:

1. Load the table into Power Query.
2. Select the column that contains the angles in radians.
3. Go to the 'Add Column' tab and click on 'Custom Column'.
4. In the 'Custom Column' dialog box, enter the following formula:

```
Number.Sinh
```

What is Hyperbolic Sine?

Before we dive into the M code, let's briefly discuss what hyperbolic sine is. The hyperbolic sine of a number is a mathematical function that describes the shape of a hanging cable or chain, among other things. It's defined as:

```
sinh(x) = (e^x – e^-x) / 2
```

Where e is the mathematical constant approximately equal to 2.71828, and x is the input value.

The hyperbolic sine function has a range of values from negative infinity to positive infinity and is an odd function, meaning that:

```
sinh(-x) = -sinh(x)
```

The M Code for Number.Sinh

The M code behind the Number.Sinh function is relatively simple. Here's what it looks like:

```
(x) => Number.Sinh(x)
```

In this code, we define a parameter x, which represents the input value. We then call the built-in Number.Sinh function, passing in x as the argument.

How the Code Works

When you use the Number.Sinh function in Power Query, you provide a single argument, which is the number you want to find the hyperbolic sine of. Power Query then uses the M code we just discussed to return the result.

For example, if you had a table of values that you wanted to find the hyperbolic sine of, you could use the following M code:

```
let
  Source = Table.FromRows({{1}, {2}, {3}}, {"Number"}),
  #"Added Custom" = Table.AddColumn(Source, "Hyperbolic Sine", each Number.Sinh([Number]))
in
  #"Added Custom"
```

In this code, we first create a table with three rows and one column called "Number". We then add a custom column called "Hyperbolic Sine" using the Table.AddColumn function. In the each clause of this function, we call the Number.Sinh function, passing in the value of the "Number" column for each row.

The result of this code would be a new table with two columns, "Number" and "Hyperbolic Sine", where the "Hyperbolic Sine" column contains the hyperbolic sine of each value in the "Number" column.

The Number.Sinh function is a useful tool in Power Query for finding the hyperbolic sine of a given number. Understanding the M code

Number.Sqrt

Understanding the Number.Sqrt Function

The Number.Sqrt function is a mathematical function that is used to calculate the square root of a given number. It is a built-in function in Power Query M and is used in various data transformation tasks. The syntax for the Number.Sqrt function is as follows:

```
Number.Sqrt(number as any) as nullable any
```

The function takes a single argument, which is the number whose square root needs to be calculated. The argument can be of any data type, including integers, decimals, and fractions. The function returns the square root of the number as a nullable any data type.

Examples of Using the Number.Sqrt Function

Let's take a look at some examples of using the Number.Sqrt function in Power Query M.

Example 1: Calculating the Square Root of a Number

Suppose we have a table that contains a column named "Value" and we want to calculate the square root of each value in that column. We can use the Number.Sqrt function to achieve this as shown below:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WizJQ0y1WljIwMjIxsTQ3MlGyMjSwNjA1N7IwNzYyMjA1lFQyNDbV0lLLSrJzCvLLQpKdYB", BinaryEncoding.Base64)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Value = _t])),
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"Value", type number}}),
  #"Added Custom" = Table.AddColumn(#"Changed Type", "Sqrt", each Number.Sqrt([Value])),
  #"Removed Columns" = Table.RemoveColumns(#"Added Custom",{"Value"})
in
  #"Removed Columns"
```

Number.Tan

In this article, we will take a deep dive into the M code behind the Power Query M function Number.Tan. We will explore how this function works and how it can be used to perform complex data transformations.

Understanding the Number.Tan Function

The Number.Tan function is a mathematical function that returns the tangent of a given angle. It takes one argument, which is the angle in radians. The function returns a decimal number that represents the tangent of the angle.

The syntax of the Number.Tan function is as follows:

```
Number.Tan(number)
```

Where `number` is the angle in radians.

For example, if we want to find the tangent of an angle of 45 degrees, we need to convert it to radians first. The formula to convert degrees to radians is:

```
radians = degrees  (pi/180)
```

Using this formula, we can convert 45 degrees to radians as follows:

```
radians = 45  (pi/180)
radians = 0.785398
```

Now, we can use the Number.Tan function to find the tangent of this angle as follows:

Number.Tanh

Understanding the Hyperbolic Tangent Function

Before diving into the M code for the Number.Tanh function, it is important to understand what the hyperbolic tangent function is and how it works. The hyperbolic tangent function, or tanh, is a mathematical function that maps real numbers to the interval (-1, 1). It is defined as the ratio of the hyperbolic sine function to the hyperbolic cosine function:

tanh(x) = sinh(x) / cosh(x)

The hyperbolic sine and cosine functions are defined as:

sinh(x) = (e^x – e^-x) / 2

cosh(x) = (e^x + e^-x) / 2

where e is the base of the natural logarithm.

The tanh function has several properties that make it useful in data analysis. First, it is an odd function, which means that tanh(-x) = -tanh(x). Second, it is continuous and differentiable on its entire domain. This makes it useful for curve fitting and optimization tasks.

The M Code for the Number.Tanh Function

Now that we have a basic understanding of the tanh function, let’s take a look at the M code for the Number.Tanh function in Power Query. The function takes a single argument, a number, and returns the hyperbolic tangent of that number.

```
(Number as nullable number) as nullable number =>
  if Number = null then null else if Number = 0 then 0 else
    let
      d = Exp(-2  Number),
      n = 1 – d,
      t = n / (1 + d)
    in
      if Number < 0 then -t else t
```

The M code for the Number.Tanh function is a bit more complex than the mathematical definition we described earlier. Let's break it

Number.ToText

What is Power Query?

Before we dive into the M code behind the Number.ToText function, let’s briefly discuss what Power Query is. Power Query is a data transformation and cleansing tool that is built into Microsoft Excel and Power BI. It allows you to easily connect to various data sources, clean and transform data, and load it into your workbook or Power BI report.

The Number.ToText Function

The Number.ToText function is a built-in function in Power Query that allows you to convert a number to text. Here’s an example of how it works:

```
Number.ToText(1234.56)
```

This will return the text "1234.56". You can also specify the number of decimal places you want to format the number to:

```
Number.ToText(1234.56, “#.00”)
```

This will return the text "1234.56". The "#" character in the format string represents a digit, while the "0" character represents a zero. So in this case, we're asking Power Query to format the number to two decimal places.

The M Code Behind Number.ToText

Now that we've covered the basics of the Number.ToText function, let's take a look at the M code behind it. When you use the Number.ToText function in Power Query, the M code that is generated looks like this:

```
let
  Source = 1234.56,
OData.Feed
```

One of the most powerful features of Power Query is the M language, which is used to write custom functions and transforms. In this article, we'll take a closer look at the M code behind the OData.Feed function in Power Query, which is used to connect to OData data sources.

What is OData?

OData is an open standard protocol for querying and updating data, based on HTTP and RESTful web services. It is designed to enable interoperability between different systems and platforms, by providing a common language for data exchange.

OData data sources can be accessed using a URL, which returns a response in XML or JSON format. This response can then be parsed and transformed using Power Query, using the OData.Feed function.

How the OData.Feed Function Works

The OData.Feed function is used to connect to an OData data source and retrieve data. It takes a single argument, which is a URL string that points to the data source.

Here's an example of the OData.Feed function in action:

```
let
  url = "https://services.odata.org/V4/Northwind/Northwind.svc",
  source = OData.Feed(url)
in
  source
```

In this example, we're connecting to the Northwind sample data source, which is hosted by the OData.org website. The OData.Feed function retrieves the data from the URL and returns a table that can be further transformed using Power Query.

Understanding the M Code Behind the OData.Feed Function

Behind the scenes, the OData.Feed function is using M code to perform the following steps:

1. Send an HTTP GET request to the specified URL
2. Parse the response XML or JSON into a table format

Odbc.DataSource

One of the most useful data sources in Power Query is ODBC (Open Database Connectivity), which allows users to connect to almost any database that has an ODBC driver installed. The Odbc.DataSource M function is used to establish a connection to an ODBC data source, and in this article, we will explore the M code behind this function in detail.

What is the Odbc.DataSource function?

The Odbc.DataSource function is a built-in M function in Power Query that is used to connect to a data source using an ODBC connection. The function takes several parameters, including the name of the data source, the ODBC driver to use, and any additional options that are required to establish the connection.

Here is an example of how the Odbc.DataSource function can be used to connect to a MySQL database:

```
Odbc.DataSource("dsn=myDatabase;uid=myUsername;pwd=myPassword", [HierarchicalNavigation=true])
```

In this example, the function takes a connection string that specifies the name of the data source, as well as the username and password required to connect to the database. The function also specifies an additional option called HierarchicalNavigation, which tells Power Query to create a hierarchical view of the data in the database.

Understanding the Odbc.DataSource function in more detail

To understand how the Odbc.DataSource function works, it is important to understand the different parameters that can be used with this function. Here is a brief overview of each parameter:

- Connection string: This is the most important parameter of the function, as it specifies the name of the data source and any additional connection details that are required to establish the connection. The connection string is usually provided by the ODBC driver, and can be found in the ODBC Data Source Administrator tool in Windows.
- Command timeout: This parameter specifies the maximum amount of time that the ODBC driver will wait for a command to complete before timing out. This can be useful when working with large datasets or slow queries.
- Hierarchical navigation: This parameter specifies whether Power Query should create a hierarchical view of the data in the database. When this option is set to true, Power Query will create relationships between tables based on foreign keys and other relationships in the database.

Odbc.InferOptions

What is the Odbc.InferOptions Function?

The Odbc.InferOptions function is a Power Query M function that allows users to get information about the data source and customize their queries accordingly. This function is used with ODBC (Open Database Connectivity) data sources, which are a type of database connectivity standard that enables different applications to communicate with each other using SQL (Structured Query Language). InferOptions function analyzes the data structure of the ODBC data source and returns a record with the following options:

How to Use the Odbc.InferOptions Function?

The Odbc.InferOptions function can be used in two ways:

1. Automatically Generate a Query

By default, the Odbc.InferOptions function is used to automatically generate a query based on the data source. Users can simply select the ODBC data source from the Power Query Editor and click on “Edit” to open the query editor. Power Query will automatically generate a query using the Odbc.InferOptions function and display the data in the preview window.

2. Customize a Query

Users can also use the Odbc.InferOptions function to customize their queries according to the data source. In this case, users need to create a new query and manually add the Odbc.InferOptions function to the M code. The following is an example of how to use the Odbc.InferOptions function to extract data from an ODBC data source:

```
let
  Source = Odbc.DataSource("dsn=ExampleDSN"),
Odbc.Query
```

Understanding the Odbc.Query function

The Odbc.Query function is a Power Query M function that allows users to connect to any data source that has an ODBC driver installed. The function takes two arguments: the connection string and the SQL query. The connection string specifies the details of the ODBC data source, such as the server name, database name, username, and password. The SQL query specifies the data that needs to be retrieved from the data source.

Here is an example of the Odbc.Query function in action:

```
let
  connectionString = “dsn=SampleDB;uid=sa;pwd=Password123;”,
  query = “SELECT  FROM Customers”
in
  Odbc.Query(connectionString, query)
```

In this example, we are connecting to a data source named SampleDB using the ODBC driver. We are then retrieving all the data from the Customers table using the SQL query "SELECT FROM Customers".

The M code behind the Odbc.Query function

The M code behind the Odbc.Query function is relatively simple. Here is the code:

```
let
  ODBCDataSource = Odbc.DataSource(connectionString),
  ODBCQuery = Odbc.Query(ODBCDataSource, query)
in
  ODBCQuery
```

OleDb.DataSource

In this article, we'll take a closer look at the M code behind the OleDb.DataSource function. We'll explore its syntax, parameters, and options, and provide practical examples that demonstrate how to use it effectively.

Understanding OleDb.DataSource

Before we dive into the code behind this function, let's briefly discuss the concept of OLE DB. OLE DB (Object Linking and Embedding, Database) is a Microsoft technology that provides a standard interface to access various data sources, including databases, spreadsheets, and flat files. By using an OLE DB connection, Power Query can communicate with different database systems, such as SQL Server, Oracle, and MySQL.

Now that we have a basic understanding of OLE DB, let's move on to the OleDb.DataSource function. This function is used to connect to a database using an OLE DB connection. It takes several parameters, including the provider, server, database, and any required credentials.

Here's the basic syntax of the OleDb.DataSource function:

```
OleDb.DataSource(connectionString as text, optional options as nullable record) as table
```

Let's break down each part of this syntax:

- connectionString: This is a required parameter that specifies the connection string used to connect to the database. The connection string contains information about the database provider, server, database name, and any necessary credentials. It must be provided as a text value.
- options: This is an optional parameter that allows you to specify additional options for the connection, such as the query timeout or the maximum number of rows to retrieve. These options are specified as a record value.
- table: This is the output of the function, which is a table containing the data retrieved from the database.

Using OleDb.DataSource in Practice

Now that we know the syntax and parameters of the OleDb.DataSource function, let's explore some practical examples of how to use it.

Example 1: Connecting to a SQL Server Database

Suppose you have a SQL Server database called "Sales" on a server named "Server1", and you want to connect to it using Power Query.

OleDb.Query

Behind the scenes, the OleDb.Query function is powered by M code, a functional programming language used by Power Query to perform data transformations. In this article, we will take a closer look at the M code behind the OleDb.Query function, and explore how it works to execute SQL queries against relational databases.

Understanding the OleDb.Query Function

Before we dive into the M code behind the OleDb.Query function, let's first take a closer look at what this function does. The OleDb.Query function is used to execute SQL queries against relational databases, and returns the results of these queries as a table. To use the OleDb.Query function, you will need to provide the following inputs:

– Connection: The connection to the database that you want to execute the query against.
– Query: The SQL query that you want to execute.
– Options: Additional options, such as specifying the encoding or specifying how null values should be handled.

Here is an example of how the OleDb.Query function might be used to execute a simple SQL query against a database:

```
let
  Source = OleDb.Query(
    "Provider=SQLNCLI11;Server=myServerinstance;Database=myDB;Trusted_Connection=yes;",
    "SELECT  FROM myTable",
    [Encoding=1252, NullValueHandling=Ignore]
  )
in
  Source
```

In this example, we are connecting to a SQL Server database using the SQL Native Client provider, and executing a simple SELECT query against a table called myTable. We are also specifying that any null values should be ignored, and that the encoding should be set to 1252.

How the M Code Behind OleDb.Query Works

Oracle.Database

Introduction to Oracle.Database function

The Oracle.Database function is part of the Power Query M language. It's used to connect to and retrieve data from Oracle databases. This function takes several parameters such as server, database, username, and password. Once connected, you can use other M functions to query and transform data.

M Code Behind Oracle.Database

The M code behind the Oracle.Database function is quite complex. It includes several steps to connect to the Oracle database and retrieve data. The following is a breakdown of the M code behind the Oracle.Database function:

1. Create a connection string: The first step is to create a connection string. This string includes information about the Oracle server, database, username, and password. The connection string is created using the following M code:

```
let
  server = "OracleServerName",
  database = "OracleDatabaseName",
  user = "OracleUserName",
  password = "OraclePassword",
  connectionstring = "Provider=OraOLEDB.Oracle;Data Source=" & server & ";User Id=" & user & ";Password=" & password & ";"
in
  connectionstring
```

2. Connect to the Oracle database: The next step is to connect to the Oracle database using the connection string. This is done using the following M code:

```
let
  connectionstring = [connection string created in step 1],
```

Parameter types

This article will focus on the M code behind Power Query's parameter types. Understanding the different parameter types available can help you more effectively work with and manipulate data.

The Three Main Parameter Types

The M language has three main parameter types: text, number, and any. Here's a brief overview of each type:

Text Parameters

Text parameters refer to any input that consists of text, such as words, phrases, and sentences. Text parameters are typically used when working with text-based data, such as names, addresses, and descriptions.

When using text parameters, it's important to consider how the input data might be formatted. For example, if you're working with names, you may want to account for cases where the first and last names are separated by a comma or middle initial.

Number Parameters

Number parameters refer to any input consisting of a numerical value. This could include integers, decimals, or percentages. Number parameters are commonly used when working with financial data, such as sales figures or budgets.

When working with number parameters, it's important to consider how the data is formatted. For example, you may need to account for currency symbols or decimal points. Additionally, you may need to convert data from one unit of measurement to another, such as from pounds to kilograms.

Any Parameters

Any parameters refer to inputs that can be either text or numeric. This is the most flexible parameter type, as it can accommodate a wide range of data types.

When using any parameters, it's important to consider the potential for ambiguous data. For example, if you're working with a column that could contain either text or numbers, you may need to specify how the data should be interpreted.

Additional Parameter Types

In addition to the three main parameter types, there are several additional parameter types that can be used in M function code. Here's a brief overview:

Logical Parameters

Logical parameters refer to inputs that represent a true or false value. This could include inputs such as "yes" or "no", "true" or "false", or "1" or "0". Logical parameters are commonly used in functions such as IF statements.

Date/Time Parameters

Pdf.Tables

Behind the scenes, the Pdf.Tables function is powered by M code, the programming language used by Power Query. In this article, we will take a closer look at the M code behind the Pdf.Tables function and explore how it works.

Understanding the Pdf.Tables Function

Before delving into the M code behind the Pdf.Tables function, it's important to understand what the function does. Essentially, the function takes a PDF file as input and returns a table of data that represents the tables contained within the PDF.

The function has three parameters:

1. Source: This parameter specifies the path to the PDF file that you want to extract tables from.
2. Option1: This parameter is optional and can be used to specify additional options for extracting tables from the PDF. For example, you can use this parameter to specify the page range that you want to extract tables from.
3. Option2: This parameter is also optional and can be used to specify additional options for extracting tables from the PDF. For example, you can use this parameter to specify the password for a password-protected PDF file.

The M Code Behind the Pdf.Tables Function

Now that we understand what the Pdf.Tables function does, let's take a look at the M code that powers it. When you use the Pdf.Tables function in Power Query, the function generates M code that looks something like this:

```
let
  Source = Pdf.Tables(“C:UsersUserNameDocumentsSample.pdf”),
  #”Table1" = Source{0}[Table]
in
  #”Table1"
```

Let's break down this code line by line to understand what's going on:

1. let: This keyword is used to declare a variable in M code. In this case, we are declaring a variable called "Source".
2. Source: This is the name of the variable we are declaring. We are using the Pdf.Tables function to extract tables from the PDF file located at "C:UsersUserNameDocumentsSample.pdf".

Percentage.From

Understanding the Percentage.From Function

The Percentage.From function is used to calculate the percentage of a number relative to another number. For example, if you want to know what percentage of sales came from a particular region, you can use the Percentage.From function.

The syntax for the Percentage.From function is as follows:

```
Percentage.From(numerator as any, denominator as any) as any
```

The numerator is the number you want to calculate the percentage for, and the denominator is the total number you are comparing it to. The function returns the percentage as a decimal.

The M Code Behind the Percentage.From Function

Now that we understand the basics of the Percentage.From function, let's take a look at the M code behind it.

The M code for the Percentage.From function is as follows:

```
let
  percentage = (numerator, denominator) =>
    if denominator = 0 then null else numerator / denominator,
  result = percentage
in
  result
```

Let's break this down line by line.

The first line starts the M code with the `let` keyword. This is used to define variables.

The next line defines the `percentage` variable as a function. The function takes in two parameters - `numerator` and `denominator`.

PostgreSQL.Database

Introduction to PostgreSQL.Database

PostgreSQL is a popular open-source relational database management system that is widely used by developers and businesses around the world. With PostgreSQL.Database, you can connect to a PostgreSQL database and extract data into Power Query for further analysis and transformation. The function takes four arguments:

– server: the name or IP address of the PostgreSQL server

– database: the name of the database you want to connect to

– options: a list of optional parameters such as user name, password, and SSL configuration

– query: an optional SQL query to filter and extract data from the database

Understanding the M Code Behind PostgreSQL.Database

When you use PostgreSQL.Database in Power Query, the function generates M code that connects to the PostgreSQL database and retrieves data. Here is an example of the M code generated by the function:

```
let
  Source = PostgreSQL.Database(“localhost”, “database_name”, [CreateNavigationProperties=true]),
  schema = Source{[Schema=”public”,Item=”table_name”]}[Data],
in
  schema
```

Let's break down this code into its components:

- The first line sets the variable "Source" to the result of the PostgreSQL.Database function. The function takes three arguments: the server name or IP address, the database name, and a list of optional parameters.
- The second line sets the variable "schema" to the data retrieved by the "Source" variable. In this example, we're extracting all the data from a table named "table_name" in the "public" schema of the database.
- The third line returns the "schema" variable, which contains the data retrieved from the database.

You can modify this code to suit your needs. For example, you can add a SQL query to the PostgreSQL.Database function to filter the

Progress.DataSourceProgress

In this article, we will explore the M code behind the Progress.DataSourceProgress function, which is used to monitor the progress of data loading operations in Power Query.

What is Progress.DataSourceProgress?

Progress.DataSourceProgress is a Power Query M function that returns a record with information about the progress of a data loading operation. This function is typically used within custom functions or scripts to monitor the progress of a data load and to provide feedback to the user.

The record returned by Progress.DataSourceProgress contains the following fields:

– CurrentByteCount: The number of bytes that have been loaded so far.

– TotalByteCount: The total number of bytes that need to be loaded.

– IsDataAvailable: A Boolean value that indicates whether data is available for processing.

– IsDone: A Boolean value that indicates whether the data loading operation is complete.

– Message: A text message that describes the current state of the data loading operation.

How to use Progress.DataSourceProgress

To use Progress.DataSourceProgress, you need to include it in your custom functions or scripts. Here is an example of how to use Progress.DataSourceProgress to monitor the progress of a data loading operation:

```
let
  Source = Excel.Workbook(File.Contents(“C:SalesData.xlsx”), null, true),
  SalesData_Table = Source{[Item=”SalesData”,Kind=”Table”]}[Data],
  ProgressMonitor = (Current, Total) =>
    let
      Progress = Progress.DataSourceProgress(),
      Message = “Loading SalesData (” & Text.From(Progress[CurrentByteCount]) & ” of ” & Text.From(Progress[TotalByteCount]) & ” bytes)”
    in
      if Progress[IsDone] then
RData.FromBinary
```

Understanding RData

Before we dive into the M code behind RData.FromBinary, it's important to understand what RData is. R is a programming language and software environment for statistical computing and graphics. It is widely used among statisticians and data scientists for data analysis, data visualization, and machine learning.

RData is a file format used by R to store data objects. It is a binary format that is optimized for storage and retrieval of large datasets. RData files can contain a variety of data objects, including data frames, lists, matrices, and vectors.

Using RData.FromBinary

The RData.FromBinary function in Power Query allows users to read RData files into Power Query for further manipulation and analysis. The function takes a single argument, which is the path and filename of the RData file to be read.

Here is an example of how to use RData.FromBinary to read an RData file into Power Query:

```
let
  Source = RData.FromBinary("C:Datamydata.RData")
in
  Source
```

This code will read the file "mydata.RData" located in the "C:Data" folder into Power Query.

The M Code Behind RData.FromBinary

Behind the scenes, the RData.FromBinary function uses M code to read the binary data from the RData file and convert it into a Power Query table. Here is the M code that is generated when the RData.FromBinary function is used:

```
let
  Source = Binary.Buffer(File.Contents("C:Datamydata.RData")),
  BinaryData = Source,
Record.AddField
```

What is Record.AddField?

Record.AddField is a Power Query M function that allows you to add a new field to a record. A record is a collection of named values, similar to a row in a table. The new field can be based on an existing field or calculated using a formula. The syntax for the Record.AddField function is as follows:

Record.AddField(record as record, field as text, value as any) as record

Where:

- record: The record to which the new field is added.
- field: The name of the new field.
- value: The value of the new field.

Creating a Simple Record

Before we dive into the details of the Record.AddField function, let's create a simple record to work with. In Power Query, a record is enclosed in curly braces {} and consists of a list of key-value pairs, where the key is a field name and the value is the field value. For example, the following code creates a record with three fields:

```
let
  myRecord = [
    FirstName = “John”,
    LastName = “Smith”,
    Age = 30
  ]
in
  myRecord
```

Record.Combine

Understanding the Record.Combine Function

The Record.Combine function is used to combine two or more records into one. It takes two or more records as input and returns a new record that contains all the fields from the input records. The function combines records based on their field names. If there are duplicate fields, the function keeps the value from the last record.

The syntax for the Record.Combine function is as follows:

```
Record.Combine(record1 as record, record2 as record, ...) as record
```

Where `record1`, `record2`, and so on, are the records that you want to combine.

How Record.Combine Works

To understand how Record.Combine works, let's take a look at an example. Suppose we have the following two records:

```
Record1:
{
  “Name”: “John”,
  “Age”: 30,
  “City”: “New York”
}
Record2:
{
  “Name”: “Jane”,
  “Gender”: “Female”
}
```

```
Record.Field
```

What is the Record.Field function?

The Record.Field function is used to extract a value from a record in Power Query. A record is a data structure that contains one or more fields, where each field has a name and a value. For example, a record might contain fields for the first name, last name, and email address of a person. The Record.Field function takes two arguments: the record and the name of the field to extract. For example, the following code extracts the value of the “first name” field from a record:

```
let
  source = #table(
    {“first name”, “last name”, “email”},
    {{“John”, “Doe”, “john.doe@example.com”}}),
  record = Table.First(source),
  first_name = Record.Field(record, “first name”)
in
  first_name
```

In this example, we create a table with three columns: "first name", "last name", and "email". We then create a record from the first row of the table using the Table.First function. Finally, we extract the value of the "first name" field using the Record.Field function and assign it to the variable "first_name".

Understanding the M code behind Record.Field

To understand the M code behind the Record.Field function, we need to understand how records are represented in Power Query. In Power Query, records are represented as lists of field names and corresponding values. For example, the record {first_name = "John", last_name = "Doe", email = "john.doe@example.com"} would be represented as the following list:

```
{
Record.FieldCount
```

What is M?

M is the programming language used in Power Query. It's a functional language, meaning that it emphasizes the evaluation of expressions and avoids changing state or mutating data. This makes it well-suited for working with large datasets, as it allows for efficient processing and manipulation of data.

Understanding Records

Before we dive into the specifics of Record.FieldCount, it's important to understand what a record is. In M, a record is a data structure that consists of a set of named fields. Each field has a name and a value, and records can be nested inside other records.

For example, consider the following record:

```
[
  CustomerName = "John Smith",
  Orders = [
    OrderID = 1,
    Product = "Widget",
    Quantity = 5
  ]
]
```

This record has two fields: CustomerName and Orders. The Orders field is itself a record with three fields: OrderID, Product, and Quantity.

Using Record.FieldCount

Now that we understand what a record is, let's look at how Record.FieldCount works. This function takes a record as its argument and returns the number of fields in that record.

For example, suppose we have the following record:

Record.FieldNames

Understanding Records in Power Query

Before we dive into the `Record.FieldNames` function, let's first understand what records are in Power Query. A record is a collection of fields that represent a single data item. Each field in a record contains a value that corresponds to a specific property. For example, a record can represent a customer and contain fields such as 'Name', 'Address', 'Phone Number', and 'Email Address'.

In Power Query, records are represented using curly braces {} and are separated by commas. The fields within a record are separated by a semicolon (;) and are represented as field name/value pairs. Here is an example of a record:

{ Name = "John Smith"; Age = 35; Occupation = "Engineer" }

The Record.FieldNames Function

The `Record.FieldNames` function is a built-in function in Power Query that returns a list of all the field names in a given record. The function takes a record as its argument and returns a list of text values. Here is the syntax of the `Record.FieldNames` function:

Record.FieldNames(record as record) as list

The `record` argument is the record for which to return the field names. The function returns a list of text values representing the field names in the record.

Here is an example of using the `Record.FieldNames` function to return the field names of a record:

```
let
  customer = { Name = "John Smith"; Age = 35; Occupation = "Engineer" },
  fieldNames = Record.FieldNames(customer)
```

Record.FieldOrDefault

What is Record.FieldOrDefault?

Record.FieldOrDefault is a function in the M language that allows you to retrieve a value from a record using a specified field name, but with the added benefit of providing a default value in case the specified field does not exist. This is useful when working with data that may not be fully standardized, where some records may have missing fields. Instead of having to handle these missing fields manually, you can use Record.FieldOrDefault to specify a default value, which will be used if the field is missing.

How does Record.FieldOrDefault work?

The syntax for Record.FieldOrDefault is as follows:

```
Record.FieldOrDefault(record as record, field as text, optional default as any) as any
```

The first parameter, record, specifies the record from which to retrieve the field value. The second parameter, field, specifies the name of the field to retrieve. The optional third parameter, default, specifies the default value to use if the field is missing. If the default parameter is not specified, the function will return null if the field does not exist.

Here is an example of how Record.FieldOrDefault can be used to retrieve a value from a record:

```
let
  myRecord = [Name=”John”, Age=30],
  nameOrDefault = Record.FieldOrDefault(myRecord, “Name”, “Unknown”),
  addressOrDefault = Record.FieldOrDefault(myRecord, “Address”, “Unknown”)
in
  [NameOrDefault=nameOrDefault, AddressOrDefault=addressOrDefault]
```

In this example, we create a record called myRecord with two fields: Name and Age. We then use Record.FieldOrDefault to retrieve the

Record.FieldValues

What is Record.FieldValues?

Record.FieldValues is a M function that is used to extract all the values from a record and return them as a list. This function is especially useful when working with nested records, where you need to extract values from a record that is inside another record.

Syntax

The basic syntax of the Record.FieldValues function is as follows:

```
Record.FieldValues(record as record) as list
```

The function takes a record as its input and returns a list of all the values in the record.

How to use Record.FieldValues

To use the Record.FieldValues function, you need to provide a record as input. The record can be created using the curly brace notation, as shown below:

```
let
  myRecord = [name = “John”, age = 30, address = [street = “Main Street”, city = “New York”]]
in
  myRecord
```

This will create a record with three fields - name, age, and address. The address field is itself a record with two fields - street and city.

To extract all the values from this record, you can use the Record.FieldValues function as follows:

```
let
Record.FromList
```

What is `Record.FromList`?

First of all, let's have a brief introduction to `Record.FromList`. This function takes a list of key-value pairs and returns a record. The list should be in the format of `{{key1, value1}, {key2, value2}, ...}`. For example, `Record.FromList({{"Name", "John"}, {"Age", 30}})` will create a record with two fields, "Name" and "Age", and their corresponding values, "John" and 30.

The M Code Behind `Record.FromList`

The M code of `Record.FromList` is quite simple. Here is the code:

```
Record.FromList = (list as list) =>
  let
    keys = List.Transform(list, each _{0}),
    values = List.Transform(list, each _{1}),
    result = [Record = Record.FromColumns({values}, keys)]
  in
    result[Record];
```

As you can see, the code takes a list as its input parameter, and it consists of three main steps.

Step 1: Extract keys

The first step is to extract all the keys from the input list. This is done by using the `List.Transform` function to iterate over each item in the list and return its first element (which is the key). The result is a list of keys.

```
keys = List.Transform(list, each _{0}),
```

Step 2: Extract values

Record.FromTable

Understanding the Record.FromTable Function

The Record.FromTable function is useful for creating a record from a table in Power Query. The function takes a table as an argument and returns a record that includes all the column names and their respective values. The function works by creating a new record for each row in the table and combining the column names and values into a single record.

Here's an example of how the Record.FromTable function works:

```
let
  Source = Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WSkksKkwrVAgA", BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Column1 = _t]),
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"Column1", Int64.Type}}),
  #"Grouped Rows" = Table.Group(#"Changed Type", {"Column1"}, {{"Count", each Table.RowCount(_), type number}}),
  #"Converted to Table" = Table.FromList(#"Grouped Rows"[Count], Splitter.SplitByNothing(), null, null, ExtraValues.Error),
  #"Renamed Columns" = Table.RenameColumns(#"Converted to Table",{{"Column1", "Count"}}),
  #"Added Custom" = Record.FromTable(#"Renamed Columns")
in
  #"Added Custom"
```

In this example, we start by creating a table from some sample data. We then group the rows in the table by the values in the "Column1" column and count the number of rows in each group. We then convert the resulting table into a list and create a new column called "Count". Finally, we use the Record.FromTable function to create a record from the table.

The resulting record will include all the column names and their respective values, which in this case will be the values from the "Count" column.

The M Code Behind Record.FromTable

To understand how the Record.FromTable function works, we need to take a closer look at the M code behind the function.

Here's the M code for the Record.FromTable function:

Record.HasFields

Understanding Records in Power Query

Before diving into the M code behind the Record.HasFields function, it’s important to understand what records are in Power Query. Records are a type of data structure that can contain multiple key-value pairs. Each key-value pair is called a field. Records are used to store complex data structures, such as JSON, and are a fundamental part of Power Query.

The M Code Behind Record.HasFields

The M code behind the Record.HasFields function is relatively simple. The function takes two arguments, a record and a list of field names. It returns a Boolean value indicating whether the record contains all of the specified fields. Here is the M code for the Record.HasFields function:

```
Record.HasFields = (record, fieldNames) => List.ForAll(fieldNames, each Record.HasFields(record, _))
```

The function uses the List.ForAll function to iterate over the list of field names, checking if each field exists in the record. If all of the fields exist, the function returns true. If any of the fields are missing, the function returns false.

Using Record.HasFields in Power Query

Now that we understand the M code behind the Record.HasFields function, let's look at some examples of how it can be used in Power Query.

Example 1: Checking if a Record has a Field

Suppose we have a record that contains information about a customer, including their name, address, and phone number. We want to check if the record contains a field called "email". Here is the M code we can use:

```
let
  customer = [Name = “John Smith”, Address = “123 Main St”, Phone = “555-555-5555”],
  hasEmail = Record.HasFields(customer, {“email”})
in
Record.RemoveFields
```

Understanding the Record.RemoveFields Function

The Record.RemoveFields function is a powerful tool within Power Query that allows you to selectively remove fields from a table. This function can be especially useful when working with large datasets that contain many unnecessary or irrelevant fields.

To use the Record.RemoveFields function, simply select the table that you want to remove fields from, and then specify the specific fields that you want to remove. The function will then automatically remove those fields from the table, leaving you with a cleaner and more streamlined dataset.

The M Code Behind Record.RemoveFields

To better understand how the Record.RemoveFields function works, let’s take a closer look at the M code behind this powerful function. The M code for Record.RemoveFields is as follows:

```
(Table as table, Optional fields as any) as table =>
  let
    fieldsToRemove = if fields = null then {} else List.Buffer(fields),
    keepColumns = List.RemoveMatchingItems(Table.ColumnNames(Table), fieldsToRemove),
    result = Table.SelectColumns(Table, keepColumns)
  in
    result
```

Let's break down each section of this code to better understand how it works.

The Input Parameters

The Record.RemoveFields function takes two input parameters: the table that you want to remove fields from, and the specific fields that you want to remove (which are optional).

```
(Table as table, Optional fields as any) as table =>
```

Record.RenameFields

What is a Record in Power Query?

In Power Query, a record is a data type that contains one or more fields. Each field in a record has a name and a value. For example, consider the following record:

[CustomerID = 1, FirstName = “John”, LastName = “Doe”, Age = 30]

This record contains four fields: CustomerID, FirstName, LastName, and Age. The value of the CustomerID field is 1, the value of the FirstName field is "John", the value of the LastName field is "Doe", and the value of the Age field is 30.

What is the Record.RenameFields Function?

The Record.RenameFields function is used to rename one or more fields in a record. The syntax of the function is as follows:

Record.RenameFields(record as record, renames as list, optional missingField as nullable any) as record

The function takes three arguments:

- record: The record that contains the fields to be renamed.
- renames: A list of pairs that specify the old and new names of the fields to be renamed.
- missingField: (Optional) The value to return if a renamed field is missing from the input record. By default, the function returns an error if a renamed field is missing.

How to Use the Record.RenameFields Function

To use the Record.RenameFields function, you first need to create a record that contains the fields to be renamed. You can create a record using the Record.FromList or Record.FromFields function. For example, consider the following record:

Record.ReorderFields

Understanding `Record.ReorderFields`

The `Record.ReorderFields` function is used to reorder the fields in a record. In Power Query, a record is a structured data type that consists of one or more named fields. Each field has a name and a value, and the value can be of any data type.

The `Record.ReorderFields` function takes two arguments: a record, and a list of field names in the desired order. It returns a new record with the same fields as the original record, but with the fields reordered according to the specified list.

Here's an example of how to use `Record.ReorderFields`:

```
let
  originalRecord = [Name="John", Age=30, Occupation="Engineer"],
  newRecord = Record.ReorderFields(originalRecord, {"Age", "Name", "Occupation"})
in
  newRecord
```

In this example, we have a record called `originalRecord` with three fields: `Name`, `Age`, and `Occupation`. We then use `Record.ReorderFields` to create a new record called `newRecord` with the same fields, but in a different order: `Age`, `Name`, and `Occupation`.

The M Code Behind `Record.ReorderFields`

To understand the M code behind `Record.ReorderFields`, it's helpful to first understand how records are represented in M. In M, a record is represented as a list of key-value pairs, where each key is a field name and each value is the corresponding field value.

For example, the following M expression defines a record with three fields:

```
[
  Name = "John",
  Age = 30,
```

Record.SelectFields

Understanding Records in Power Query

Before we can delve into the Record.SelectFields function, it is important to understand what a record is in Power Query. A record is a collection of named values, similar to a row in a database table. Each value is assigned a unique name or key, and can contain any data type, including numbers, text, and arrays.

For example, consider the following record:

```
[Name="John", Age=35, Email="john@example.com"]
```

This record contains three named values: Name, Age, and Email. The value of each key can be accessed by using dot notation, like so:

```
Record.Name // Returns "John"
Record.Age // Returns 35
Record.Email // Returns "john@example.com"
```

The Record.SelectFields Function

The Record.SelectFields function is a built-in function in Power Query that allows you to select a subset of columns from a record. This function takes two arguments: the record to be filtered, and a list of keys to include in the filtered record.

Here is the basic syntax for the Record.SelectFields function:

```
Record.SelectFields(Record, {"Key1", "Key2", ..., "KeyN"})
```

Record.ToList

What is a Record in Power Query M?

In Power Query M, a record is a data type that consists of a set of named values. Each value in a record is associated with a unique name, and the values can be of different data types such as text, numbers, dates, and so on. Records can be created using the Record.FieldValues function or by using the curly brace notation.

What is the Record.ToList Function?

The Record.ToList function is used to convert a record into a list of values. This function takes a record as its argument and returns a list of values in the order they were defined in the record. The values in the list can be of any data type, and they are not associated with any names.

How to Use the Record.ToList Function?

To use the Record.ToList function in Power Query M, you need to provide a record as its argument. The following code snippet shows how to use this function:

```
let
  myRecord = [Name = “John”, Age = 30, Sex = “Male”],
  myList = Record.ToList(myRecord)
in
  myList
```

In this code snippet, we define a record called "myRecord" with three fields: Name, Age, and Sex. We then use the Record.ToList function to convert this record into a list of values and assign it to a variable called "myList". Finally, we return the "myList" variable. The output of this code snippet is as follows:

```
{“John”, 30, “Male”}
```

Record.ToTable

Understanding the Record Data Type

Before we dive into the M code behind Record.ToTable, it's important to understand the record data type. In Power Query, a record is a collection of fields and values that are stored as key-value pairs. For example, a record might contain fields such as Name, Age, and Gender, each with a corresponding value. Records are often used to represent complex data structures, such as JSON objects.

The Basic Syntax of Record.ToTable

The basic syntax of Record.ToTable is as follows:

```
Record.ToTable(record as record)
```

This function takes a single argument, which is the record that you want to convert into a table. The result is a table that contains two columns: one for the field names and one for the corresponding values. Each row in the table represents a single field-value pair from the original record.

Converting a Record into a Table

To convert a record into a table using Record.ToTable, you simply need to provide the function with the record that you want to convert. For example, to convert a record called CustomerInfo into a table, you would use the following M code:

```
Record.ToTable(CustomerInfo)
```

This would result in a table that contains two columns: one for the field names and one for the corresponding values. Each row in the table represents a single field-value pair from the original record.

Renaming the Table Columns

By default, the columns in the resulting table created by Record.ToTable are named "Name" and "Value". However, you can rename these columns to better reflect the contents of the record. To do this, you can use the optional second argument of Record.ToTable, Record.TransformFields

Understanding TransformFields

TransformFields is a function in the Power Query M language that allows users to modify the contents of a table by transforming individual fields. This function is particularly useful when working with large data sets that require complex transformations.

To use TransformFields, users must provide two arguments: the table to be transformed, and a function that defines the transformation to be applied to each field. The function takes a single argument, which represents the current value of the field. The function should return the new value of the field.

TransformFields then applies the specified function to each field in the table, returning a new table with the transformed data.

Key Features of TransformFields

The TransformFields function offers a number of key features that make it a powerful tool for data transformation:

Customizable Transformations

One of the most powerful features of the TransformFields function is its ability to apply custom transformations to table data. Users can define their own M code to transform each field in the table, allowing for a high degree of flexibility and customization.

Automatic Detection of Data Types

The TransformFields function automatically detects the data type of each field in the table, allowing users to apply appropriate transformations to each field. This feature makes it easy to work with large and complex data sets that contain multiple data types.

Support for Nested Tables

TransformFields also supports nested tables, enabling users to apply transformations to nested fields and sub-tables within the main table. This feature is particularly useful when working with complex data structures that contain multiple layers of nested data.

Use Cases for TransformFields

TransformFields is a versatile tool that can be used in a variety of data transformation scenarios. Some common use cases include:

Data Cleaning and Standardization

TransformFields can be used to clean and standardize data by applying custom transformations to individual fields. For example, users could use TransformFields to remove leading or trailing whitespace from text fields, or to convert dates to a standardized format.

Data Aggregation and Summarization

TransformFields can also be used to aggregate and summarize data by applying custom functions to fields. For example, users could use TransformFields to calculate the average or maximum value of a field across multiple rows.

Data Filtering and Sorting

Replacer.ReplaceText

The Replacer.ReplaceText function in Power Query M Language is used to replace a specific text string with another text string. This function is very useful for cleaning up data, especially when working with large datasets.
The syntax for the Replacer.ReplaceText function is as follows:

= Table.ReplaceValue(table as table, old_value as any, new_value as any, replacer as function, column_names as any, optional options as nullable record)

In this syntax, the `table` argument is the input table that needs to be transformed. The `old_value` argument is the text string that needs to be replaced. The `new_value` argument is the text string that will replace the old value.
The `replacer` argument is a function that is used to replace the text string. This function takes two arguments: the old value and the new value. The `column_names` argument specifies which columns in the table will be searched for the old value. The `options` argument is optional and is used to specify additional options for the function.
How to Use the Replacer.ReplaceText Function
Here is an example of how to use the Replacer.ReplaceText function in Power Query:
Suppose we have a table named `Sales` that contains sales data for a company. The table has two columns: `Product` and `SalesAmount`. The `Product` column contains the names of the products that were sold, and the `SalesAmount` column contains the amount of sales for each product.
We want to replace the name of one of the products, say "Product A", with "Product B". To do this, we need to use the Replacer.ReplaceText function. Here is the M code to achieve this:

```
let
  Source = Sales,
  ReplaceProdA = Table.ReplaceValue(Source,”Product A”,”Product B”,Replacer.ReplaceText,{“Product”})
in
Replacer.ReplaceValue
```

What is the Replacer.ReplaceValue M Function?

The Replacer.ReplaceValue M function is used to replace specific values in a column with new values. It is a part of the Replacer functions in Power Query. The function syntax is as follows:

```
Replacer.ReplaceValue(source as any, oldValue as any, newValue as any, Replacer.ReplaceValueFlags) as any
```

Here, the parameters are:

- source: The column or table that you want to replace the values in.
- oldValue: The value that you want to replace in the column or table.
- newValue: The value that you want to replace the old value with.
- Replacer.ReplaceValueFlags: This optional parameter allows you to set flags for the replacement operation, such as ignoring case sensitivity.

How Does the Replacer.ReplaceValue M Function Work?

The Replacer.ReplaceValue M function uses a combination of List.ReplaceMatchingItems and List.Transform functions internally to replace the values in a column or table. When you call the Replacer.ReplaceValue function, it first creates a list of all the matching items in the source column or table using the List.ReplaceMatchingItems function. It then transforms the list using the List.Transform function to replace each matching item with the new value.

Examples of Using the Replacer.ReplaceValue M Function

Let's take a look at some examples of using the Replacer.ReplaceValue M function in Power Query.

Example 1: Replacing a Single Value in a Column

Suppose we have a table named "Products" with a column named "Category". We want to replace all the values in the "Category" column that say "Electronics" with "Gadgets". Here is the M code to achieve this:

```
let
RowExpression.Column
```

What is RowExpression.Column Function?

The RowExpression.Column function is a part of the M language and is used to extract the value of a column from a table. The function takes two arguments – the first argument is the table, and the second argument is the name of the column. The function returns a list of values from the specified column.

The M Code Behind RowExpression.Column Function

The RowExpression.Column function is implemented in the M language as follows:

```
(table as table, column as text) as list =>
  Table.Column(table, column)
```

The function takes two parameters - table and column. The first parameter is of the type “table,” which is the table from which you want to extract the column. The second parameter is of the type “text,” which is the name of the column you want to extract.

The function returns a list of values from the specified column using the Table.Column function. The Table.Column function takes two arguments - the first argument is the table, and the second argument is the name of the column. The function returns a list of values from the specified column.

How to Use RowExpression.Column Function?

The RowExpression.Column function can be used to extract a single column or multiple columns from a table. Here are a few examples:

Extracting a Single Column

To extract a single column from a table, you can use the following syntax:

```
= Table.AddColumn(#”Previous Step”, “Column Name”, each RowExpression.Column([Table Name], “Column Name”))
```

In this syntax, the “Previous Step” refers to the previous step in the Power Query Editor. The “Column Name” refers to the name of the

RowExpression.From

What is Power Query M?

Power Query M is a programming language used in Power Query, a data transformation and analysis tool used in Microsoft Excel and Power BI. It is a functional language that is used to perform data transformations and create custom functions.

RowExpression.From

The RowExpression.From function is used to create a new row expression. It takes two arguments: a list of column names and a list of expressions. The column names are used to define the columns of the new row expression, and the expressions are used to define the values of the columns.

Here is an example of the RowExpression.From function:

```
RowExpression.From({“Name”, “Age”, “Gender”}, {“John Doe”, 30, “Male”})
```

This function creates a new row expression with three columns: Name, Age, and Gender. The values of these columns are "John Doe", 30, and "Male", respectively.

Understanding the M Code

The M code behind the RowExpression.From function is relatively simple. Here is the M code for the function:

```
let
  CreateRow = Record.FromList,
  Result = CreateRow(columns, values)
in
  Result
```

The M code defines two variables: CreateRow and Result. The CreateRow variable is used to create a new record from a list, and the

RowExpression.Row

What is the RowExpression.Row function?

The RowExpression.Row function is a part of the Power Query M language, which is used to manipulate data in Power Query. The function is used to create a new row in a table by specifying the values for each column in the row. Here is the syntax of the RowExpression.Row function:

```
RowExpression.Row(column1, column2, ..., columnN)
```

Where column1, column2, ..., columnN are the values for each column in the row.

How does the RowExpression.Row function work?

The RowExpression.Row function works by creating a new record with the specified column values. In Power Query, a table is represented as a list of records, where each record represents a row in the table. A record is a collection of key-value pairs, where the key is the name of the column and the value is the value for that column in the row.

When you call the RowExpression.Row function, it creates a new record with the specified column values and appends it to the end of the table. Here is an example of how to use the RowExpression.Row function:

```
let
  Source = Table.FromRecords({
    [ID = 1, Name = “John”, Age = 30],
    [ID = 2, Name = “Jane”, Age = 25]
  }),
  NewRow = RowExpression.Row(3, “Bob”, 40),
  Result = Table.InsertRows(Source, Table.RowCount(Source), {NewRow})
in
  Result
```

Salesforce.Data

What is M Code?

M code is the language used by the Power Query formula language, which is used to transform and manipulate data within the Power Query Editor. It’s a functional language that allows you to perform a wide range of data transformation tasks, from basic filtering and sorting to complex data modeling and analysis.

In the case of Salesforce.Data, the M code is used to establish a connection to the Salesforce API, retrieve data from Salesforce objects, and perform any necessary transformations on that data. Let’s take a closer look at the M code behind this powerful function.

Connecting to the Salesforce API

The first step in using Salesforce.Data is to establish a connection to the Salesforce API. This is accomplished using the following M code:

```
let
  Source = Salesforce.Data(“https://login.salesforce.com”, [ApiVersion=51]),
  #”Authenticated” = Source{[Name=”Authenticated”]}[Data]
in
  #”Authenticated”
```

This code creates a new query using the Salesforce.Data function, which takes two arguments: the URL of the Salesforce login page and the API version to be used (in this case, version 51).

Once the query is created, the code uses the #"Authenticated" step to retrieve the authentication token from Salesforce. This token is required for all subsequent API requests and is stored in memory for the duration of the query.

Retrieving Data from Salesforce Objects

Once you've established a connection to the Salesforce API, you can start retrieving data from Salesforce objects. This is done using the following M code:

```
Salesforce.Reports
```

Power Query is a data connection technology that allows users to connect, transform, and manipulate data from various sources within Excel. The M function is a key component of Power Query, which provides a powerful and flexible way to transform data using custom code. In this article, we will take a closer look at the M code behind the Power Query M function Salesforce.Reports, which is used to connect to Salesforce reports within Excel.

Connecting to Salesforce Reports

The first step in using the Salesforce.Reports function is to connect to the Salesforce report that you want to retrieve data from. This can be done by providing the URL of the report to the function. The URL can be obtained by navigating to the report in Salesforce and copying the URL from the address bar.

```
Source = Salesforce.Reports("https://.my.salesforce.com/")
```

The part of the URL should be replaced with your actual Salesforce domain name, and the part should be replaced with the actual ID of the report that you want to retrieve data from.

Authentication

Before connecting to Salesforce reports, it is important to authenticate with Salesforce by providing your username and password. This can be done using the following code:

```
Source = Salesforce.Reports("https://.my.salesforce.com/", [ApiVersion = 28, Username = "", Password = ""])
```

The ApiVersion parameter is used to specify the version of the Salesforce API that you want to use. The default version is 28, but you can specify a different version if needed.

Transforming Data

Once you have connected to the Salesforce report and authenticated with Salesforce, you can use the M code to transform the data in

SapBusinessWarehouse.Cubes

Understanding SapBusinessWarehouse.Cubes

Before we dive into the M code, it's important to understand what SapBusinessWarehouse.Cubes is and what it does. This function is used in Power Query, which is a data connection and transformation tool that allows users to connect to various data sources, including SAP Business Warehouse (BW). In particular, SapBusinessWarehouse.Cubes is used to retrieve data from BW cubes.

A BW cube is essentially a data structure that stores multidimensional data in a way that is optimized for querying. By using SapBusinessWarehouse.Cubes in Power Query, users can easily access data from these cubes and use it in their analysis.

The M Code Behind SapBusinessWarehouse.Cubes

The M code behind SapBusinessWarehouse.Cubes is what makes it possible to retrieve data from BW cubes. When using this function, the M code is automatically generated by Power Query. However, it's still helpful to have an understanding of what the code is doing and how it works.

At a high level, the M code for SapBusinessWarehouse.Cubes is responsible for connecting to the BW system, retrieving data from the specified cube, and transforming that data into a format that can be used in Power Query. Let's take a closer look at each of these steps.

Connecting to the BW System

To connect to the BW system, the M code needs to know the system's hostname, client ID, system number, and other connection details. These values are typically provided by the user when configuring the connection.

Once the connection details are known, the M code uses the SAP NetWeaver RFC SDK to establish a connection to the BW system. This SDK provides a set of libraries and APIs that allow programs to communicate with SAP systems.

Retrieving Data from the Cube

After connecting to the BW system, the M code needs to retrieve data from the specified cube. To do this, it uses the SAP Business Warehouse Provider for Power Query. This provider is essentially a set of functions that allow Power Query to interact with BW cubes.

SapBusinessWarehouse.Cubes specifically uses the function BW.Cube, which takes the name of the cube and other parameters as input. This function returns a table that contains the data from the cube.

Transforming the Data

Once the data has been retrieved from the cube, the M code needs to transform it into a format that can be used in Power Query. This typically involves converting data types, renaming columns, and performing other data cleaning operations.

The specific transformations that are applied will depend on the needs of the user and the data being retrieved. However, Power Query provides a wide range of tools and functions that make it easy to perform these transformations.

SapHana.Database

What is Power Query?

Power Query is a data transformation and cleansing tool that allows you to extract data from various sources, clean and transform it, and load it into a destination. Power Query is available as an add-in for Excel, as a standalone tool in Power BI, and as a built-in tool in SQL Server Integration Services (SSIS).

Power Query has a rich set of functions that allow you to perform various data transformations such as filtering, sorting, grouping, merging, pivoting, splitting, and unpivoting. The M function is one of the most powerful functions in Power Query that allows you to write custom code to manipulate data.

What is the SapHana.Database function?

The SapHana.Database function is a Power Query M function that allows you to extract data from SAP HANA databases. SAP HANA is an in-memory database that is used to store and retrieve large amounts of data quickly. The SapHana.Database function allows you to connect to a SAP HANA database, specify a SQL statement to extract data, and load the data into Power Query.

To use the SapHana.Database function, you need to provide the following parameters:

– Server: The name of the SAP HANA server.

– Database: The name of the SAP HANA database.

– Query: The SQL statement to extract data.

– Options: Any additional options such as user ID and password.

The SapHana.Database function returns a table that contains the data extracted from the SAP HANA database. You can then use the rich set of Power Query functions to transform the data as needed.

The M Code Behind the SapHana.Database Function

The M code behind the SapHana.Database function is a bit complex, but it is well documented and easy to understand. Here is the M code behind the SapHana.Database function:

```
let
 Source = (server as text, database as text, query as text, options as record) =>
  let
   url = “jdbc:hana://” & server & “:3” & options[Port] & “;databaseName=” & database,
SharePoint.Contents
```

To fully understand the SharePoint.Contents function, it is important to take a deeper look into the M code that powers this function. In this article, we will explore the M code behind the SharePoint.Contents function, as well as its various parameters and options.

Understanding the SharePoint.Contents Function

The SharePoint.Contents function is used to access data from SharePoint lists, folders, and files. It is part of the Power Query M language, which is used to extract and transform data from various sources. The function takes several parameters, including the URL of the SharePoint site, the relative URL of the file or folder, and the authentication method used to access the data.

The function returns a table containing the data from the specified SharePoint location. This table can then be further transformed and analyzed using Power Query's other features.

The M Code Behind SharePoint.Contents

The M code behind the SharePoint.Contents function is responsible for accessing and retrieving the data from SharePoint. The code uses several built-in functions and operators to perform this task.

Here is an example of the M code used to retrieve data from a SharePoint list:

```
let
  Source = SharePoint.Contents("https://contoso.sharepoint.com/sites/test", [ApiVersion = 15]),
  List = Source{[Name="TestList"]}[Items]
in
  List
```

This code uses the SharePoint.Contents function to connect to the specified SharePoint site and retrieve data from the "TestList" list. The [ApiVersion = 15] parameter specifies the version of the SharePoint API to use.

The code then uses the curly brackets ({}) to specify the row or rows of data to retrieve from the list. In this case, we are retrieving all the items in the "TestList" list.

Parameters and Options

The SharePoint.Contents function takes several parameters and options that can be used to customize the data retrieval process.

SharePoint.Files

Overview of SharePoint.Files

SharePoint.Files is a Power Query M function that allows users to connect to SharePoint and retrieve files and folders from a SharePoint site. It is one of the many connectors available in Power Query, and it can be accessed via the “Get Data” button on the Power Query ribbon.

When the SharePoint.Files function is used, it prompts the user to enter the URL of the SharePoint site they want to connect to, as well as the credentials they want to use to authenticate to the site. Once connected, the user can then select the files and folders they want to retrieve from the site.

The M Code Behind SharePoint.Files

Behind the scenes, the SharePoint.Files function is powered by M code. M is a functional programming language used by Power Query to define data transformations. When the SharePoint.Files function is used, Power Query generates M code that connects to SharePoint and retrieves the desired files and folders.

Here is an example of the M code that is generated when the SharePoint.Files function is used:

```
let
  Source = SharePoint.Files(“https://contoso.sharepoint.com/sites/marketing”, [ApiVersion = 15]),
  #”Filtered Rows” = Table.SelectRows(Source, each ([Folder Path] = “/Shared Documents”)),
  #”Removed Other Columns” = Table.SelectColumns(#”Filtered Rows”,{“Name”, “Content”}),
  #”Sorted Rows” = Table.Sort(#”Removed Other Columns”,{{“Name”, Order.Ascending}})
in
  #”Sorted Rows”
```

Let's break down what is happening in this code:

- The "let" statement defines the beginning of the query. It is used to define variables that can be referenced later in the code.
- The "Source" variable is defined using the SharePoint.Files function. The URL of the SharePoint site is specified, as well as the API version to use (in this case, version 15).

SharePoint.Tables

Understanding the SharePoint.Tables Function

SharePoint is a popular collaboration platform used by many organizations to store and manage documents and data. Power Query provides a SharePoint.Tables function that allows users to connect to SharePoint lists and libraries and retrieve data.

The SharePoint.Tables function takes in two arguments: URL and options. The URL argument specifies the URL of the SharePoint site or subsite to connect to, while the options argument specifies additional options such as the name of the SharePoint list or library to retrieve data from.

Here's an example of the SharePoint.Tables function in action:

```
let
  Source = SharePoint.Tables("https://mysharepointsite.sharepoint.com/sites/MySite", [ApiVersion = 15]),
  #"My List" = Source{[Name="My List"]}[Items]
in
  #"My List"
```

In this example, we're connecting to a SharePoint site called "MySite" and retrieving data from a list called "My List".

The M Code Behind SharePoint.Tables

Behind the scenes, the SharePoint.Tables function uses M code to connect to the SharePoint site and retrieve data. Let's take a closer look at the M code behind the function.

First, the function uses the SharePoint.Contents function to retrieve the contents of the SharePoint site:

```
let
  Source = SharePoint.Contents("https://mysharepointsite.sharepoint.com/sites/MySite", [ApiVersion = 15])
in
  Source
```

Single.From

What is the Single.From Function?

The Single.From function is used to extract a single value from a table. It takes a table as input and returns the value in the first row and first column of the table. If the table is empty or has more than one row or column, an error is returned.

Here is an example of the Single.From function in action:

```
let
  Source = Table.FromRows({{1, “Red”}, {2, “Blue”}, {3, “Green”}}, {“ID”, “Color”}),
  Result = Single.From(Source)
in
  Result
```

In this example, we have a table with two columns: ID and Color. The Single.From function returns the value in the first row and first column of the table, which is 1.

The M Code Behind Single.From

The M code behind the Single.From function is relatively simple. Here is the code:

```
(table as table) as any =>
  let
    result = try table{0}[Column1] otherwise null
  in
    result
```

Let's break down this code and see what each part does.

Soda.Feed

In order to fully utilize the Soda.Feed function, it is important to understand the M code that underlies it. In this article, we will break down the M code behind Soda.Feed and explore how it works.

Understanding the Soda.Feed Function

Before we dive into the M code behind Soda.Feed, it is important to understand what the function does. Essentially, Soda.Feed allows users to connect to a SODA API endpoint and import data into Power Query. The function takes two arguments: the URL of the SODA endpoint, and an optional options record that allows users to customize the import process.

Here is an example of how the Soda.Feed function might be used:

```
let
  Source = Soda.Feed(“https://data.cityofchicago.org/resource/jcxq-k9xf.json”),
  #”Expanded Vehicles” = Table.ExpandRecordColumn(Source, “Vehicles”, {“id”, “make”, “model”, “year”}, {“Vehicles.id”, “Vehicles.make”, “Vehicles.model”, “Vehicles.year”})
in
  #”Expanded Vehicles”
```

In this example, we are importing data from the City of Chicago's SODA endpoint for vehicle crashes. We then expand the "Vehicles" column to create a more usable table.

Breaking Down the M Code

Now that we understand what the Soda.Feed function does, let's take a look at the M code that makes it work. Here is an example of the M code behind the function:

```
let
  Source = Json.Document(Web.Contents(url, options)),
  #”Converted to Table” = Record.ToTable(Source)
Splitter.SplitByNothing
```

In this article, we will take a closer look at the M code behind the Splitter.SplitByNothing function and explore its various use cases.

Understanding the Splitter.SplitByNothing Function

The Splitter.SplitByNothing function is used to split a text string into multiple columns based on a delimiter. However, unlike other split functions, it does not require a delimiter to perform the split. Instead, it simply splits the text string at every character boundary.

The syntax for the Splitter.SplitByNothing function is as follows:

```
Splitter.SplitByNothing(text as nullable text, optional options as nullable record) as list
```

The text parameter is the text string that you want to split, while the options parameter is an optional record that contains additional options for the split operation.

Using the Splitter.SplitByNothing Function

The Splitter.SplitByNothing function can be used in a variety of scenarios. Here are a few examples:

Splitting Text into Columns

One of the most common use cases for the Splitter.SplitByNothing function is to split text into columns. To do this, you simply select the column that contains the text you want to split, and then choose the Split Column option from the Transform tab.

In the Split Column dialog box, you can choose the Splitter.SplitByNothing function as the split delimiter. Power Query will automatically split the text into multiple columns based on every character boundary.

Removing Extra Spaces

Another useful application of the Splitter.SplitByNothing function is to remove extra spaces from text. To do this, you can use the Text.Replace function to replace all instances of multiple spaces with a single space, and then use the Splitter.SplitByNothing function to split the text into individual words.

For example, consider the following text string:

```
"  This  is  a  test  "
```

Splitter.SplitTextByAnyDelimiter

Understanding the Function

Before we dive into the M code, let's first understand how the function works. The Splitter.SplitTextByAnyDelimiter function takes two arguments: the text to be split and a list of delimiters. The function splits the text into columns based on any of the delimiters in the list. For example, if we have the text "apple,banana,orange" and we want to split it into columns using the delimiters "," and "-", we can use the following M code:

```
Splitter.SplitTextByAnyDelimiter("apple,banana-orange", {",","-"})
```

This will result in a table with three columns: "apple", "banana", and "orange".

Breaking Down the M Code

Now that we understand how the Splitter.SplitTextByAnyDelimiter function works, let's take a closer look at the M code behind it. The function is defined as follows:

```
( text as text, delimiters as list ) as list
```

The function takes two arguments: "text" and "delimiters". "Text" is the text to be split and "delimiters" is a list of delimiters to use when splitting the text.

The M code for the Splitter.SplitTextByAnyDelimiter function is as follows:

```
let
  splitText = (text as text, delimiters as list) as list =>
    List.Select(
```

Splitter.SplitTextByCharacterTransition

Splitter.SplitTextByCharacterTransition is a useful M function that splits a text string into multiple columns based on a specific character transition. In this article, we will explore the M code behind this function and how it can be used in Power Query.

Understanding Splitter.SplitTextByCharacterTransition

Splitter.SplitTextByCharacterTransition is a function that splits a text string into multiple columns based on a specific character transition. For example, if we have a text string “John,Smith,35” and we want to split it into three columns based on the comma character, we can use Splitter.SplitTextByCharacterTransition.

The function takes three arguments: the text string to split, the character to split by, and the number of columns to split into. It returns a table with the split columns.

Here is an example of how to use Splitter.SplitTextByCharacterTransition in Power Query:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMjQ1MlGyMlYwVjI0NjQ2MzGyNzIzMjQy1SjNyywqKMhVC
KpMqwJ1gZG5QpK7ODMFAA==", BinaryEncoding.Base64)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table
[Column1 = _t]),
  #"Split Column by Comma" = Table.SplitColumn(Source, "Column1", Splitter.SplitTextByCharacterTransition(",", (c) =>
List.Count(Text.PositionOfAny(c, ","))=1), {"Column1.1", "Column1.2", "Column1.3"})
in
  #"Split Column by Comma"
```

In this example, we have a table with a single column “Column1” that contains the text string “John,Smith,35”. We use Table.SplitColumn to split the column into three new columns based on the comma character. We pass in the function Splitter.SplitTextByCharacterTransition to specify the character to split by and the number of columns to split into.

The M Code Behind Splitter.SplitTextByCharacterTransition

The M code behind Splitter.SplitTextByCharacterTransition is relatively simple. Here is the code:

Splitter.SplitTextByDelimiter

In this article, we will take a closer look at the M code behind the Splitter.SplitTextByDelimiter function, how it works, and how you can use it to split text values in your data.

What is Splitter.SplitTextByDelimiter?

The Splitter.SplitTextByDelimiter function is a built-in function in Power Query that enables users to split a text value into multiple values based on a specified delimiter. The function returns a list of values resulting from the split operation.

The M Code Behind Splitter.SplitTextByDelimiter

The M code behind the Splitter.SplitTextByDelimiter function is relatively straightforward. The function takes three arguments: the text value to split, the delimiter character(s), and an optional parameter that specifies how many splits to perform.

Here is the M code for the Splitter.SplitTextByDelimiter function:

```
(text as text, delimiter as text, optional limit as nullable number) as list
```

The first argument, “text”, is the text value that you want to split. The second argument, “delimiter”, is the character or characters that you want to use as the delimiter. If you want to split the text value based on a comma delimiter, for example, you would specify “,” as the delimiter.

The third argument, “limit”, is an optional parameter that specifies how many splits to perform. If you omit this parameter, the function will split the text value based on all occurrences of the delimiter.

How to Use Splitter.SplitTextByDelimiter

To use the Splitter.SplitTextByDelimiter function, you first need to load your data into Power Query. Once your data is loaded, you can add a new column to your query that uses the Splitter.SplitTextByDelimiter function to split a text value.

Here is an example of how to use Splitter.SplitTextByDelimiter:

1. Load your data into Power Query.
2. Click on the “Add Column” tab in the Power Query Editor.
3. Click on “Custom Column” in the “Add Column” dropdown menu.
4. Enter a name for your new column.

Splitter.SplitTextByEachDelimiter

Introduction to Splitting Text

Splitting text is a common task in data analysis and reporting. It involves taking a string of text and dividing it into smaller pieces based on a specific character or sequence of characters called a delimiter. For example, consider the following string:

"John,Smith,25,123 Main St,Anytown,USA"

If we use the comma as the delimiter, we can split this string into separate columns for each piece of information:

First Name	Last Name	Age	Address	City	Country
John	Smith	25	123 Main St	Anytown	USA

Power Query makes this task easy with its Split Column feature, which allows us to split text by a delimiter and create new columns in the process.

The Splitter.SplitTextByEachDelimiter Function

Behind the scenes, Power Query uses a complex formula language called M to perform operations like splitting text. The Splitter.SplitTextByEachDelimiter function is one of the M functions that Power Query uses to split text based on multiple delimiters. The syntax of the Splitter.SplitTextByEachDelimiter function is as follows:

Splitter.SplitTextByEachDelimiter(text as text, delimiters as list, optional quoteStyle as nullable number, optional extraValues as nullable number) as list

Splitter.SplitTextByLengths

Understanding the Splitter.SplitTextByLengths function

The Splitter.SplitTextByLengths function is used to split text into multiple columns based on the lengths of the substrings. It takes two arguments – the first argument is the text to be split, and the second argument is a list of integers that represent the lengths of the substrings. The function then splits the text into columns based on the lengths specified in the second argument.

For example, consider the following text: “John Doe 123 Main St New York NY 10001”. Suppose we want to split this text into four columns – one for the first name, one for the last name, one for the address, and one for the city, state, and zip code. We can use the Splitter.SplitTextByLengths function as follows:

```
Splitter.SplitTextByLengths("John Doe 123 Main St New York NY 10001", {4, 3, 10, 18})
```

This will split the text into four columns with the following values:

Column 1	Column 2	Column 3	Column 4
———-	———-	————-	——————
John	Doe	123 Main St	New York NY 10001

The M code behind the Splitter.SplitTextByLengths function

The M code behind the Splitter.SplitTextByLengths function is relatively simple. It consists of a function that takes two arguments – the text to be split and the list of lengths. The function then uses the List.Zip function to combine the text with the lengths and create a list of tuples. Each tuple contains a substring of the text and its corresponding length.

The list of tuples is then passed to the List.Accumulate function, which iterates over each tuple and splits the text into columns based on the lengths specified in the tuple. The result is a list of lists, where each inner list represents a column of the split text.

Here is the M code for the Splitter.SplitTextByLengths function:

```
let
  SplitTextByLengths = (text as text, lengths as list) =>
Splitter.SplitTextByPositions
```

In this article, we will explore the M code behind the Splitter.SplitTextByPositions function and how it can be used in your data transformations. We will also provide some examples to demonstrate the function's capabilities.

Overview of the Splitter.SplitTextByPositions function

The Splitter.SplitTextByPositions function is used to split a text string into multiple columns by specifying the start and end positions of each column. This function takes two arguments:

– delimiter: The delimiter used to separate the columns. This can be any character or string.

– positions: A list of tuples that specifies the start and end positions of each column.

The function returns a table with the specified columns.

Examples

Let's take a look at some examples of how the Splitter.SplitTextByPositions function can be used.

Example 1: Splitting a text string into multiple columns

Suppose we have a text string that contains data in the format of "name, age, city". We want to split this text string into three separate columns for further analysis.

Here's the M code to achieve this:

```
let
  Source = "John, 25, New York",
  Split = Splitter.SplitTextByPositions(",", {{0,4}, {6,7}, {9,17}})
in
  Split
```

In this example, we used a comma delimiter and specified the start and end positions of each column in the positions argument. The resulting table contains three columns: name, age, and city.

Example 2: Extracting specific portions of a text string

Suppose we have a text string that contains data in the format of "product code-quantity-price". We want to extract the quantity column

Splitter.SplitTextByRanges

In this article, we will explore the M code behind the SplitTextByRanges function and how it can be used to extract and transform data.

What is the SplitTextByRanges Function?

The SplitTextByRanges function in Power Query is used to split a given text string into separate columns based on the specified ranges. This function takes two arguments – the text string to be split and a list of ranges that define the start and end positions of each column.

For example, consider the following text string: “John,Smith,30,New York”. We can use the SplitTextByRanges function to split this text into four separate columns – First Name, Last Name, Age, and City – using the following ranges:

– First Name: 0-3
– Last Name: 5-9
– Age: 10-11
– City: 12-19

Using these ranges, the SplitTextByRanges function will split the text string into four columns, as shown below:

First Name	Last Name	Age	City
————	————	—	————
John	Smith	30	New York

The M Code Behind the SplitTextByRanges Function

To understand the M code behind the SplitTextByRanges function, we first need to understand how ranges are defined in Power Query. A range is defined as a record with two fields – Start and Length. The Start field specifies the starting position of the range, and the Length field specifies the length of the range.

For example, the range {Start: 0, Length: 3} defines a range that starts at position 0 and has a length of 3.

The SplitTextByRanges function uses the List.Transform function to split the text string into separate columns based on the specified ranges. The List.Transform function applies a given function to each element in a list and returns a new list with the results.

In the case of the SplitTextByRanges function, the function applied to each range is the Text.Range function, which extracts a substring from a given text string based on the specified range.

The M code for the SplitTextByRanges function is shown below:

```
let
Splitter.SplitTextByRepeatedLengths
```

Understanding the Power Query M Language

Power Query is a data transformation and cleaning tool that is part of the Microsoft Power BI suite. Power Query uses the M language to transform and shape data. M is a functional programming language that is used to write custom functions and perform complex data transformations in Power Query.

In order to fully understand the code behind Splitter.SplitTextByRepeatedLengths, it is important to have a basic understanding of the M language. M is a case-sensitive language and is based on functional programming concepts such as immutability, recursion, and higher-order functions.

Using Splitter.SplitTextByRepeatedLengths

Splitter.SplitTextByRepeatedLengths is a function that splits text into equal-length chunks. This function takes two arguments: the text to be split and the length of each chunk. For example, if we wanted to split the word “hello” into two equal-length chunks, we would use the following code:

```
Splitter.SplitTextByRepeatedLengths("hello", 2)
```

This would return the following output:

```
{"he", "ll", "o"}
```

As you can see, the word “hello” has been split into three equal-length chunks of two characters each.

The M Code Behind Splitter.SplitTextByRepeatedLengths

Splitter.SplitTextByRepeatedLengths is a built-in function in Power Query, which means that the code behind it is not accessible to users. However, we can still gain a basic understanding of the code by examining the function’s behavior.

When Splitter.SplitTextByRepeatedLengths is called, the M engine first converts the input text into a list of characters. It then calculates

Splitter.SplitTextByWhitespace

What is Splitter.SplitTextByWhitespace?

The Splitter.SplitTextByWhitespace function is used to split a text string into a list of words. The function takes a single parameter, which is the text string to be split. The output of the function is a list of words, where each word is a separate element in the list.

The M Code Behind Splitter.SplitTextByWhitespace

The M code behind the Splitter.SplitTextByWhitespace function is relatively simple. The function is defined as follows:

```
Splitter.SplitTextByWhitespace = (text as text) as list => List.Split(Text.Trim(text), {" "}, Splitter.SplitTextOptions.TrimEmpty)
```

The function takes a single parameter, which is a text string. The text string is first trimmed using the Text.Trim function to remove any leading or trailing whitespace. The trimmed text string is then split using a delimiter of ” ” (a single space character). The Splitter.SplitTextOptions.TrimEmpty option is used to remove any empty elements from the resulting list.

Using Splitter.SplitTextByWhitespace

The Splitter.SplitTextByWhitespace function can be used in a variety of data manipulation scenarios. Here are a few examples of how the function can be used:

Example 1: Splitting a Name into First and Last

Suppose we have a table of names in the format “First Last”. We can use the Splitter.SplitTextByWhitespace function to split the names into separate columns for first and last name. Here is an example M code:

```
let
  Source = Table.FromRows({{"John Smith"}, {"Jane Doe"}}, {"Name"}),
  SplitNames = Table.AddColumn(Source, "First", each List.First(Splitter.SplitTextByWhitespace([Name]))),
  SplitNames = Table.AddColumn(SplitNames, "Last", each List.Last(Splitter.SplitTextByWhitespace([Name])))
in
  SplitNames
```

Sql.Database

Connecting to a SQL Server Database

Before we dive into the M code behind the Sql.Database function, let's first take a look at how to connect to a SQL Server database using Power Query.

1. Open Power Query and click on the "From Database" dropdown in the "Get Data" section of the ribbon.
2. Select "From SQL Server Database" and enter the server name and database name that you want to connect to.
3. If you need to specify any additional options, such as authentication or a specific query, you can do so in the "Advanced options" section.
4. Click on "Connect" and Power Query will connect to your SQL Server database and retrieve a list of tables.

Once you have connected to your SQL Server database, you can start using the Sql.Database function to access data.

The M Code Behind the Sql.Database Function

The Sql.Database function is a part of the M language used in Power Query. It allows you to connect to a SQL Server database and retrieve data using a SQL query. Here is the basic syntax of the Sql.Database function:

```
Sql.Database(server as text, database as text, optional options as nullable record) as table
```

Let's break down each part of the function:

– `server`: The name of the SQL Server instance that you want to connect to.
– `database`: The name of the database that you want to connect to.
– `options`: An optional record that allows you to specify additional options, such as authentication or a specific query.
– `table`: The table that contains the data retrieved from the SQL Server database.

Here is an example of how to use the Sql.Database function to retrieve data from a SQL Server database:

```
let
  Source = Sql.Database("localhost", "AdventureWorks2019"),
Sql.Databases
```

What is Sql.Databases?

The Sql.Databases function is a M function that returns a table with metadata about the databases available on a SQL Server instance. It takes two arguments: the server name and an optional connection string.

Here is an example of how to use the Sql.Databases function:

```
let
  server = "localhost",
  databaseList = Sql.Databases(server)
in
  databaseList
```

This code will return a table with metadata about the databases available on the “localhost” SQL Server instance.

Breaking Down the M Code

Let’s take a closer look at the M code behind the Sql.Databases function. Here is the M code:

```
(SqlServerDataSource(server, [HierarchicalNavigation=true])
 {[Name="Databases"]}
)[Data]
```

This code is a bit complex, but we can break it down into three parts:

Part 1: SqlServerDataSource

The SqlServerDataSource function is a M function that establishes a connection to a SQL Server. It takes two arguments: the server name and an optional connection string.

SqlExpression.SchemaFrom

Understanding the SqlExpression.SchemaFrom Function

The SqlExpression.SchemaFrom function is used to retrieve metadata about a SQL database table. This metadata includes information about the columns in the table, such as the name, data type, and length. This information can be used to generate a schema for the table, which can then be used in Power Query to transform and analyze the data.

To use the SqlExpression.SchemaFrom function, you must provide the following arguments:

– ConnectionString: The connection string for the SQL database.

– Catalog: The name of the database catalog that contains the table.

– Schema: The name of the schema that contains the table.

– Table: The name of the table for which to retrieve metadata.

Once you have provided these arguments, the function will return a table that contains metadata about the columns in the specified table.

The M Code Behind SqlExpression.SchemaFrom

The M code behind the SqlExpression.SchemaFrom function is relatively simple. It involves creating a SQL statement that retrieves metadata about the columns in the specified table. Here is an example of the M code that generates a schema from a table called “Customers” in a SQL database:

```
let
  Source = Sql.Database(ConnectionString, Catalog),
  Schema = SqlExpression.SchemaFrom(Source, [Schema=Schema, Table=Table]),
  Output = #table(Schema[Name], Schema[Type])
in
  Output
```

Let’s break down each part of this code:

– Source: This line establishes a connection to the SQL database using the provided connection string and catalog name.

SqlExpression.ToExpression

What is SqlExpression.ToExpression?

The SqlExpression.ToExpression function is used to convert a SQL expression into a Power Query expression. This function is useful when you need to perform data transformations that involve SQL expressions. The function takes a SQL expression as input and returns a Power Query expression that represents the same logic as the SQL expression.

How does SqlExpression.ToExpression work?

The SqlExpression.ToExpression function works by parsing the SQL expression and generating the equivalent Power Query expression. Let's take a look at an example to understand how this works.

Suppose we have a SQL expression that looks like this:

```
SELECT FirstName, LastName, Email
FROM Customers
WHERE Age > 30
```

To convert this SQL expression into a Power Query expression using the SqlExpression.ToExpression function, we would use the following M code:

```
let
  sqlExpression = "SELECT FirstName, LastName, Email FROM Customers WHERE Age > 30",
  pqExpression = SqlExpression.ToExpression(sqlExpression)
in
  pqExpression
```

This M code first defines the SQL expression as a string variable called sqlExpression. It then passes this variable to the Sybase.Database

What is Sybase.Database?

Sybase is a relational database management system that is used by many organizations to manage their data. The Sybase.Database function in Power Query allows users to connect to and extract data from Sybase databases.

The M Code Behind Sybase.Database

When you use the Sybase.Database function in Power Query, the M code that is generated behind the scenes is actually quite simple. Here's an example of the M code that is generated when you use the Sybase.Database function to connect to a Sybase database:

```
let
  Source = Sybase.Database("[Server]", "[Database]", [Query="SELECT  FROM [Table]"]),
  #"Filtered Rows" = Table.SelectRows(Source, each ([Column1] = "Value")),
  #"Sorted Rows" = Table.Sort(#"Filtered Rows",{{"Column2", Order.Ascending}})
in
  #"Sorted Rows"
```

Let's break down each section of this code to understand what it does.

Step 1: Connect to the Sybase Database

The first step in the M code behind the Sybase.Database function is to connect to the Sybase database. This is done using the following code:

```
let
  Source = Sybase.Database("[Server]", "[Database]", [Query="SELECT  FROM [Table]"]),
```

In this code, you specify the server name and database name that you want to connect to. You can also specify an optional SQL query to

Table.AddColumn

Understanding the Table.AddColumn Function

The Table.AddColumn function takes three arguments: the table to modify, the name of the new column to create, and the function to use to generate the values for the new column. The function should take a single argument (a record) and return a single value (the value for the new column).

Here is an example of how to use Table.AddColumn to create a new column that combines the values in two existing columns:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMjQ2VNJRMgYyMjQ1VrIyNjC3GBgYGBgYGBgYGBgYGJgY
GBgYGJgYGBgYGBgYGBgYGBgYGBgYGBgYGBgYGBgYGBgYGBgYGBgYGBgYGBgYGBgYGBgYGBgrA8=", BinaryEncoding.Base64)),
let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [#"Column 1" = _t, #"Column 2" = _t]),
  #"Added Custom" = Table.AddColumn(Source, "Combined", each [Column 1] & "-" & [Column 2])
in
  #"Added Custom"
```

This code creates a new column called “Combined” that concatenates the values in the “Column 1” and “Column 2” columns, separated by a hyphen.

The M Code Behind the Table.AddColumn Function

The M code behind the Table.AddColumn function is relatively straightforward. Here is the basic structure of the function:

```
(Table as table, NewColumnName as text, ColumnExpression as function) as table =>
let
  NewColumn = Table.AddColumn(Table, NewColumnName, ColumnExpression, type any)
in
Table.AddFuzzyClusterColumn
```

One of the M functions used in Power Query is the Table.AddFuzzyClusterColumn function. This function is used to create fuzzy clusters based on the similarity between values in a column. This article will explain the M code behind the Table.AddFuzzyClusterColumn function and how it can be used to cluster data.

What are Fuzzy Clusters?

Fuzzy clustering is a method of grouping data points based on their similarity. Unlike traditional clustering methods, which assign each data point to a single cluster, fuzzy clustering assigns each data point a degree of membership to each cluster. This allows data points to belong to multiple clusters at the same time.

Fuzzy clustering is useful when dealing with data that has overlapping characteristics or when a data point cannot be clearly assigned to a single cluster. For example, in customer segmentation, a customer may have characteristics that belong to multiple segments.

How Does the Table.AddFuzzyClusterColumn Function Work?

The Table.AddFuzzyClusterColumn function takes four parameters:

– Table: The table to which the fuzzy cluster column will be added.

– Column: The column on which the fuzzy clustering will be performed.

– Options: A record containing the options for clustering. This includes the number of clusters, the similarity metric, and the maximum number of iterations.

– NewColumnName: The name of the new column that will be added to the table.

The function works by first calculating the similarity between each value in the column using the specified similarity metric. It then performs an iterative process to assign each value to a cluster based on its similarity to the other values in the column. The process continues until the maximum number of iterations is reached or until the clusters converge.

The result of the Table.AddFuzzyClusterColumn function is a new column in the table that contains the fuzzy cluster assignments for each value in the column.

Example Usage of the Table.AddFuzzyClusterColumn Function

Suppose we have a table containing customer data with columns for age, income, and spending. We want to perform fuzzy clustering on the spending column to identify groups of customers with similar spending patterns.

We can use the Table.AddFuzzyClusterColumn function to add a new column to the table containing the fuzzy cluster assignments for each customer. The function call might look like this:

Table.AddIndexColumn

The Table.AddIndexColumn function is part of the M language, the language used by Power Query to perform data transformations. In this article, we'll take a closer look at the M code behind the Table.AddIndexColumn function.

What is the Table.AddIndexColumn Function?

The Table.AddIndexColumn function is used to add an index column to a table. An index column is a column that contains a unique identifier for each row in a table. This can be useful for sorting, filtering, and joining tables.

The syntax for the Table.AddIndexColumn function is as follows:

```
Table.AddIndexColumn(table as table, newColumnName as text, optional offset as number, optional step as number) as table
```

The `table` parameter is the table to which the index column will be added. The `newColumnName` parameter is the name of the new index column. The `offset` parameter is the starting value for the index column (default is 0), and the `step` parameter is the increment value for the index column (default is 1).

The M Code Behind Table.AddIndexColumn

The M code behind the Table.AddIndexColumn function is relatively simple. Here's the code:

```
let
  addIndex = Table.AddIndexColumn(table, newColumnName, offset, step)
in
  addIndex
```

The first line of the code defines a new variable called `addIndex`. This variable uses the Table.AddIndexColumn function to add the index column to the table.

The second line of the code returns the `addIndex` variable, which contains the modified table.

Table.AddJoinColumn

The M code behind this function is what makes it so powerful. Let's take a closer look at how it works.
Basic Syntax of Table.AddJoinColumn
The basic syntax of `Table.AddJoinColumn` is as follows:

Table.AddJoinColumn(table1 as table, column1 as text, table2 as table, column2 as text, joinKind as nullable number, joinAlgorithm as nullable number) as table

Here's what each argument means:
– `table1`: The first table to join.
– `column1`: The column in `table1` to join on.
– `table2`: The second table to join.
– `column2`: The column in `table2` to join on.
– `joinKind`: An optional parameter that specifies the type of join to perform (e.g. inner join, left outer join, etc.).
– `joinAlgorithm`: An optional parameter that specifies the algorithm to use for joining the tables.
Joining Tables with Table.AddJoinColumn
To join two tables using `Table.AddJoinColumn`, you need to specify the two tables and the columns to join on. For example, let's say we have two tables: `Customers` and `Orders`. The `Customers` table contains information about customers, such as their name and address, while the `Orders` table contains information about the orders they have made.
To join these tables on the `CustomerID` column, we would use the following M code:

Table.AddJoinColumn(Customers, "CustomerID", Orders, "CustomerID")

This will create a new table that combines the columns from both tables, based on the `CustomerID` column.

Table.AddKey

What is a Key Column?

A key column is a column in a table that contains unique values for each row. It can be used to identify and combine related data across different tables. A key column is often created by concatenating other columns in the table to create a unique identifier for each row.

The Table.AddKey Function

The Table.AddKey function is a Power Query M function that adds a key column to a table. It takes two arguments: the first argument is the table to which the key column is to be added, and the second argument is a list of columns to use for creating the key column.

The syntax for the Table.AddKey function is as follows:

```
Table.AddKey(table as table, keyColumns as list, optional name as nullable text)
```

The first argument, `table`, is the table to which the key column is to be added. The second argument, `keyColumns`, is a list of columns to use for creating the key column. The third argument, `name`, is optional and is used to specify the name of the key column. If not specified, the name of the key column will be “Key”.

Example Usage

Let’s consider an example to understand the usage of the Table.AddKey function. Suppose we have a table named “Sales” with the following columns:

– Product
– Region
– Sales Amount

We want to add a key column to this table using the columns “Product” and “Region”. We can use the following M code to achieve this:

```
let
  Source = Excel.CurrentWorkbook(){[Name="Sales"]}[Content],
  #"Added Custom" = Table.AddKey(Source, {"Product", "Region"}, "SalesKey")
```

Table.AddRankColumn

In this article, we will explore the M code behind the Table.AddRankColumn function and how you can use it in your Power Query workflows.

What is Table.AddRankColumn?

Table.AddRankColumn is a function in Power Query that adds a rank column to your data based on a specified column. This function is useful when you want to sort and rank your data based on a particular column. The function takes two arguments: the table to add the rank column to, and the column to rank the data by.

Here is an example of how to use the Table.AddRankColumn function:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WclTSUcpWK80rVayj1cQ1NwIz0zNJCgtKLSnJzUvOLEnNzUnJzEwtLlKTSjVUoqKMhNjTUz1M1NSWlAgA=", BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type text) meta [Serialized.Text = true]) in type table [Column1 = _t]),
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"Column1", Int64.Type}}),
  #"Added Index" = Table.AddIndexColumn(#"Changed Type", "Index", 1, 1),
  #"Added Rank" = Table.AddRankColumn(#"Added Index", "Column1", true, ascending)
in
  #"Added Rank"
```

In this example, we start by creating a table from some sample data. We then transform the data by changing the data type of the “Column1” column to Int64. Next, we add an index column to the data using the Table.AddIndexColumn function. Finally, we add a rank column to the data using the Table.AddRankColumn function, ranking the data by the “Column1” column in ascending order.

How does Table.AddRankColumn work?

The Table.AddRankColumn function works by first sorting the data by the specified column. It then adds a new column to the data, which contains the rank of each row based on the sorted column. The rank is calculated by assigning a value of 1 to the first row, a value

Table.AggregateTableColumn

What is Table.AggregateTableColumn?

Table.AggregateTableColumn is an M function in Power Query that allows users to aggregate data in a table based on a specific column. This function takes three arguments: the table to aggregate, the column to group by, and the aggregation function to apply. The aggregation function can be any of the built-in functions in Power Query, such as Sum, Average, Min, Max, or Count.

The M Code Behind Table.AggregateTableColumn

The M code behind Table.AggregateTableColumn is relatively simple. The function takes three arguments: the table to aggregate, the column to group by, and the aggregation function to apply. The M code for this function is as follows:

```
(Table as table, ColumnToGroupBy as text, AggregationFunction as function) =>
  Table.Group(Table, {ColumnToGroupBy}, {{"Aggregation", AggregationFunction, ColumnToAggregate}})
```

Let’s break down this code line by line:

– `(Table as table, ColumnToGroupBy as text, AggregationFunction as function) =>` : This line defines the function parameters, which are the table to aggregate, the column to group by, and the aggregation function to apply.

– `Table.Group(Table, {ColumnToGroupBy}, {{“Aggregation”, AggregationFunction, ColumnToAggregate}})` : This line groups the table by the specified column and applies the specified aggregation function. The result is a new table with the aggregated values.

Using Table.AggregateTableColumn in Power Query

Table.AggregateTableColumn can be used in Power Query to transform and analyze data in a variety of ways. Here are some examples of how this function can be used:

Example 1: Summing a Column by Group

Suppose we have a table with sales data for different products and we want to sum the sales by product category. We can use Table.AggregateTableColumn to aggregate the sales data by category and sum the values. Here’s the M code to do this:

```
let
Table.AlternateRows
```

In this article, we will explore the M code behind the Table.AlternateRows function and learn how to use it effectively.

What is Power Query?

Before we dive into the specifics of the Table.AlternateRows function, let's first understand what Power Query is. Power Query is a data transformation and cleansing tool that is built into Microsoft Excel and other Office applications.

Power Query allows you to connect to a variety of data sources, combine and transform data, and load the results into Excel or other destinations. With Power Query, you can easily clean and reshape data, automate data preparation tasks, and create powerful data models.

Using Table.AlternateRows to Alternate Row Colors

The Table.AlternateRows function is used to alternate the colors of rows in a table. This function takes two arguments: the name of the table and the number of rows to alternate between.

To use the Table.AlternateRows function, start by selecting the table you want to format. Then, go to the "Transform" tab in the Power Query Editor and select "Add Column" from the ribbon. In the dropdown menu, select "Custom Column".

In the "Custom Column" dialog, enter a name for the new column and then enter the following formula in the "Custom column formula" field:

```
= Table.AlternateRows(#"Previous Step Name", Number.ToText(Number of Rows), Number.ToText(Number of Rows))
```

Replace "Previous Step Name" with the name of the step that contains the table you want to format. Replace "Number of Rows" with the number of rows you want to alternate between. For example, if you want to alternate between every two rows, enter "2" for both arguments.

After entering the formula, click "OK" to create the new column. You should now see a new column in your table with alternating row colors.

Understanding the M Code

Now that we have successfully used the Table.AlternateRows function, let's take a closer look at the M code behind it.

The Table.AlternateRows function is actually a wrapper function that uses several other M functions to achieve the alternating row

Table.ApproximateRowCount

Introduction to Power Query

Power Query is a data transformation and cleansing tool that is used to extract, transform, and load (ETL) data from a variety of sources. It is a part of Microsoft Excel and is also available as a separate add-in for Excel 2010 and 2013. Power Query allows you to transform and cleanse data from a variety of sources such as text files, Excel workbooks, databases, web pages, and more. Power Query is a part of the Power BI suite and is also integrated with other Microsoft products such as SharePoint, SQL Server, and Power Apps.

What is Table.ApproximateRowCount?

Table.ApproximateRowCount is a Power Query M function that returns an estimate of the number of rows in a table. This function is used to improve the performance of your Power Query queries by providing an estimate of the number of rows in a table before it is loaded into memory. This estimate is used to optimize the loading and transformation of data, which can significantly improve the performance of your queries.

How to use Table.ApproximateRowCount

To use Table.ApproximateRowCount, you need to provide a table or a reference to a table. The function then returns an estimate of the number of rows in the table. The syntax of the function is as follows:

```
Table.ApproximateRowCount(table as table) as number
```

The function takes a single argument, which is the table or a reference to the table. The function then returns an estimate of the number of rows in the table.

The following example demonstrates how to use Table.ApproximateRowCount in Power Query:

```
let
  Source = Excel.Workbook(File.Contents("C:Sales.xlsx"), null, true),
  Sales_Table = Source{[Item="Sales",Kind="Table"]}[Data],
  RowCount = Table.ApproximateRowCount(Sales_Table)
```

Table.Buffer

In this article, we will dive into the M code behind the Table.Buffer function, exploring how it works and how to use it effectively in your Power Query workflows.

Understanding Table.Buffer

Before we dive into the M code behind Table.Buffer, let's briefly touch on what this function does and why it's useful.

When you create a query in Power Query, it's executed each time you refresh your data. This means that if you have multiple queries that use the same data source or that perform similar transformations, your data is being loaded and processed multiple times. This can slow down your query performance and increase the load on your data source.

Table.Buffer allows you to store the results of a query or transformation in memory, essentially creating a cache that can be used by subsequent queries and transformations. This means that if you have multiple queries that use the same data source or perform similar transformations, they can all reference the cached data, improving performance and reducing the load on your data source.

The M Code Behind Table.Buffer

The M code behind Table.Buffer is relatively simple. When you apply the Table.Buffer function to a table, it creates a reference to that table that is stored in memory. Any subsequent queries or transformations that reference that table will use the cached data instead of re-executing the original query or transformation.

Here's an example of how you might use Table.Buffer in a simple Power Query workflow:

```
let
  Source = Excel.Workbook(File.Contents("C:data.xlsx"), null, true),
  Sheet1_Sheet = Source{[Item="Sheet1",Kind="Sheet"]}[Data],
  #"Promoted Headers" = Table.PromoteHeaders(Sheet1_Sheet, [PromoteAllScalars=true]),
  #"Filtered Rows" = Table.SelectRows(#"Promoted Headers", each ([Column1] <> null)),
  Buffer = Table.Buffer(#"Filtered Rows")
in
  Buffer
```

Table.Column

Understanding the Table.Column Function

The `Table.Column` function is a powerful M function in Power Query that allows users to extract a specific column from a table. The function takes two arguments: the table to extract the column from, and the name of the column to extract. Here is the syntax for the `Table.Column` function:

```
Table.Column(Table as table, ColumnName as text) as list
```

The `Table` argument is the table to extract the column from, while the `ColumnName` argument is the name of the column to extract. The function returns a list of values from the specified column.

The M Code Behind Table.Column

The `Table.Column` function is implemented in M code, which is the language behind Power Query. The M code for the `Table.Column` function is relatively simple and can be broken down into three main steps:

1. Find the index of the column to extract
2. Extract the values from the specified column
3. Return the list of values

Here is the M code for the `Table.Column` function:

```
(Table as table, ColumnName as text) =>
  let
    ColumnIndex = Table.ColumnIndex(Table, ColumnName),
    Values = Table.Column(Table, ColumnIndex),
    Result = Values
  in
    Result
```

Table.ColumnCount

Table.ColumnCount is a function that returns the number of columns in a given table. In this article, we will explore the M code behind this function and how it works.

Understanding the M Language

Before diving into the M code behind Table.ColumnCount, it's important to understand the M language. M is the scripting language used in Power Query to manipulate and transform data. It's a functional language that is similar to other programming languages such as Python and R.

M code is made up of expressions and statements. An expression is a value or a calculation that returns a value. A statement is a group of expressions that are executed in sequence.

The M Code Behind Table.ColumnCount

To understand the M code behind Table.ColumnCount, we will need to create a sample table. In Power Query, go to the Home tab and click on the "From Table" button. This will open a dialog box where you can enter your data.

For this example, we will create a table with three columns:

Column 1	Column 2	Column 3
———-	———-	———-
Value 1	Value 2	Value 3
Value 4	Value 5	Value 6
Value 7	Value 8	Value 9

Once you have created your table, go to the "Advanced Editor" in Power Query. This will open the M code for your table.

The M code for our sample table should look like this:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WclTSUTIxVkjNyclWJVYyNzLXMjQwNDC3MDfRyM/Nz83My0xNTI3MLQwNjIwM7QwNzAzsDGyMDDWNTU3sDQys7Ty1s0tKLAE=", BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Column1 = _t, Column2 = _t, Column3 = _t]),
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"Column1", type text}, {"Column2", type text}, {"Column3", type text}})
Table.ColumnNames
```

In this article, we will take an in-depth look at the M code behind the Table.ColumnNames function and explore how it works.

Understanding the Table.ColumnNames Function

The Table.ColumnNames function is a simple but powerful function that returns a list of column names for a given table. The function takes a single parameter, which is the name of the table that you want to retrieve the column names for. Here is the syntax for the Table.ColumnNames function:

```
Table.ColumnNames(table as table) as list
```

The function returns a list of column names for the specified table. Here is an example of how to use the Table.ColumnNames function in Power Query:

```
let
  Source = Excel.Workbook(File.Contents("C:MyWorkbook.xlsx"), null, true),
  Sheet1_Sheet = Source{[Item="Sheet1",Kind="Sheet"]}[Data],
  #"Promoted Headers" = Table.PromoteHeaders(Sheet1_Sheet, [PromoteAllScalars=true]),
  ColumnNames = Table.ColumnNames(#"Promoted Headers")
in
  ColumnNames
```

In this example, we are using the Table.ColumnNames function to retrieve a list of column names for a table that we have loaded from an Excel workbook. We are first loading the workbook using the Excel.Workbook function, and then selecting the Sheet1 worksheet using the Source{[Item=”Sheet1",Kind=”Sheet”]}[Data] syntax. We are then promoting the headers of the table using the Table.PromoteHeaders function, and then using the Table.ColumnNames function to retrieve a list of column names for the resulting

Table.ColumnsOfType

Understanding the Table.ColumnsOfType Function

The Table.ColumnsOfType function is one of the many M functions available in Power Query. As the name suggests, this function allows you to filter out columns of a specific data type from a table. The syntax of the Table.ColumnsOfType function is as follows:

The first argument of the function is the table from which you want to filter out columns. The second argument is the data type of the columns that you want to filter. The function returns a list of columns that match the specified data type.

Exploring the M Code Behind the Table.ColumnsOfType Function

To understand the M code behind the Table.ColumnsOfType function, let's consider an example. Suppose we have a table with the following columns:

Name	Age	Salary
John	25	5000
Jane	30	6000
Jack	35	7000

Suppose we want to filter out the columns that contain numeric data types. To do this, we can use the Table.ColumnsOfType function as follows:

The above code will return a list containing two columns: Age and Salary. Let’s break down the M code behind this function to understand how it works.

The above code creates a table from a list of records. Each record represents a row in the table. In this case, we have three records representing the three rows in the table.

Table.Combine

What is M Code?

M code is the language used by Power Query to transform and manipulate data. It is a functional language that is similar to Excel formulas, but more powerful. M code is used to create queries that bring data from various sources, transform it, and load it into a destination. You don't need to be an expert in M code to use Power Query, but having a basic understanding of how it works can make your life easier.

What is Table.Combine?

Table.Combine is a Power Query M function that allows you to combine multiple tables into a single table. It is useful when you have multiple tables with the same schema and you want to merge them into a single table. The function takes a list of tables as input and returns a single table that contains all the rows from each table.

How does Table.Combine work?

Table.Combine works by concatenating the rows from each table into a single table. The function takes a list of tables as input and combines them into a single table by appending the rows from each table to the end of the previous table. The function ignores the column headers of each table and assumes that all tables have the same schema. If the tables have different schemas, you will need to use other functions to transform the data before using Table.Combine.

Syntax of Table.Combine

The syntax of Table.Combine is as follows:

```
Table.Combine(list as list)
```

The function takes one argument, which is a list of tables. The list can contain any number of tables, but they must all have the same schema. The function returns a single table that contains all the rows from each table.

Example of Table.Combine

Let's look at an example of how to use Table.Combine. Suppose we have two tables, Table1 and Table2, with the same schema. We want to combine these tables into a single table. We can use the following M code to achieve this:

Table.CombineColumns

What is Table.CombineColumns?

Table.CombineColumns is a Power Query M function that allows us to combine the values of two or more columns into a single column. This can be useful when we have data that is split across multiple columns and we want to combine it into a single column for analysis or reporting purposes.

How to Use Table.CombineColumns

The syntax for Table.CombineColumns is as follows:

```
Table.CombineColumns(table as table, column1 as text, column2 as text, separator as text, newColumnName as text)
```

`table`: The table that contains the columns we want to combine.
`column1`, `column2`: The names of the columns we want to combine.
`separator`: The character or text string that we want to use to separate the values in the combined column.
`newColumnName`: The name of the new column that will contain the combined values.

Here is an example of how we might use Table.CombineColumns:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMjQ1VrJQ0lEq1lEyNDXUNjQwNjA0MjB2MnFyMnUwNjA0MjB2MnFyMnUwMjAwNjA0MjB2MnFyMnUwNjA0MjB2MnFyMnUwMjAwNjA0MjB2MnFyMnUwNjA0MjB2MnFyMnUwMjAwNjA0MjB2MnFyMnUwNjA0MjB2MnFyMnUwMjAwNjA0MjB2MnFyMnUwNjA0MjB2MnFyMnUwMjAwNjA0MjB2MnFyMnUwMjAwNjEyMzRyNjAzNlqNzYCjbQ1mlKwEqAE=", BinaryEncoding.Base64)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Column1 = _t, Column2 = _t, Column3 = _t, Column4 = _t, Column5 = _t, Column6 = _t, Column7 = _t]),
  #"Combined Columns" = Table.CombineColumns(Source,{"Column2", "Column3"},", ","NewColumn")
in
Table.CombineColumnsToRecord
```

Understanding the Table.CombineColumnsToRecord Function

The Table.CombineColumnsToRecord function is used to combine multiple columns in a table into a single record. The resulting record is created by combining the values of the selected columns, with the column names used as the record field names. Here's the basic syntax of the function:

```
Table.CombineColumnsToRecord(table as table, columnNames as list, newColumnName as text)
```

The function takes three arguments:

– table: The table containing the columns to be combined.

– columnNames: A list of column names to combine.

– newColumnName: The name of the new column to create.

Here's an example of how to use the function:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WcslNzEtVsllwMjIwVdQyKMlMzXWz0vz8xNzyzLBwA=",
BinaryEncoding.Base64)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Column1 = _t, Column2 = _t, Column3
= _t])),
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"Column1", type text}, {"Column2", type text}, {"Column3", type text}}),
  #"Combined Columns" = Table.CombineColumnsToRecord(#"Changed Type",{"Column1", "Column2", "Column3"}, "Combined")
in
  #"Combined Columns"
```

Table.ConformToPageReader

What is the Table.ConformToPageReader Function?

The Table.ConformToPageReader function is used in Power Query M to ensure that tables are formatted correctly for the page reader. Page readers are used to read data from a web page or document, and they have specific requirements for how data is formatted. The Table.ConformToPageReader function ensures that tables are formatted in a way that is readable by the page reader.

How Does the Table.ConformToPageReader Function Work?

The Table.ConformToPageReader function works by taking a table as input and returning a new table that conforms to the page reader's requirements. The function works by analyzing the table and making changes to the formatting and structure of the table as necessary.

The Table.ConformToPageReader function performs several operations on the input table to ensure that it conforms to the page reader's requirements. These operations include:

– Removing unnecessary columns
– Renaming columns to make them more readable
– Formatting columns to ensure that they are readable by the page reader
– Converting data types to ensure that they are compatible with the page reader

Examples of Using the Table.ConformToPageReader Function

Here are some examples of how you can use the Table.ConformToPageReader function in Power Query M:

Example 1: Removing Unnecessary Columns

Suppose you have a table with several columns, but you only want to display a few of them in the page reader. You can use the Table.ConformToPageReader function to remove the unnecessary columns. The code for this would look something like this:

```
let
  Source = ,
  ConformedTable = Table.ConformToPageReader(Source, {{"Column1", type text}, {"Column2", type number}, {"Column3", type text}}, true)
in
  ConformedTable
```

Table.Contains

What is the Table.Contains Function?

The Table.Contains function is a M function in Power Query that checks if a table contains a certain value or not. It returns a Boolean value (true or false) depending on whether the value is present in the table or not.

The syntax for the Table.Contains function is as follows:

```
Table.Contains(table as table, compareCriteria as any, optional equationCriteria as nullable function) as logical
```

– `table`: The table to check for the value.
– `compareCriteria`: The value to check for in the table.
– `equationCriteria`: (Optional) A function that defines how to compare the value with the table.

Understanding the M code behind the Table.Contains Function

To understand the M code behind the Table.Contains function, let's take a look at an example. Consider the following table:

Name	Age	Gender
John	30	Male
Jane	35	Female
Peter	40	Male

Suppose we want to check if the "Name" column contains the value "John". We can do this using the following M code:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45W0llyU9YvVgoyUjIwNzJQyMlWsgwNzVQyMzTHQ0MzU2NzSxBQj08sVZB2QZI/BQ=", BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Name = _t, Age = _t, Gender = _t]),
```

Table.ContainsAll

The M language is a functional programming language that is used to define queries and transformations in Power Query. It allows you to manipulate data in a powerful and flexible way. In this article, we will explore the M code behind the Power Query M function Table.ContainsAll.

What is Table.ContainsAll?

Table.ContainsAll is a M function in Power Query that is used to check if a table contains all the values in a given list. It returns true if all the values are found in the table, and false otherwise. The syntax of the Table.ContainsAll function is as follows:

```
Table.ContainsAll(table as table, values as list, optional equationCriteria as any) as logical
```

The first parameter is the table that you want to check, the second parameter is the list of values that you want to check for, and the third parameter is an optional equation criteria that you can use to specify the comparison method.

How does Table.ContainsAll work?

The Table.ContainsAll function works by iterating over the list of values and checking if each value is present in the table. If at least one value is not found, it returns false. However, you can customize the behavior of the function by using the equation criteria parameter. The equation criteria parameter allows you to specify the comparison method that should be used to match the values in the table. By default, the function uses the equality operator (=) to compare the values. However, you can use other comparison operators such as <=, >=, <, >, and <>.

You can also use a predicate function as the equation criteria parameter. A predicate function is a function that takes two arguments and returns a logical value. It can be used to implement custom comparison logic.

Examples of using Table.ContainsAll

Let’s take a look at some examples of how to use the Table.ContainsAll function in Power Query.

Example 1: Basic usage

Suppose we have a table called “Sales” that contains data about sales transactions. We want to check if the table contains all the product names in a list called “ProductList”. We can use the following M code:

Table.ContainsAny

Overview of Table.ContainsAny Function

The Table.ContainsAny function is used to check if any of the values in a given list are present in a column of a table. It returns a Boolean value of True if at least one of the values in the list is found in the column, and False otherwise. The syntax of the Table.ContainsAny function is:

Table.ContainsAny(table as table, values as list, columns as list, comparer as nullable function) as logical

The arguments of the Table.ContainsAny function are:

- table: The table to be searched.
- values: A list of values to search for.
- columns: A list of columns to search in.
- comparer: An optional argument that specifies the comparison method to be used when searching for values.

The M Code Behind Table.ContainsAny Function

The M code behind the Table.ContainsAny function is a combination of several other M functions, including List.ContainsAny, Table.SelectColumns, and List.Intersect. The following is the M code for the Table.ContainsAny function:

```
(Table as table, values as list, columns as list, optional comparer as nullable function) =>
let
   SelectColumns = Table.SelectColumns(Table, columns),
   Intersection = List.Intersect(SelectColumns{0}, values, comparer),
   Result = List.Count(Intersection) > 0
in
   Result
```

Table.DemoteHeaders

What is the Table.DemoteHeaders function?

The Table.DemoteHeaders function is a Power Query M function that is used to demote the header row of a table to a regular data row. When you import data into Power Query, the first row of the data is automatically promoted to a header row. This is useful when you want to work with the data in Excel, but it can be a problem when you want to use the data in Power Query.

The Table.DemoteHeaders function is used to solve this problem. It takes a table as input and returns a new table with the header row demoted to a regular data row.

The M code behind the Table.DemoteHeaders function

The M code behind the Table.DemoteHeaders function is relatively simple. Here is the code:

```
(table as table) as table =>
let
  header = Table.PromoteHeaders(table),
  demoted = Table.AddIndexColumn(header, "Index", 0, 1),
  newHeaders = Table.ColumnNames(demoted){0},
  newTable = Table.Skip(demoted, 1),
  renamed = Table.RenameColumns(newTable, {{newHeaders, "Header"}}),
  reordered = Table.ReorderColumns(renamed, {"Index", "Header", Table.ColumnNames(renamed){2..}})
in
  reordered
```

Let’s break down this code step by step.

1. The first line of the code defines the input parameter of the function, which is a table.
2. The second line of the code promotes the header row of the table to a header row.
3. The third line of the code adds an index column to the table.
4. The fourth line of the code gets the name of the new header column.

Table.Distinct

Table.Distinct is a powerful function that can help you remove duplicate values from your data. In this article, we will explore the M code behind the Table.Distinct function and how it can be used to clean and manipulate your data.

Understanding the Table.Distinct Function

The Table.Distinct function is used to remove duplicate rows from a table. It takes a table as an input and returns a new table with only the unique rows.

Let's take a look at the syntax of the Table.Distinct function:

```
Table.Distinct(table as table, optional equationCriteria as any, optional columnsToCompare as any) as table
```

The function takes three arguments:

– `table`: This is the input table that you want to remove duplicates from.

– `equationCriteria`: This is an optional argument that allows you to specify the criteria for determining duplicates. By default, Table.Distinct considers all columns in the input table when determining duplicates. However, you can use the equationCriteria argument to specify a custom comparison function.

– `columnsToCompare`: This is an optional argument that allows you to specify the columns to use when determining duplicates. By default, Table.Distinct considers all columns in the input table. However, you can use the columnsToCompare argument to specify a subset of columns to compare.

Now that we understand the syntax of the Table.Distinct function, let's take a look at the M code behind it.

Understanding the M Code Behind Table.Distinct

The M code behind the Table.Distinct function is relatively simple. It uses the Table.Buffer function to create a copy of the input table and then uses the List.Distinct function to remove duplicates.

Here is the M code for the Table.Distinct function:

```
(table as table, optional equationCriteria as any, optional columnsToCompare as any) =>
Table.DuplicateColumn
```

Understanding the Table.DuplicateColumn Function

The Table.DuplicateColumn function is a simple yet highly useful function in Power Query. It allows users to create a new column in a table that is a copy of an existing column, with a new name. This can be useful in a range of scenarios, such as when you want to apply a different set of transformations to the same data, or when you want to perform calculations on two different columns.

Here is an example of how to use the Table.DuplicateColumn function in Power Query:

```
let
  Source = Table.FromRecords({
    [Name="Alice", Age=25, City="London"],
    [Name="Bob", Age=30, City="Paris"],
    [Name="Charlie", Age=35, City="New York"]
  }),
  DuplicateColumn = Table.DuplicateColumn(Source, "Age", "AgeCopy")
in
  DuplicateColumn
```

In this example, we start by creating a new table called “Source” with three rows of data. We then use the Table.DuplicateColumn function to create a new column called “AgeCopy”, which is a copy of the “Age” column in the “Source” table.

The M Code Behind the Table.DuplicateColumn Function

To better understand how the Table.DuplicateColumn function works, it’s important to take a look at the M code behind it. The M code is the language used by Power Query to perform data transformations and create queries.

Here is an example of the M code behind the Table.DuplicateColumn function:

```
(Table as table, ColumnName as text, NewColumnName as text) as table =>
Table.ExpandListColumn
```

What is Table.ExpandListColumn?

Table.ExpandListColumn is a function in the M language that allows users to expand a list column within a table into multiple rows. This is particularly useful when working with data that has a hierarchical structure. For example, if you have a table that contains a column of orders, each order might contain a nested column of products. By using Table.ExpandListColumn, you can expand the list of products within each order into individual rows, making it easier to analyze the data.

The Syntax of Table.ExpandListColumn

The syntax of Table.ExpandListColumn is as follows:

```
Table.ExpandListColumn(table as table, column as text) as table
```

The first argument, “table”, is the table that you want to expand. The second argument, “column”, is the name of the column that contains the list that you want to expand. The function returns a new table with the expanded list.

Examples of Using Table.ExpandListColumn

Let’s take a look at some examples of how to use Table.ExpandListColumn.

Example 1: Expand a Simple List Column

Suppose you have a table called “Orders” that contains a column called “Products”. The “Products” column contains a list of products for each order. To expand the list into individual rows, you can use the following code:

```
let
  Source = Orders,
  ExpandProducts = Table.ExpandListColumn(Source, "Products")
in
  ExpandProducts
```

Table.ExpandRecordColumn

This function is widely used in data transformations, and understanding its M code is essential for advanced users who want to harness the full power of Power Query. In this article, we will take a closer look at the M code behind Table.ExpandRecordColumn and provide some examples of its usage.

Understanding Table.ExpandRecordColumn

Table.ExpandRecordColumn is a Power Query M function that expands a nested record column into separate columns. It takes two arguments: the table to expand and the name of the column that contains the nested records. The nested records are then expanded into separate columns, and the original column is removed.

The syntax for Table.ExpandRecordColumn is as follows:

```
Table.ExpandRecordColumn(table as table, column as text) as table
```

Here, `table` is the source table that contains the nested record column, and `column` is the name of the nested record column. The function returns a new table with the expanded columns.

Examples of Table.ExpandRecordColumn

Let’s look at some examples of how Table.ExpandRecordColumn can be used in Power Query.

Example 1: Expanding a single nested record column

Suppose we have a table called `Sales` that contains a nested record column called `Product`. The `Product` column contains multiple properties such as `ProductID`, `ProductName`, `Price`, `Category`, etc. We want to expand the `Product` column into separate columns.

Here’s the M code to achieve this:

```
let
  Source = Sales,
  ExpandedProduct = Table.ExpandRecordColumn(Source, "Product"),
```

Table.ExpandTableColumn

The Basics of Table.ExpandTableColumn

The `Table.ExpandTableColumn` function takes two arguments: the first is the table to expand, and the second is the name of the column to expand. Let's take a look at an example:

```
let
  Source = Table.FromRecords({
    [Name="John", Age=30, Address=[Street="123 Main St", City="Anytown", State="CA"]],
    [Name="Jane", Age=25, Address=[Street="456 Oak Ave", City="Smalltown", State="TX"]]
  }),
  ExpandAddress = Table.ExpandTableColumn(Source, "Address")
in
  ExpandAddress
```

In this example, we have a table with three columns: Name, Age, and Address. The Address column contains a nested record with three fields: Street, City, and State. We use the `Table.ExpandTableColumn` function to expand the Address column into three separate columns, resulting in a table with six columns.

Handling Lists and Records

Table.ExpandTableColumn is not limited to expanding columns containing nested records. It can also be used to expand columns containing lists or records. Let's take a look at an example with a list:

```
let
  Source = Table.FromRecords({
    [Name="John", Age=30, Skills={"Programming", "Data Analysis", "Project Management"}],
    [Name="Jane", Age=25, Skills={"Marketing", "Graphic Design"}]
```

Table.FillDown

How Table.FillDown Works

Table.FillDown is a function that can be applied to any column in a table in Power Query. Its purpose is to populate null or empty cells in that column with the last non-null value in that column. This can be useful when analyzing data that has gaps or missing values, and you want to fill those gaps with data from the most recent non-null value.

For example, let's say you have a table with a column that contains the following values:

Column A
1
null
null
2
null
3
null
null
4

If you apply the Table.FillDown function to this column, it will fill in the null values with the last non-null value in that column. The resulting column will look like this:

Column A
1
1
1

Table.FillUp

In this article, we will explore the M code behind the Table.FillUp function, how it works, and how it can be used in Power Query.

Understanding the Table.FillUp Function

The Table.FillUp function is a transformation function in Power Query that is used to fill null or blank values in a table with the value that appears in the previous non-null cell. It is similar to the Excel function “Fill Down” but can be used on any table in Power Query.

The syntax for the Table.FillUp function is as follows:

```
Table.FillUp(table as table, optional columns as nullable list, optional orderCriteria as nullable any)
```

`table` – the table to be filled up

`columns` – the list of columns to be filled up. If not specified, all columns in the table will be filled up.

`orderCriteria` – the order in which the table is sorted. If not specified, the table will be sorted by the first column.

How the Table.FillUp Function Works

The Table.FillUp function works by filling null or blank values in a table with the value that appears in the previous non-null cell. It does this by iterating over each row in the table and filling up the null values in each column based on the value in the previous non-null cell.

For example, consider the following table:

ID	Name	Age	Gender
1	John	30	Male
2		35	Male
3	Jane		Female
4			Male

Applying the Table.FillUp function to this table would result in the following table:

ID	Name	Age	Gender
1	John	30	Male

Table.FilterWithDataTable

In this article, we will take a deep dive into the M code behind the Power Query M function Table.FilterWithDataTable. We will explore its capabilities, syntax, and usage scenarios.

What is Table.FilterWithDataTable Function?

Table.FilterWithDataTable function is a Power Query M function that filters a table based on the contents of another table. The function takes two tables as inputs: the source table and the filter table. The filter table contains the filter criteria, which are used to filter the source table.

The function returns a table that contains only the rows from the source table that match the filter criteria specified in the filter table.

Syntax of Table.FilterWithDataTable Function

The syntax of the Table.FilterWithDataTable function is as follows:

```
Table.FilterWithDataTable(table as table, filterTable as table, joinColumns as list, optional columnsToInclude as nullable list) as table
```

– `table` is the source table that needs to be filtered.

– `filterTable` is the table that contains the filter criteria.

– `joinColumns` is a list of column names that are used to join the two tables. The column names must be present in both the source table and the filter table.

– `columnsToInclude` is an optional parameter that specifies the list of columns to include in the output table. If this parameter is not specified, all columns are included in the output table.

How to use Table.FilterWithDataTable Function?

Let’s take an example to understand how to use the Table.FilterWithDataTable function.

Suppose we have two tables: `Sales` and `Filter`. The `Sales` table contains the sales data of a company, and the `Filter` table contains the list of products that need to be included in the analysis. We want to filter the `Sales` table to include only the products that are present in the `Filter` table.

The `Sales` table looks like this:

| Product | Region | Sales |

Table.FindText

What is the M language?

Before we dive into the code, it's important to understand what the M language is. M is the programming language used to create custom functions and queries in Power Query. It's a functional language, meaning that it's based on the concept of functions and expressions.

Syntax of Table.FindText

Let's take a look at the syntax of the Table.FindText function:

```
Table.FindText(table as table, searchCriteria as text, optional occurrence as nullable number) as nullable number
```

The function takes three arguments:

1. `table`: The table to search within.
2. `searchCriteria`: The text to search for.
3. `optional occurrence`: The occurrence number of the text to find.

The function returns the position of the first occurrence of the text in the table, or null if the text is not found.

The M code behind Table.FindText

The M code behind Table.FindText is relatively simple. Here is the code:

```
(table as table, searchCriteria as text, optional occurrence as nullable number) =>
  let
    tableHeaders = Table.ColumnNames(table),
    searchColumns = List.Select(tableHeaders, each Type.Is(Text.Type, Table.ColumnType(table, _))),
    foundRows = Table.SelectRows(table, each List.Contains(List.Transform(searchColumns, each Record.Field(_, searchCriteria)), true)),
    result = if List.IsEmpty(foundRows) then null else Table.PositionOf(foundRows, foundRows{0}),
Table.First
```

Understanding the Table.First Function

The Table.First function is used to return the first row of a table. It takes a table as input and returns a record representing the first row of the table. The syntax for the Table.First function is as follows:

```
Table.First(table as table) as record
```

The Table.First function takes one argument, which is a table. The function returns a record representing the first row of the table.

The M Code Behind Table.First

The M code behind the Table.First function is straightforward. When you call the Table.First function, Power Query generates the following M code:

```
(table as table) =>
  let
    Source = Table.FirstN(table,1),
    #"Converted to Table" = Record.ToTable(Source{0})
  in
    #"Converted to Table"
```

The M code above defines an anonymous function that takes a table as input. The function then uses the Table.FirstN function to return the first row of the table as a table value. Next, the function uses the Record.ToTable function to convert the record into a table.

Breaking Down the M Code

Now that we've seen the M code behind Table.First, let's break down each step in the function.

Defining the Function

Table.FirstN

What Is Power Query M?

Power Query M is the powerful functional programming language used in Power Query. It provides a way to create custom functions, transforms data, and perform calculations. Power Query M is based on the F# programming language, and it is used to create formulas that are used to transform data in Power Query. The M code is written in a text editor and executed in the Power Query Editor.

What Is the Table.FirstN Function?

The Table.FirstN function is a Power Query M function that returns the first n rows of a table. It takes two arguments, the table to be filtered, and the number of rows to return. The syntax of the function is as follows:

```
Table.FirstN(table as table, countOrCondition as any) as table
```

The first argument, `table as table`, specifies the table to be filtered. The second argument, `countOrCondition as any`, specifies the number of rows to return or a Boolean expression that evaluates to true or false.

How to Use the Table.FirstN Function

To use the Table.FirstN function, follow these steps:

1. Open the Power Query Editor by selecting “Edit Queries” from the “Data” tab in Excel or Power BI.
2. Import the data you want to filter into Power Query.
3. Select the table you want to filter in the “Queries” pane.
4. Click the “Add Column” tab in the “Ribbon” and select “Custom Column”.
5. In the “Custom Column” dialog box, enter the following formula:

```
Table.FirstN([Table], 10)
```

This formula will return the first 10 rows of the table.

Table.FirstValue

One of the most commonly used M functions in Power Query is the Table.FirstValue function. This function is used to return the first value of a column in a table. In this article, we will discuss the M code behind the Table.FirstValue function and how it can be used in Power Query.

Syntax

The syntax for the Table.FirstValue function is as follows:

```
Table.FirstValue(table as table, column as text, optional defaultValue as any) as any
```

– `table`: The table containing the column to be evaluated.
– `column`: The name of the column to be evaluated.
– `defaultValue`: An optional value to be returned if the column is empty.

Example

Let’s take a look at an example of how the Table.FirstValue function can be used in Power Query:

```
let
  Source = Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WcisqzcnJSijKzUwMUbJwMlCKjW0MlFV4A", BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Column1 = _t]),
  FirstValue = Table.FirstValue(Source, "Column1")
in
  FirstValue
```

In this example, we have a table named “Source” with one column named “Column1”. We want to return the first value of the “Column1” column. We use the Table.FirstValue function to achieve this.

Table.FromColumns

In this article, we will delve into the M code behind the Table.FromColumns function, and explore how it works and how to use it effectively.

The Syntax of Table.FromColumns

Before we dive into the details of the M code behind Table.FromColumns, let's first take a look at its syntax. The basic syntax of the function is as follows:

```
Table.FromColumns(columns as list, optional headers as list)
```

The `columns` parameter is a required list of columns that will be used to create the table. Each column in the list should be a list of values of the same data type. The `headers` parameter is an optional list of headers for each column. If headers are not provided, the function will create default headers for each column.

The M Code Behind Table.FromColumns

The M code behind Table.FromColumns is relatively simple, yet powerful. The function takes a list of columns as input, and constructs a table from it. Here is the M code behind the function:

```
(Table as list, Headers as list) =>
let
  Headers = if Headers = null then List.Transform(Table, each "Column " & Text.From(List.PositionOf(Table, _))) else Headers,
  ColumnCount = List.Count(Table{0}),
  TableWithHeaders = Table.FromColumns(Table, Headers),
  Result = Table.Resize(TableWithHeaders, {ColumnCount})
in
  Result
```

Table.FromList

Understanding the Table.FromList function

The Table.FromList function is a handy tool for creating tables in Power Query. Essentially, it takes a list of values and returns a table where each value is its own row in the table. For example, suppose you have a list of names:

{"John", "Mary", "David", "Sarah"}

Using the Table.FromList function, you can create a table where each name is its own row:

Name
John
Mary
David
Sarah

This can be incredibly useful for a variety of data transformation tasks. But how does the Table.FromList function actually work? Let's take a look at the M code behind the function to find out.

Breaking down the M code

When you use the Table.FromList function in Power Query, the M code that gets generated looks like this:

```
let
  Source = List,
  #"Converted to Table" = Table.FromList(Source, Splitter.SplitByNothing(), null, null, ExtraValues.Error)
```

Table.FromPartitions

Understanding the Table.FromPartitions Function

The Table.FromPartitions function works by combining data from multiple partitions, which are essentially subsets of a larger dataset. This can be useful when working with data that is too large to handle in a single table, or when dealing with data that is stored in multiple files or databases.

To use Table.FromPartitions, you must first create a list of partitions that contain the data you want to merge. These partitions can be generated using the Table.Partition function, which splits a larger dataset into smaller, manageable pieces.

Once you have your list of partitions, you can use the Table.FromPartitions function to combine them into a single table. The resulting table will contain all of the data from all of the partitions, allowing you to work with a complete dataset without having to manually merge individual tables.

Writing the M Code for Table.FromPartitions

To write the M code for Table.FromPartitions, you must first define your list of partitions. This can be done using the Table.Partition function, as mentioned earlier.

For example, suppose you have a dataset that is stored in two separate CSV files. You could create a list of partitions using the following M code:

```
let
  Source1 = Csv.Document(File.Contents("C:Datafile1.csv"),[Delimiter=",", Encoding=1252, QuoteStyle=QuoteStyle.None]),
  Source2 = Csv.Document(File.Contents("C:Datafile2.csv"),[Delimiter=",", Encoding=1252, QuoteStyle=QuoteStyle.None]),
  Partition1 = Table.Partition(Source1, "PartitionColumn", [1,2,3]),
  Partition2 = Table.Partition(Source2, "PartitionColumn", [4,5,6]),
  Partitions = {Partition1, Partition2}
in
  Partitions
```

In this example, we first use the Csv.Document function to load the contents of each CSV file into a table. We then create two partitions

Table.FromRecords

What is Table.FromRecords Function?

Table.FromRecords is a Power Query M function used to create a new table from a list of records. A record is a collection of key-value pairs that represent a single row in a table. The function takes a list of records and converts it into a table with the columns specified in the keys.

The syntax for the Table.FromRecords function is as follows:

```
Table.FromRecords(records as list, optional columns as any)
```

The records parameter is a list of records, and the columns parameter is optional. If not specified, the function will use the keys from the first record in the list as the column names.

Understanding the M Code behind Table.FromRecords

To understand the M code behind the Table.FromRecords function, let's consider an example. Suppose we have a list of records as follows:

```
let
  records = {
    [Name="John", Age=25, Gender="Male"],
    [Name="Jane", Age=30, Gender="Female"],
    [Name="Mark", Age=35, Gender="Male"]
  }
in
  Table.FromRecords(records)
```

Table.FromRows

One of the most versatile M functions in Power Query is Table.FromRows. This function enables users to create a table from a list of rows. In this article, we will take a closer look at the M code behind this function and explore some examples of how it can be used.

Creating a Table from Rows

The Table.FromRows function takes a list of rows as its input and returns a table. Each row in the list is itself a list of values that correspond to the columns in the table. The syntax for the function is as follows:

```
Table.FromRows(rows as list, optional columns as any) as table
```

The ‘rows’ parameter is a list of lists, where each inner list represents a row in the resulting table. The optional ‘columns’ parameter allows users to specify the column names and data types for the table. If this parameter is not provided, Power Query will attempt to infer the column names and data types from the data itself.

Here is an example of how the function can be used to create a simple table:

```
let
  rows = { { 1, "John" }, { 2, "Jane" }, { 3, "Bob" } },
  table = Table.FromRows(rows)
in
  table
```

In this example, we have defined a list of rows that contains the values for the table. We then pass this list to the Table.FromRows function, which creates a table with two columns (‘Column1’ and ‘Column2’) and three rows.

Specifying Column Names and Data Types

As mentioned earlier, the Table.FromRows function can also accept a ‘columns’ parameter that allows users to specify the column

Table.FromValue

Overview of Table.FromValue Function

The Table.FromValue function is used to create a table in Power Query from a single value or a list of values. It takes two parameters: the first parameter is the value or list of values, and the second parameter is an optional string that represents the name of the column.

Syntax: `Table.FromValue(value as any, optional columnName as text)`

Example: `Table.FromValue(5, “Value”)`

This code will create a table with one column named “Value” and one row with the value of 5.

Understanding the M Code Behind Table.FromValue

The M code behind Table.FromValue is straightforward. Let’s take a look at the code:

```
let
  Source = #table({"Value"}, {{value}})
in
  Source
```

The code starts with the let statement, which is used to define a variable “Source.” The variable is assigned the value of the #table function. The #table function is used to create a table with the specified columns and rows.

In this case, the #table function is creating a table with one column named “Value” and one row with the value passed as the parameter in the Table.FromValue function.

The curly braces represent a list of column names, and the double curly braces represent a list of rows. Since we are passing only one value, there is only one row in the table.

Advanced Usage of Table.FromValue Function

The Table.FromValue function can be used in many ways, including creating a table from a list of values, creating a table with multiple columns, and creating a table with dynamic column names.

Creating a Table from a List of Values

To create a table from a list of values, you can pass the list of values as the first parameter in the Table.FromValue function. For

Table.FuzzyGroup

In this article, we'll take a closer look at the M code behind the Table.FuzzyGroup function, how it works, and how you can use it to improve your data analysis workflow.

Understanding Fuzzy Matching

Before we dive into the Table.FuzzyGroup function, it's important to understand what fuzzy matching is and why it's useful. Fuzzy matching is a technique used to identify strings that are similar to each other, even if they're not an exact match.

For example, let's say you have a column of customer names in your data, and some of the names are spelled slightly differently. Fuzzy matching can help you group those similar names together, even if they're not identical.

Fuzzy matching works by comparing strings based on a set of rules or algorithms. There are several algorithms that can be used for fuzzy matching, including the Levenshtein distance algorithm, which calculates the number of edits (insertions, deletions, and substitutions) needed to transform one string into another.

Introducing the Table.FuzzyGroup Function

The Table.FuzzyGroup function in Power Query uses a variant of the Levenshtein distance algorithm called the Jaro-Winkler distance algorithm. This algorithm compares two strings and returns a score between 0 and 1, where 0 means the strings are completely different and 1 means the strings are an exact match.

The Table.FuzzyGroup function takes a table as input and returns a new table with an additional column that groups similar strings together based on a fuzzy matching algorithm. The function takes two arguments: the name of the column to group, and a threshold value that determines how similar two strings need to be in order to be grouped together.

Here's an example of how to use the Table.FuzzyGroup function in Power Query:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i42Kc7ZxVc+zyrRUMfUz0/MLUItK1wbT1Y2tCQjI0N9UyNzJz1Sz
9JLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvM
Sy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0
u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSizJLzs0u1CvMSy8pSi
zJLz0tKLElMDS4tKTEoLilMDAwMDAwMDQwNDBiMThmNWEwMDA0BQ==", BinaryEncoding.Base64)), let _t = ((type nullable text) meta
Table.FuzzyJoin
```

What is Fuzzy Matching?

Fuzzy matching is a technique used to compare two strings of text and determine the likelihood that they represent the same thing. It is often used in data cleansing and data integration to join data from different sources with varying degrees of accuracy. Fuzzy matching algorithms consider factors such as spelling, phonetics, and synonyms to determine the similarity between two strings.

How does Table.FuzzyJoin Work?

Table.FuzzyJoin is an M function in Power Query that allows you to perform fuzzy matching on two tables. The function returns a new table that combines the rows from the two input tables based on a specified matching algorithm.

Here is the syntax for the Table.FuzzyJoin function:

```
Table.FuzzyJoin(
   table1 as table,
   key1 as any,
   table2 as table,
   key2 as any,
   algorithm as nullable number,
   optional options as nullable record
) as table
```

The function takes the following arguments:

– `table1` – The first table to join
– `key1` – The column or columns in `table1` to match
– `table2` – The second table to join
– `key2` – The column or columns in `table2` to match
– `algorithm` – The matching algorithm to use (optional)
– `options` – Additional options for the matching algorithm (optional)

Table.FuzzyNestedJoin

What is the Table.FuzzyNestedJoin Function?

The Table.FuzzyNestedJoin function is a type of join that matches two tables based on a fuzzy match of one or more columns. This function is similar to the regular Nested Join function in Power Query, but it allows for partial matches and can handle differences in casing, spelling, and other minor variations.

The Table.FuzzyNestedJoin function takes four arguments:

1. Table1: The first table to join.
2. JoinColumnName1: The name of the column in Table1 to use for the join.
3. Table2: The second table to join.
4. JoinColumnName2: The name of the column in Table2 to use for the join.

The function then compares the values in the two join columns and returns a new table that contains all the columns from both tables where there is a fuzzy match on the join columns.

The M Code Behind Table.FuzzyNestedJoin

The M code behind the Table.FuzzyNestedJoin function is complex and can be intimidating for beginners. However, understanding how this code works can help you to customize the function and apply it to your own datasets.

The code for the Table.FuzzyNestedJoin function is as follows:

```
let
  FuzzyJoin = (t1 as table, joincol1 as text, t2 as table, joincol2 as text) =>
    let
      CleanAndSplit = (str as text) =>
        Text.Split(Text.Replace(Text.Lower(str),"[^a-z]",""),"  "),
      StrToList = (str as text) =>
        List.Distinct(CleanAndSplit(str)),
      ListToTable = (list as list) =>
        Table.FromList(list, type table [Word = text]),
      TableToList = (table as table) =>
Table.Group
```

In this article, we will explore the M code behind the Table.Group function and how it can be used to transform data.

What is the Table.Group function?

The Table.Group function is used to group rows of data based on one or more columns. It takes an input table and a list of grouping columns as arguments. The output is a table with one row for each unique combination of grouping column values.

For example, let's say we have a table of sales data with columns for Region, Product, and Sales Amount. We want to know the total sales by region and product. We can use the Table.Group function to group the data by Region and Product and sum the Sales Amount column.

Syntax of the Table.Group function

The syntax of the Table.Group function is as follows:

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

– `table`: The input table to group
– `key`: A list of column names or expressions to group by
– `aggregatedColumns`: A list of columns to aggregate
– `groupKind`: An optional parameter to specify the grouping algorithm. The default is 0, which uses hash-based grouping.

Examples of using the Table.Group function

Let's look at some examples of using the Table.Group function.

Example 1: Grouping by a single column

Suppose we have a table of sales data with columns for Region, Product, and Sales Amount. We want to group the data by Region and find the total sales for each region.

The M code for this would be:

```
Table.Group(
Table.HasColumns
```

Understanding the Table.HasColumns Function

The Table.HasColumns function is a Boolean function that returns “true” if a table contains a specific column or a set of columns, and “false” otherwise. The syntax of the function is as follows:

```
Table.HasColumns(table as table, columns as any) as logical
```

In this syntax, “table” is the table that we want to check for the presence of a column, and “columns” can be a single column name or a list of column names.

The Table.HasColumns function is particularly useful when dealing with large datasets that contain multiple columns. Instead of manually scanning through each column to check for the presence of a specific column, we can use the Table.HasColumns function to automate the process.

The M Code Behind the Table.HasColumns Function

The Table.HasColumns function is written in M code, which is the language used by Power Query to perform data transformations. The M code behind the Table.HasColumns function is as follows:

```
(Table as table, columns as any) =>
let
  columnNames = if Type.Is(columns, type list) then columns else {columns},
  tableColumns = Table.ColumnNames(table),
  matches = List.Intersect(columnNames, tableColumns)
in
  List.Count(matches) = List.Count(columnNames)
```

Table.InsertRows

Syntax of the Table.InsertRows Function
The `Table.InsertRows` function has the following syntax:

```
Table.InsertRows(table as table, insert as list, optional options as nullable record) as table
```

The function takes three arguments:
– `table` – the table that you want to add the rows to
– `insert` – a list of rows that you want to add to the table
– `options` – an optional record that specifies additional options for the function
How to Use the Table.InsertRows Function
To use the `Table.InsertRows` function in Power Query, you first need to create a query that returns the table that you want to add rows to. You can do this by using the `From Table` option in the `From Other Sources` dropdown in the `Home` tab of the Power Query Editor.
Once you have the table loaded into Power Query, you can use the `Table.InsertRows` function in a subsequent step to add rows to the table. To do this, you need to create a list of records that represent the rows that you want to add to the table.
Here is an example of how to use the `Table.InsertRows` function to add a single row to a table:

```
let
  Source = Excel.CurrentWorkbook(){[Name="Table1"]}[Content],
  NewRow = [Column1="NewValue1", Column2="NewValue2"],
  InsertedRows = Table.InsertRows(Source, {NewRow})
in
  InsertedRows
```

Table.IsDistinct

What is the Table.IsDistinct M function?

The Table.IsDistinct M function is a Power Query function that takes a table as its input and returns a table that contains only the unique rows from the original table. This means that any rows that contain identical values in all columns will be removed, leaving only one instance of each unique row.

The syntax for the Table.IsDistinct function is as follows:

```
Table.IsDistinct(table as table, optional equationCriteria as any, optional columnsToCompare as any)
```

The first parameter, `table`, is the table that you want to check for duplicates. The second parameter, `equationCriteria`, is an optional parameter that allows you to specify the criteria for comparing rows. Finally, the third parameter, `columnsToCompare`, is also an optional parameter that allows you to specify which columns should be used in the comparison.

How does the Table.IsDistinct M function work?

The Table.IsDistinct M function works by first creating a list of all the rows in the original table. It then compares each row in this list to every other row, using the specified criteria and columns. If two rows are found to be identical, only one of them is kept in the final output table.

To illustrate this, consider the following example:

```
let
  Source = Table.FromRows({{"John", "Doe", 25}, {"John", "Doe", 25}, {"Jane", "Doe", 30}}, {"First Name", "Last Name", "Age"}),
  DistinctTable = Table.IsDistinct(Source)
in
  DistinctTable
```

Table.IsEmpty

The Table.IsEmpty function returns true if the table is empty or false if it contains data. In this article, we will discuss the M code behind this function and how it can be used in Power Query.

Understanding Table.IsEmpty Function

The Table.IsEmpty function is used to determine whether a table contains data or not. It takes a table as an argument and returns a boolean value. The function returns true if the table is empty and false if it contains data.

The M code behind the Table.IsEmpty function is as follows:

```
(Table as table) =>
let
  Source = Table.RowCount(Table.SelectRows(Table.Distinct(Table.Buffer(Table)), each true)),
  Result = if Source = 0 then true else false
in
  Result
```

Breaking Down the M Code

Let's break down the M code behind the Table.IsEmpty function and understand how it works.

The first line of the M code `(Table as table) =>` is the function signature. It defines the input argument `Table` as a table.

The second line `let` starts a new block of code. This block of code is used to declare variables and create expressions.

The third line `Source = Table.RowCount(Table.SelectRows(Table.Distinct(Table.Buffer(Table)), each true))` creates a variable called `Source`. This variable counts the number of rows in the table and selects the rows that are not empty. This is done by using the Table.SelectRows function with the `each true` condition.

The fourth line `Result = if Source = 0 then true else false` creates a variable called `Result`. This variable determines whether the table is empty or not. If the `Source` variable is equal to 0, it means that the table is empty and the function returns true. Otherwise, the function returns false.

The final line `in Result` is used to return the value of the `Result` variable.

Table.Join

What is Table.Join?

Table.Join is a Power Query M function that allows you to join two or more tables based on a common column. The function takes two or more tables as input, and returns a new table that combines the columns from each input table. You can specify the type of join to use (left, right, inner, or full outer), as well as the columns to include in the output table.

The M Code Behind Table.Join

To use the Table.Join function in Power Query, you need to understand the M code behind the function. The M code is a functional programming language used to create queries in Power Query. Here's an example of the M code for a simple Table.Join function:

```
let
  Source1 = Table.FromRows({{"John", 25}, {"Jane", 30}}, {"Name", "Age"}),
  Source2 = Table.FromRows({{"John", "New York"}, {"Jane", "Los Angeles"}}, {"Name", "City"}),
  JoinTable = Table.Join(Source1, "Name", Source2, "Name")
in
  JoinTable
```

In this example, we have two tables: Source1 and Source2. Source1 contains two columns: Name and Age, while Source2 contains two columns: Name and City. We want to join these tables based on the common column "Name". The M code for the Table.Join function is:

```
Table.Join(Source1, "Name", Source2, "Name")
```

This code tells Power Query to join the tables Source1 and Source2 on the column "Name". The output table will contain columns from both input tables, with matching rows joined together.

Types of Joins

Table.Keys

What are Keys in Power Query?

In Power Query, a key is a column or set of columns that uniquely identifies each row in a table. Keys are used to create relationships between tables, allowing users to combine data from multiple tables into a single view. When two tables are related, the key column(s) in the primary table (the one that contains the unique values) is used to match the corresponding rows in the related table.

What is the Table.Keys M Function?

The Table.Keys M function is a built-in function in Power Query that allows users to create and manage keys within their data sets. The function takes a table as its input and returns a table with the key columns identified. The syntax for the Table.Keys M function is as follows:

```
Table.Keys(table as table, keyColumns as list) as table
```

The table parameter is the input table, while the keyColumns parameter is a list of the column names that should be used as keys. If multiple columns are specified, the keys are created using a combination of the values in those columns.

How to Use Table.Keys to Create Keys

To create keys using the Table.Keys M function, follow these steps:

1. Open the Power Query Editor in Excel or Power BI.
2. Load the table that you want to create keys for.
3. Click on the table to select it.
4. Go to the “Add Column” tab and click on “Custom Column”.
5. In the “Custom Column” dialog box, enter a name for the new column (e.g. “Key”) and enter the following formula:

```
= Table.Keys(#"Previous Step", {"Column1", "Column2"})
```

Table.Last

In this article, we will explore the M code behind the Table.Last function in Power Query M, including its syntax, usage, and examples.

Understanding the Table.Last Function

The Table.Last function in Power Query M is a function used to extract the last row of a table. It takes a table as its input and outputs a record with the values from the last row of the table.

The syntax for the Table.Last function is as follows:

```
Table.Last(table as table) as record
```

In this syntax, the table parameter is the input table, and the function returns a record with the values from the last row of the table.

Examples of the Table.Last Function

To better understand the Table.Last function, let's explore some examples of how it can be used in Power Query M.

Example 1: Extracting the Last Row of a Table

Suppose we have a table with the following data:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMlSK1YlWMgKyklPw0GJiBmI2PQzKz0zPSdJSSnILUxRiEjMgA==", BinaryEncoding.Base64)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Column1 = _t, Column2 = _t])),
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"Column1", Int64.Type}, {"Column2", Int64.Type}})
in
  #"Changed Type"
```

To extract the last row of this table, we can use the Table.Last function as follows:

```
Table.LastN
```

In this article, we will take a closer look at the M code behind the Table.LastN function and how it can be used to improve your data analysis.

What is the Table.LastN Function?

The Table.LastN function is a Power Query function that allows you to extract the last n rows of a table. It takes two arguments: the table to extract the rows from, and the number of rows to extract.

Here is a basic example of how to use the Table.LastN function:

```
let
  Source = Excel.CurrentWorkbook(){[Name="Table1"]}[Content],
  LastNRows = Table.LastN(Source, 10)
in
  LastNRows
```

In this example, we are extracting the last 10 rows from the table named “Table1”. The resulting table will contain only the last 10 rows of the original table.

How Does the Table.LastN Function Work?

The Table.LastN function works by first determining the total number of rows in the input table. It then subtracts the number of rows to extract from this total to determine the index of the first row to keep. Finally, it filters out all rows before this index.

Here is the M code for the Table.LastN function:

```
(Table as table, CountOrCondition as any) as table =>
  let
    Count = if Type.Is(CountOrCondition, type number) then
      CountOrCondition
Table.MatchesAllRows
```

Table.MatchesAllRows is a function that compares two tables and returns all the rows in the first table that match all the rows in the second table. This function is particularly useful when you need to merge two tables based on multiple columns. In this article, we will take a closer look at the M code behind Table.MatchesAllRows and how to use it effectively.

The Syntax of Table.MatchesAllRows

The syntax of Table.MatchesAllRows is straightforward and easy to understand. It takes two parameters: the first parameter is the table to search, and the second parameter is the table to match. Here is the basic syntax of Table.MatchesAllRows:

```
Table.MatchesAllRows(table1 as table, table2 as table) as table
```

Table.MatchesAllRows returns a table that contains all the rows in table1 that match all the rows in table2. The matching is based on the column values in both tables, and the order of the columns does not matter.

The M Code Behind Table.MatchesAllRows

The M code behind Table.MatchesAllRows is not complicated, but it is essential to understand how it works to use it effectively. Here is the M code behind Table.MatchesAllRows:

```
let
  matches = Table.NestedJoin(table1, {key_columns}, table2, {key_columns}, "matched"),
  filtered = Table.SelectRows(matches, each List.Contains(List.Transform(Table.Column(_,[matched]), each Record.Field(_, key_columns)), true))
in
  filtered
```

The code above consists of two basic steps: matching and filtering. Let’s break down each step:

Table.MatchesAnyRows

Understanding the Table.MatchesAnyRows Function

Before we dive into the M code, let's first take a quick look at what the Table.MatchesAnyRows function does. This function takes two arguments: a table and a condition. It then returns a Boolean value that indicates whether the table contains any rows that match the condition.

Here is an example of how to use this function:

```
let
  Source = Table.FromRows({{"John", "Doe"}, {"Jane", "Doe"}, {"Bob", "Smith"}}, {"First Name", "Last Name"}),
  MatchesAny = Table.MatchesAnyRows(Source, each [First Name] = "John")
in
  MatchesAny
```

In this example, we create a table called "Source" with three rows and two columns: "First Name" and "Last Name". We then use the Table.MatchesAnyRows function to check if the table contains any rows where the "First Name" column equals "John". Since the table does contain such a row, the function returns True.

The M Code Behind Table.MatchesAnyRows

Now that we understand what the Table.MatchesAnyRows function does, let's take a closer look at the M code behind it. Here is the M code for the function:

```
(Table as table, Condition as function) as logical =>
  List.Contains(
    List.Transform(
      Table.ToList(Table.SelectColumns(Table.AddColumn(Table.DemoteHeaders(Table), "Index", each Table.PositionOf(Table, _){0}),
"Index")),
Table.Max
```

In this article, we will explore the M code behind the Table.Max function and how it can be used to transform data in Power Query.

Overview of the Table.Max Function

The Table.Max function is used to find the maximum value in a column of a table. It takes two arguments: the table and the name of the column. Here is the syntax for the Table.Max function:

```
Table.Max(table as table, columnName as text, optional comparer as nullable function) as any
```

The first argument, `table`, is the table that contains the column to be evaluated. The second argument, `columnName`, is the name of the column to be evaluated. The third argument, `comparer`, is an optional argument used to specify a custom comparison function.

The M Code Behind Table.Max Function

The Table.Max function is implemented in M code, which is the programming language used in Power Query. The M code for the Table.Max function is as follows:

```
(table as table, columnName as text, optional comparer as nullable function) =>
  let
    column = Table.Column(table, columnName),
    max = List.Max(column, comparer)
  in
    max
```

The M code starts by defining the function and its arguments. The code then uses the `Table.Column` function to extract the column from the table. The `List.Max` function is then used to find the maximum value in the column. Finally, the maximum value is returned as the output of the function.

Table.MaxN

Introduction to Table.MaxN

The Table.MaxN function is used to return the top N rows from a table based on a specific column. This function is particularly useful when working with large datasets where you need to filter out the top N rows based on a specific criteria. The syntax for the Table.MaxN function is as follows:

```
Table.MaxN(table as table, count as number, optional comparisonCriteria as any) as table
```

The first argument of this function is the table that you want to filter. The second argument is the number of rows that you want to return. The third argument is optional and is used to specify the criteria that you want to use for filtering the rows.

Understanding the M Code Behind Table.MaxN

When you use the Table.MaxN function in Power Query, the M code behind it is automatically generated. However, it is useful to understand the M code behind the function in order to customize it for your specific needs. The M code for the Table.MaxN function is as follows:

```
let
  Source = table,
  Column = Table.Column(Source, column),
  SortList = List.Sort(Column, comparisonCriteria, Order.Descending),
  TakeNRows = List.FirstN(SortList, count),
  FilterTable = Table.SelectRows(Source, each List.Contains(TakeNRows, Table.Column(Source, column)))
in
  FilterTable
```

Table.Min

What is Power Query?

Power Query is a data transformation and cleansing tool that is a part of the Microsoft Power BI suite. It is used to extract, transform, and load data from various sources into a structured format that can be analyzed and visualized. The Power Query Editor provides a user-friendly interface that allows you to perform a wide range of data transformations using a simple drag-and-drop approach.

Understanding the Table.Min Function

The Table.Min function is a part of the M language that is used in Power Query. It is used to find the minimum value of a specific column in a table. The syntax for the Table.Min function is as follows:

```
Table.Min(table as table, column as text) as any
```

The first argument, table, is the table that you want to search for the minimum value. The second argument, column, is the name of the column that you want to search. The function returns the minimum value of the specified column.

How to Use the Table.Min Function

To use the Table.Min function, you need to follow these steps:

1. Open the Power Query Editor.
2. Click on the Home tab.
3. Click on the New Source option.
4. Select the type of data source that you want to use.
5. Connect to the data source and import the data into Power Query.
6. In the Power Query Editor, select the table that you want to search.
7. Click on the Add Column tab.
8. Click on the Custom Column option.
9. In the Custom Column dialog box, enter a name for the new column.
10. In the Custom Column dialog box, enter the following formula:

```
Table.MinN
```

Understanding the Table.MinN function

Before delving into the M code behind the Table.MinN function, it is important to understand how this function works. The syntax of the function is as follows:

```
Table.MinN(Table as table, count as number, optional comparisonCriteria as any) as table
```

The first argument is the table from which the N smallest values will be returned. The second argument is the count of the smallest values to return. The optional third argument is the comparison criteria used to determine the smallest values. If this argument is not provided, the function will use the default comparison criteria.

The M code behind the Table.MinN function

The M code behind the Table.MinN function is not as complicated as one might think. The code is designed to sort the table in ascending order based on the comparison criteria and then return the top N values. Here is the M code for the Table.MinN function:

```
(Table as table, count as number, optional comparisonCriteria as any) as table =>
let
  sortedTable = Table.Sort(Table, comparisonCriteria, Order.Ascending),
  minNRows = Table.FirstN(sortedTable, count)
in
  minNRows
```

As you can see, the code is quite straightforward. The table is first sorted in ascending order based on the comparison criteria. The first N rows of the sorted table are then returned as the result.

Examples of using the Table.MinN function

Table.NestedJoin

In this article, we will explore the M code behind the Table.NestedJoin function and provide examples of how it can be used to solve real-world data problems.

What is Table.NestedJoin?

Table.NestedJoin is a function in Power Query M that allows you to join two tables together based on a common column or set of columns. The difference between Table.NestedJoin and traditional SQL join is that it can handle more complex data structures, such as nested tables and lists.

Table.NestedJoin takes four arguments:

– The first argument is the first table you want to join.

– The second argument is the column or columns you want to join on.

– The third argument is the second table you want to join.

– The fourth argument is the join kind, which specifies how the tables should be joined.

The M Code Behind Table.NestedJoin

To understand how Table.NestedJoin works, let's take a look at the M code behind the function.

```
= Table.NestedJoin(
   table1 as table,
   joinColumn as text or list of text or record,
   table2 as table,
   joinKind as nullable number
)
```

The first argument, `table1`, is the first table you want to join. This can be any table in your Power Query workspace.

The second argument, `joinColumn`, is the column or columns you want to join on. This can be a single column name, a list of column names, or a record that defines the join keys.

The third argument, `table2`, is the second table you want to join. This can be any table in your Power Query workspace.

Table.Partition

What is Table.Partition?

Table.Partition is an M function that allows you to split a large table into smaller tables based on a specified partitioning column. This partitioning column is used to group the data in the original table, creating smaller tables that can be processed more efficiently. This function is commonly used when working with large datasets that would otherwise take a long time to load or manipulate.

The M Code Behind Table.Partition

The M code for Table.Partition is relatively straightforward. Here's an example that partitions a table called "Sales" based on the "Region" column:

```
let
  Source = Excel.CurrentWorkbook(){[Name="Sales"]}[Content],
  Partitioned = Table.Partition(Source, "Region")
in
  Partitioned
```

In this code, the "Source" variable refers to the original table. The "Table.Partition" function is then applied to the "Source" table, with "Region" specified as the partitioning column. The resulting table is then stored in the "Partitioned" variable.

How to Use Table.Partition Effectively

Now that we understand the M code behind Table.Partition let's explore how to use it effectively. Here are some tips:

Choose the Right Partitioning Column

The key to using Table.Partition effectively is to choose the right partitioning column. This column should have a relatively low number of distinct values, so the resulting tables are not too small or too large. You should also consider the data type of the partitioning column, as some data types may work better than others.

Use Parallel Processing

When processing large datasets, it's essential to take advantage of parallel processing to speed up the process. Table.Partition automatically parallelizes the data processing across multiple CPU cores if your computer has them. This can significantly improve the

Table.PartitionValues

What is Table.PartitionValues?

Table.PartitionValues is a M function that returns a table containing the distinct values of a specified column in a given table. The function takes two arguments: the table to partition and the name of the column to partition. Here is the syntax:

```
Table.PartitionValues(table as table, column as text)
```

The function returns a table with a single column and one row for each distinct value in the specified column of the input table.

Understanding the M Code Behind Table.PartitionValues

The M code behind Table.PartitionValues is relatively simple. The function uses the Table.Group function to group the input table by the specified column. Here is the M code:

```
let
  PartitionValues = (table as table, column as text) =>
  let
    GroupedTable = Table.Group(table, {column}, {{"Count", each _, type table [#"Column1":type]}})
  in
    GroupedTable[Column1]
in
  PartitionValues
```

The function takes the input table and column name as arguments and assigns them to the variables “table” and “column”, respectively. The Table.Group function is then used to group the input table by the specified column. The resulting table is assigned to the variable “GroupedTable”.

Table.Pivot

Pivoting is the process of converting rows of data into columns. This process is useful when you want to summarize data in a more meaningful way. The Table.Pivot M function in Power Query helps to simplify this process.
Understanding The Table.Pivot Function
The Table.Pivot function transforms a table into a new table by pivoting the values from a specified column into new columns. The function allows you to specify the aggregation function to apply to the values that are pivoted.
Syntax:

Table.Pivot(Table as table, pivotColumn as text, valueColumn as text, [aggregationFunction], [additionalAggregationFunctions], [groupByColumns])

Parameters
– Table – The input table to be pivoted.
– pivotColumn – The name of the column whose values will be pivoted.
– valueColumn – The name of the column that contains the values to be aggregated.
– aggregationFunction – (optional) The aggregation function to apply to the values being pivoted. The default aggregation function is “sum”.
– additionalAggregationFunctions – (optional) The additional aggregation functions to apply to the values being pivoted.
– groupByColumns – (optional) The columns to group by.
Examples
Example 1: Basic Usage
Suppose we have a table of sales data that looks like this:
Product	Year	Quarter	Sales
———	——	———	——-
A	2020	Q1	100
A	2020	Q2	200

Table.PositionOf

What is Table.PositionOf?

Table.PositionOf is an M function in Power Query that enables you to determine the position or index of a given value within a table column. The function takes two arguments; the first argument is the table column you want to search, while the second is the value you want to find. Table.PositionOf returns the index of the first occurrence of the value in the table column.

Syntax

The syntax for Table.PositionOf is as follows:

```
Table.PositionOf(table as table, value as any, optional equationCriteria as any) as nullable number
```

Parameters

Table.PositionOf takes the following parameters:

– table: The table to search

– value: The value to search for

– optional equationCriteria: A comparison operator to use to match the value. This parameter is optional.

How to use Table.PositionOf

You can use the Table.PositionOf function in Power Query to determine the index of a value in a table column. Here is an example of how to use Table.PositionOf:

Suppose you have a table named Sales that contains columns for Product, Region, and Sales Quantity. To find the position of the value “East” in the Region column, you can use the following M code:

```
let
  Source = Sales,
  Position = Table.PositionOf(Source[Region], "East")
in
```

Table.PositionOfAny

In this article, we will dive deep into the M code behind the Table.PositionOfAny function and explore how it can be used to extract data from tables in Power Query.

Understanding the Table.PositionOfAny Function

The Table.PositionOfAny function is used to find the position of a value within a column of a table. This function takes two parameters: the column and the list of values to search for.

The syntax for the Table.PositionOfAny function is as follows:

```
Table.PositionOfAny(table as table, column as text, values as list, optional missingValue as nullable number) as list
```

– `table`: The table to search for values.
– `column`: The name of the column to search.
– `values`: The list of values to search for.
– `missingValue`: (Optional) The value to return if the searched value is not found.

The function returns a list of positions for each value found in the searched column.

How to Use Table.PositionOfAny in Power Query

To use the Table.PositionOfAny function in Power Query, we need to create a query that returns a table with the column we want to search.

Let's say we have a table named "Sales" with the following columns: "Date", "Product", "Sales Amount". We want to find the position of the first occurrence of the values "Apples" or "Oranges" in the "Product" column.

To accomplish this, we need to create a new query that references the "Sales" table and returns only the "Product" column:

```
let
  Source = Sales,
  Product = Source[Product]
```

Table.PrefixColumns

What is Table.PrefixColumns?

Table.PrefixColumns is a built-in M function in Power Query that allows users to add a prefix to column names in a table. The function takes two arguments: the table to be transformed and the prefix string to be added to the column names. The function returns a new table with the prefixed column names.

The M Code Behind Table.PrefixColumns

To understand the M code behind Table.PrefixColumns, we need to break down the syntax of the function. The syntax is as follows:

```
Table.PrefixColumns(table as table, prefix as text)
```

The table argument represents the input table to be transformed, while the prefix argument represents the text string to be added as a prefix to the column names. Let's examine the M code behind Table.PrefixColumns more closely.

Step 1: Identify the Table to be Transformed

The first step in the M code of Table.PrefixColumns is to identify the table to be transformed. This is done using the table as table argument in the function. The table can be an existing table in Power Query or a result of a previous transformation.

Step 2: Extract the Column Names

The next step is to extract the column names from the input table. This is done using the Table.ColumnNames function in M. The Table.ColumnNames function returns a list of column names in the input table.

```
columnNames = Table.ColumnNames(table)
```

Step 3: Prefix the Column Names

The third step in the M code of Table.PrefixColumns is to prefix the column names with the specified prefix string. This is done using the List.Transform function in M. The List.Transform function applies a transformation function to each element of a list and returns a new list.

```
prefixedColumnNames = List.Transform(columnNames, each prefix & _)
```

The above code prefixes each column name in the columnNames list with the prefix string. The underscore (_) represents the current element in the list being transformed.

Step 4: Rename the Table Columns

The final step in the M code of Table.PrefixColumns is to rename the table columns with the prefixed column names. This is done using the Table.RenameColumns function in M. The Table.RenameColumns function renames the columns in a table based on a list of old and new column names.

Table.Profile

What is the Power Query M Language?

Power Query uses a language called M to perform data transformations. M is a functional programming language that is used to build the queries that Power Query executes. The language is based on a set of functions and expressions that can be combined to perform complex data manipulations.

What is the Table.Profile Function?

The Table.Profile function is a built-in function in Power Query that provides a quick and easy way to analyze data. It returns a table that contains a profile of the data in the input table. The profile includes statistics such as the total number of rows, the number of distinct values, and the data type of each column.

Understanding the M Code Behind Table.Profile

The Table.Profile function is essentially a wrapper for a series of other M functions. Understanding how these functions work together can help you create more powerful data transformations in Power Query.

The M code for the Table.Profile function is as follows:

```
let
  Source = ,
  Types = Table.TransformColumnTypes(Source, Table.TransformColumnTypesOptions.Ignore),
  Stats = Table.Group(Types, {}, {{"Column", each Text.Combine(List.Transform(Table.ColumnNames(_), Text.From), ", "), type text},
                {"Count", each Table.RowCount(_), type number},
                {"Distinct Count", each List.Count(List.Distinct(Table.Column(_,"Value"))), type number},
                {"Null Count", each List.Count(Table.SelectRows(_, each Record.Field(_, "Value") = null), type number},
                {"Min", each List.Min(Table.Column(_, "Value")), type any},
                {"Max", each List.Max(Table.Column(_, "Value")), type any},
                {"Average", each List.Average(Table.Column(_, "Value")), type number},
                {"Median", each List.Median(Table.Column(_, "Value")), type any},
                {"Standard Deviation", each List.StandardDeviation(Table.Column(_, "Value")), type number}})
in
Table.PromoteHeaders
```

What is Table.PromoteHeaders?

Table.PromoteHeaders is a Power Query M function that allows you to promote the first row of a table to the header row. When a table is created in Power Query, the first row is typically assumed to be the header row. However, this may not always be the case. Table.PromoteHeaders allows you to specify which row should be promoted to the header row, which can be especially useful when dealing with tables that have multiple header rows.

How to Use Table.PromoteHeaders

To use Table.PromoteHeaders in Power Query, follow these steps:

1. Open the Power Query editor by selecting “Edit Queries” from the “Home” tab in Excel.
2. Select the table you want to modify.
3. From the “Transform” tab, select “Use First Row As Headers”.
4. If the first row is not the header row, select “Use Headers As First Row” instead.
5. If the table has multiple header rows, select “Use Headers As First Rows” and specify the number of rows to promote.
6. Once you have selected the appropriate option, the header row will be promoted and the table will be updated accordingly.

The M Code Behind Table.PromoteHeaders

The M code behind Table.PromoteHeaders is relatively simple. When you select “Use First Row As Headers”, Power Query generates the following M code:

```
= Table.PromoteHeaders(#"Changed Type")
```

This code takes the table that has been modified using the “Changed Type” function and promotes the first row to the header row. If you select “Use Headers As First Row” instead, the M code will look like this:

```
= Table.PromoteHeaders(#"Changed Type", [PromoteAllScalars=true])
```

Table.Range

Understanding Table.Range

The Table.Range function extracts a subset of rows from a table based on a start index and a count. The function has two arguments: the table that needs to be sliced, and the range of rows to be extracted. The range is defined by the start index, which is the position of the first row to be extracted, and the count, which is the number of rows to extract from the start index.

The syntax to use the Table.Range function is as follows:

```
Table.Range(Table as table, start as number, count as number)
```

Here, Table is the name of the table that needs to be sliced, and Start and Count are the parameters that define the range of rows to be extracted.

Examples

Let's take a look at some examples of how Table.Range can be used.

Example 1: Extracting the first five rows of a table

Suppose we have a table named "SalesData" that contains sales data for a company. We want to extract the first five rows of the table. We can use the following formula:

```
Table.Range(SalesData, 0, 5)
```

The first argument is the name of the table "SalesData", the second argument is the start index, which is 0, and the third argument is the count, which is 5. This formula will extract the first five rows of the SalesData table.

Example 2: Extracting rows from a specific index

Suppose we want to extract rows from the SalesData table starting from the third row. We can use the following formula:

Table.RemoveColumns

Introduction to Table.RemoveColumns Function

The Table.RemoveColumns function is used to remove a specified list of columns from a table in Power Query. The syntax for this function is as follows:

```
Table.RemoveColumns(table as table, columnNames as list) as table
```

Here, `table` refers to the input table from which the columns need to be removed, and `columnNames` is a list of column names that need to be removed. The output of this function is a new table with all the specified columns removed.

Understanding the M Code Behind Table.RemoveColumns Function

The Power Query uses a functional language called M to perform data transformations. The Table.RemoveColumns function is a built-in M function that can be customized based on user requirements. Let's take a look at the M code behind the Table.RemoveColumns function.

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMlQyUFJSgjJwNDYyUDM2LzS2MDEwVayCUB",
BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Column1 = _t,
Column2 = _t, Column3 = _t]),
  RemovedColumns = Table.RemoveColumns(Source,{"Column1", "Column2"})
in
  RemovedColumns
```

Here, the M code first creates a source table from a compressed binary value. The source table contains three columns named

Table.RemoveFirstN

One of the most useful functions in Power Query is the Table.RemoveFirstN function, which allows users to remove a specified number of rows from the beginning of a table. This function is particularly useful when working with data that has headers or other metadata that needs to be removed before analysis.

In this article, we will take a closer look at the M code behind the Table.RemoveFirstN function, and explore some of the ways in which it can be used to streamline data analysis workflows.

Understanding the M Code Behind Table.RemoveFirstN

In Power Query, all data transformation tasks are accomplished through the use of M code. M is a functional language that is used to define queries and transformations in Power Query.

The Table.RemoveFirstN function is a built-in M function that allows users to remove a specified number of rows from the beginning of a table. The function takes two arguments: the table to be modified, and the number of rows to remove.

Here is an example of the basic syntax of the Table.RemoveFirstN function:

```
Table.RemoveFirstN(table as table, count as number) as table
```

In this example, the “table” argument represents the table that you want to modify, while the “count” argument represents the number of rows that you want to remove.

For example, if you have a table that contains headers in the first two rows, and you want to remove these headers, you could use the following M code:

```
Table.RemoveFirstN(#"Original Table", 2)
```

In this code, the “#” before the table name indicates that the table is a reference to a previous step in the query. The “2” indicates that we want to remove the first two rows of the table.

Table.RemoveLastN

What is the Table.RemoveLastN Function?

The Table.RemoveLastN function is a Power Query function that allows users to remove a specified number of rows from the end of a table. This function is useful when working with large tables and you only need to use a portion of the data. By removing unnecessary rows at the end of the table, you can reduce the size of the table and improve performance.

Understanding the M Code Behind Table.RemoveLastN

The M code behind the Table.RemoveLastN function is fairly simple. This function takes two arguments: the table you want to modify and the number of rows you want to remove from the end of the table. Here is the M code for the Table.RemoveLastN function:

```
(Table as table, CountOrCondition as any) as table =>
  let
    Count = if Value.Is(Value.Type(CountOrCondition), type number) then CountOrCondition else
Table.RowCount(Table.Buffer(Table.SelectRows(Table, CountOrCondition))),
    Result = if Count >= Table.RowCount(Table) then #table({}, {}) else Table.FirstN(Table, Table.RowCount(Table) - Count)
  in
    Result
```

Let’s break down this code to understand how it works.

Arguments

The first line of the M code defines the arguments for the Table.RemoveLastN function. The function takes two arguments:

– Table: The table you want to modify

– CountOrCondition: The number of rows you want to remove from the end of the table, or a function that returns a logical value for each row in the table

Calculating the Count

The next few lines of code calculate the number of rows to remove from the end of the table. The CountOrCondition argument can either be a number or a function that returns a logical value for each row in the table. If it is a number, then that number is used as the count. If

Table.RemoveMatchingRows

Understanding the Table.RemoveMatchingRows Function

The Table.RemoveMatchingRows function is used to remove rows from a table that match a certain criteria. The syntax for this function is as follows:

```
Table.RemoveMatchingRows(table as table, condition as function) as table
```

In this syntax, the first argument (table as table) is the input table from which the rows need to be removed. The second argument (condition as function) is a function that specifies the criteria for removing rows. The function takes a single argument (a row from the input table) and returns a boolean value that determines whether the row should be removed or not.

The Table.RemoveMatchingRows function works by iterating through each row of the input table and evaluating the condition function for that row. If the condition function returns true, the row is removed from the table. If the condition function returns false, the row is kept in the table.

Customizing the Table.RemoveMatchingRows Function

The Table.RemoveMatchingRows function can be customized to fit specific data cleaning needs by creating a custom condition function. This function can be written in M code and can be as simple or as complex as needed.

For example, suppose we have a table of customer data that includes a column for customer age. We want to remove all rows where the customer age is less than 18. To do this, we can create a custom condition function that checks the value of the customer age column for each row and returns true if the value is less than 18, and false if the value is greater than or equal to 18.

The code for this custom condition function would look like this:

```
(customer) => customer[age] < 18
```

We can then pass this function as the second argument to the Table.RemoveMatchingRows function to remove all rows where the

Table.RemoveRows

What is Power Query?

Power Query is a data transformation and cleaning tool that is part of Microsoft Excel and Microsoft Power BI. It allows users to connect to various data sources, clean and transform data, and load it into Excel or Power BI for analysis. Power Query uses a functional language called M to define data transformations.

Understanding the Table.RemoveRows Function

The Table.RemoveRows function is used to remove rows from a table based on specified criteria. The syntax for this function is as follows:

```
Table.RemoveRows(table as table, rows as any, optional missingField as nullable number) as table
```

– table: This is the table that you want to remove rows from.
– rows: This is the criteria that you want to use to remove the rows. It can be a list of values, a function, or a logical expression.
– missingField: This is an optional parameter that specifies how to handle missing values. If set to null, missing values will be ignored.

The Table.RemoveRows function returns a new table with the specified rows removed.

The M Code Behind Table.RemoveRows

The M code for Table.RemoveRows is as follows:

```
let
  RemoveRows = (table as table, rows as any, optional missingField as nullable number) as table =>
    let
      columnNames = Table.ColumnNames(table),
      #"Filtered Rows" = Table.SelectRows(table, each not List.Contains(rows, Record.ToList(_))),
      #"Reordered Columns" = Table.ReorderColumns(#"Filtered Rows", columnNames)
    in
Table.RemoveRowsWithErrors
```

What is Table.RemoveRowsWithErrors?

Table.RemoveRowsWithErrors is a function in Power Query M that allows you to remove rows from a table that contain errors. The function takes a table as its input and returns a new table with the rows that contain errors removed.

The function is particularly useful when dealing with large datasets that contain errors or invalid data. By removing these rows, you can ensure that your analysis is based on clean and accurate data.

How to Use Table.RemoveRowsWithErrors

To use the Table.RemoveRowsWithErrors function in Power Query M, you first need to load your data into Power Query. Once you have done this, you can follow these steps:

1. Select the column that contains the errors you want to remove.
2. Click on the “Remove Errors” button in the “Transform” tab.
3. Choose the option “Remove Rows with Errors”.
4. Click on “OK”.

Once you have completed these steps, Power Query M will automatically generate the M code for the Table.RemoveRowsWithErrors function. You can then edit the code if necessary or simply click “Close & Apply” to apply the changes to your data.

The M Code Behind Table.RemoveRowsWithErrors

The M code behind the Table.RemoveRowsWithErrors function is relatively simple and consists of a single line of code:

```
Table.RemoveRowsWithErrors(table as table, optional errorHandling as nullable number) as table
```

Here’s a breakdown of what each part of the code does:

– `Table.RemoveRowsWithErrors` is the name of the function.

– `table as table` specifies that the input to the function is a table.

– `optional errorHandling as nullable number` indicates that the function has an optional parameter that allows you to specify how errors should be handled. This parameter is not required and can be left blank.

Customizing the Error Handling Behavior

Table.RenameColumns

Understanding the Table.RenameColumns Function

In Power Query, the Table.RenameColumns function is used to rename the columns of a table. The syntax of the function is as follows:

```
Table.RenameColumns(table as table, columnNames as list, newNames as list)
```

The function takes three arguments:

– `table`: this is the table whose columns you want to rename.
– `columnNames`: this is a list of the current column names.
– `newNames`: this is a list of the new column names.

For example, suppose we have a table with three columns named “Name”, “Age”, and “Gender”. We can rename these columns using the Table.RenameColumns function as follows:

```
Table.RenameColumns(
  Source,
  {"Name", "Age", "Gender"},
  {"Full Name", "Years", "Sex"}
)
```

In this example, we are renaming the “Name” column to “Full Name”, the “Age” column to “Years”, and the “Gender” column to “Sex”.

The M Code Behind the Table.RenameColumns Function

The Table.RenameColumns function is a built-in function in Power Query, and its M code is not visible by default. However, you can view the M code by clicking on the “Advanced Editor” button on the “View” tab in Power Query.

When you click on the “Advanced Editor” button, the Power Query Editor will show you the M code for the current query. The M code for

Table.ReorderColumns

What is Table.ReorderColumns?

Table.ReorderColumns is a Power Query M function that rearranges the order of columns in a table. This function allows users to customize the order of columns in a table based on their needs.

Understanding the M Code Behind Table.ReorderColumns

The M code behind Table.ReorderColumns is relatively straightforward. It essentially reorders the list of column names in a table based on the user's input.

Here's an example of the M code:

```
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMjIwtjQ0MjQ2NjY3Mq1TS0lWitWJVItKzUxNrA0MMY0NAMz1DMyMDQ0MDQ0MAQa", BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [Column1 = _t, Column2 = _t, Column3 = _t, Column4 = _t]),
  ReorderedColumns = Table.ReorderColumns(Source,{"Column4", "Column1", "Column2", "Column3"})
in
  ReorderedColumns
```

In this example, we have a table with four columns, and we want to reorder them based on our preferences. We start by creating a Source table with the initial column order. We then apply the Table.ReorderColumns function to the Source table, specifying the new column order we want. The result is a new table with the columns reordered as per our preferences.

How to Use Table.ReorderColumns

Using Table.ReorderColumns is quite simple. Here's how to use it:

1. Open Microsoft Excel and go to the Power Query Editor.
2. Load the data source you want to manipulate.
3. Select the table you want to reorder columns in.
4. Click on the "Transform Data" button to open the Power Query Editor.

Table.Repeat

What is the Power Query M Function Table?

The Power Query M function Table is a function that allows users to create tables of data in Excel. These tables can be used to hold and manipulate data, which can then be used in further calculations and analyses. The Table function can be used to create tables from a variety of sources, including text files, Excel spreadsheets, and databases.

The M Code Behind the Power Query M Function Table

The M code behind the Power Query M function Table is what makes it such a powerful tool. This code is used to create, manipulate, and transform tables of data in Excel. The following is a brief overview of the M code that is used to create tables in Power Query.

Creating Tables

To create a table in Power Query, users must first define the columns of the table. This is done using the Table.AddColumn function, which allows users to add columns to the table. Columns can be added using a variety of methods, including hard-coding values, referencing other columns, and using custom functions.

Transforming Tables

Once a table has been created, users can begin to transform the data within the table. This is done using a variety of functions, including Table.TransformColumnTypes, which allows users to change the data type of a column, and Table.SelectColumns, which allows users to select specific columns to include in the table.

Filtering Tables

Users can also filter tables in Power Query using the Table.SelectRows function. This function allows users to filter tables based on specific criteria, such as column values or text matches. Users can also use the Table.Combine function to combine tables together into a single table.

Sorting Tables

Tables can be sorted in Power Query using the Table.Sort function. This function allows users to sort tables based on one or more columns, and can be used to sort tables in ascending or descending order.

Grouping Tables

The Table.Group function can be used to group data within a table. This function allows users to group data based on one or more columns, and can be used to calculate aggregate values, such as sums or averages, for each group.

The Power Query M function Table is a powerful tool that allows users to create, manipulate, and transform tables of data in Excel. The M code behind this function is what makes it such a useful tool, allowing users to create new tables, transform existing data, filter and sort

Table.ReplaceErrorValues

What are Error Values?

In Power Query, error values are values that indicate that a formula or transformation has failed. Common error values include #N/A, #VALUE!, #REF!, and #DIV/0!. These error values can occur for a variety of reasons, such as when a formula references a non-existent cell or when a division by zero occurs. Error values can be problematic when working with large datasets, as they can cause downstream calculations and transformations to fail.

What is Table.ReplaceErrorValues?

Table.ReplaceErrorValues is a function provided by the M language that allows users to replace error values in a table with a specified default value. The function takes two arguments: the first argument is the input table, and the second argument is the default value to replace error values with. The function returns a new table with the same schema as the input table, but with error values replaced with the specified default value.

Here's an example of how Table.ReplaceErrorValues works:

```
let
  Source = Table.FromRows({{1, 2}, {3, null}, {5, 0}}, {"Column1", "Column2"}),
  ReplacedErrors = Table.ReplaceErrorValues(Source, 999)
in
  ReplacedErrors
```

In this example, we have a table with two columns: Column1 and Column2. The second row of Column2 contains a null value, which is an error value. We want to replace this error value with the default value of 999. We use the Table.ReplaceErrorValues function to do this, passing in the input table (Source) and the default value (999). The function returns a new table (ReplacedErrors) with the same schema as the input table, but with the error value replaced with the default value.

How to Use Table.ReplaceErrorValues

To use Table.ReplaceErrorValues, you first need to have a table with error values that you want to replace. You then need to decide on a default value to replace the error values with. Once you have these two pieces of information, you can use the Table.ReplaceErrorValues

Table.ReplaceKeys

Understanding the Table.ReplaceKeys Function

The Table.ReplaceKeys function is used to replace the column names in a table with new column names. This function takes in two arguments, the first being the table to be modified, and the second being a list of key-value pairs that provide the old column names and the new column names.

For example, consider the following table:

Name	Age	Gender
John	32	Male
Jane	28	Female
Peter	42	Male

If we want to rename the “Name” column to “Full Name” and the “Gender” column to “Sex,” we can use the Table.ReplaceKeys function as follows:

```
let
  Source = Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WcvTcrVXSUTIwNlSK1YlIzUwFg9ACbQd", BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type text) meta [Serialized.Text = true]) in type table [#"Name" = _t, #"Age" = Int64.Type, Gender = _t]),
  ReplacedKeys = Table.ReplaceKeys(Source, {{"Name", "Full Name"}, {"Gender", "Sex"}})
in
  ReplacedKeys
```

The resulting table will be:

Full Name	Age	Sex
John	32	Male

Table.ReplaceMatchingRows

One such function is the Table.ReplaceMatchingRows function, which allows you to replace rows in a table that match a specified condition. In this article, we will explore the M code behind this function and how to use it effectively.

Understanding the Table.ReplaceMatchingRows Function

The Table.ReplaceMatchingRows function is used to replace rows in a table that match a specified condition. This function takes three arguments:

1. table: This is the table that you want to modify.
2. condition: This is the condition that you want to use to match the rows that you want to replace.
3. replacement: This is the replacement row that you want to use to replace the matching rows.

The syntax of the function is as follows:

```
Table.ReplaceMatchingRows(table as table, condition as function, replacement as function) as table
```

Using the Table.ReplaceMatchingRows Function

To use the Table.ReplaceMatchingRows function, you need to follow these steps:

1. Open the Power Query Editor.
2. Select the table that you want to modify.
3. Click on the “Add Column” tab and select “Custom Column”.
4. Enter a name for the new column.
5. In the “Custom Column Formula” dialog box, enter the Table.ReplaceMatchingRows function with the appropriate arguments.
6. Click “OK” to create the new column.

For example, suppose you have a table named “Sales” that contains the following columns: “Region”, “Product”, “Salesperson”, and “Sales”. You want to replace all rows where the “Region” column is equal to “West” with a new row that has the “Region” column set to “Midwest”. To do this, you can use the following formula in the “Custom Column Formula” dialog box:

Table.ReplaceRelationshipIdentity

Understanding Relationships in Power Query

Before we dive into the M code behind the Table.ReplaceRelationshipIdentity function, it is important to understand relationships in Power Query. In Power Query, relationships are used to connect tables based on a common column. For example, if you have a table of customers and a table of orders, you can create a relationship between the two tables based on the customer ID column. This allows you to create queries that combine data from both tables.

What is the Table.ReplaceRelationshipIdentity Function?

The Table.ReplaceRelationshipIdentity function is used to replace the identity of a relationship in a table. In Power Query, every relationship has a unique identity. This identity is used to identify the relationship when it is used in a query. Sometimes, it is necessary to replace the identity of a relationship. This can happen when you are creating a new table that has the same structure as an existing table, but with different data. In this case, you need to replace the identity of the relationship in order to connect the new table to the existing table.

The M Code Behind the Table.ReplaceRelationshipIdentity Function

The M code behind the Table.ReplaceRelationshipIdentity function is relatively simple. The function takes two arguments: the table that contains the relationship that you want to replace and the new relationship that you want to use. The M code for the function looks like this:

```
Table.ReplaceRelationshipIdentity(table as table, oldRelation as any, newRelation as any) as table
```

Let’s break down each part of the code:

– `Table.ReplaceRelationshipIdentity` is the name of the function.
– `table as table` is the first argument of the function. This is the table that contains the relationship that you want to replace.
– `oldRelation as any` is the second argument of the function. This is the old relationship that you want to replace.
– `newRelation as any` is the third argument of the function. This is the new relationship that you want to use.
– `as table` at the end of the code is the return type of the function. This tells Power Query that the function will return a table.

Examples of the Table.ReplaceRelationshipIdentity Function

Table.ReplaceRows

Understanding the Table.ReplaceRows Function

The Table.ReplaceRows function in Power Query allows you to replace a specific row or set of rows in a table with new ones. This function takes three arguments:

To use the Table.ReplaceRows function, you need to pass in the table you want to replace rows in, the row indices you want to replace, and a function that will create the new rows.

The M Code Behind the Table.ReplaceRows Function

When you use the Table.ReplaceRows function in Power Query, it generates M code to perform the replacement. The M code generated by the Table.ReplaceRows function is as follows:

Let's break down this code to understand what it does:

Creating the Replacerows Function

The replacerows function is responsible for creating the new rows that will replace the old ones. This function takes the old row as input and returns the new row.

To create the replacerows function, you can use the Record.AddField function to add or modify the fields in the old row. Here’s an example of the replacerows function that adds a new column to the table and modifies an existing column:

Table.ReplaceValue

What is the Table.ReplaceValue Function?

The Table.ReplaceValue function is used to replace specific values in a table or column with new values. It has the following syntax:

Table.ReplaceValue(table as table, old_value as any, new_value as any, optional replacer as nullable function) as table

The function takes four parameters:

– `table` – The table or column where the replacement should occur.
– `old_value` – The value that should be replaced.
– `new_value` – The new value that should replace the old value.
– `replacer` – (optional) A function that should be used to generate the new value. This function takes two parameters: the old value and the new value.

The function returns a new table or column with the specified values replaced.

How to Use the Table.ReplaceValue Function?

To use the Table.ReplaceValue function, you need to follow these steps:

1. Load the data into Power Query.
2. Select the table or column where the replacement should occur.
3. Go to the Transform tab and click on the Replace Values button.
4. In the Replace Values dialog box, enter the old value and the new value.
5. Click on OK to replace the specified values.

Alternatively, you can use the M language to write the code for the replacement. Here's an example:

```
let
  Source = Excel.CurrentWorkbook(){[Name="Table1"]}[Content],
  ReplaceValue = Table.ReplaceValue(Source,"old_value","new_value",Replacer.ReplaceText,{"Column1"})
```

Table.ReverseRows

What is Table.ReverseRows?

Table.ReverseRows is a function in Power Query that is used to reverse the order of rows in a table. This function takes a table as its input and returns a table with the same columns but with the rows in reverse order. The function can be useful when you want to view the data in a table in reverse order, or when you want to perform further transformations on the reversed data.

The M Code Behind Table.ReverseRows

The M code behind Table.ReverseRows is relatively simple. Here is what the code looks like:

```
(table as table) =>
  let
    Source = Table.FromRecords({}),
    ColumnCount = Table.ColumnCount(table),
    RowCount = Table.RowCount(table),
    Rows = List.Reverse(table[Rows]),
    NewTable = Table.FromColumns(Rows, Table.ColumnNames(table))
  in
    NewTable
```

Let’s break down this code line by line to understand what it does.

```
(table as table) =>
```

This line defines the input parameter for the function. In this case, the input parameter is a table and it is named “table”.

Table.RowCount

What is Power Query M Language?

Power Query is a data transformation and analysis tool that is a part of Microsoft Excel and Power BI. The M language is the programming language used by Power Query to perform data transformation and analysis. It is a functional programming language that is designed to work with data.

The M language is used to perform various tasks in Power Query, such as data cleaning, data transformation, and data aggregation. It is a powerful language that can handle large amounts of data and complex data transformations.

What is Table.RowCount Function?

The Table.RowCount function is used to count the number of rows in a table in Power Query. It returns the number of rows in the table as an integer value.

The syntax of the Table.RowCount function is as follows:

```
Table.RowCount(table as table, optional countNulls as nullable logical) as number
```

The table argument is the table for which you want to count the number of rows. The optional countNulls argument specifies whether to count null values as part of the row count. If countNulls is set to true, then null values are counted as part of the row count. If countNulls is set to false or not specified, then null values are not counted as part of the row count.

How Table.RowCount Function Works?

The Table.RowCount function works by counting the number of rows in the table. It first checks whether the table argument is a table type. If the table argument is not a table type, then an error is returned.

If the table argument is a table type, then the function checks whether the optional countNulls argument is specified. If the countNulls argument is specified and set to true, then the function counts the null values as part of the row count. If countNulls is not specified or set to false, then null values are not counted as part of the row count.

The function then counts the number of rows in the table and returns the row count as an integer value.

Examples of Table.RowCount Function

Let’s take a look at some examples of the Table.RowCount function in Power Query.

Table.Schema

In this article, we will take a deep dive into the M code behind the Power Query M function Table.Schema. We will explore how this function works, how to use it effectively, and some common use cases.

What is Table.Schema?

The Table.Schema function in Power Query is used to inspect the structure of a table. It returns a record that describes the schema of the table, including information about the columns, data types, and other metadata. The schema record is returned as a single row table, which can then be used for further analysis and transformation.

The Table.Schema function takes a single argument, which is the table that you want to inspect. The function returns a record with the following fields:

– Name: The name of the table.
– Columns: A list of records describing each column in the table.
– NavigationProperties: A list of records describing any navigation properties in the table.
– Meta: A record containing any additional metadata associated with the table.

How to Use Table.Schema

Using Table.Schema is straightforward. Simply pass the table that you want to inspect as an argument to the function. The resulting schema record can then be used for further analysis and transformation.

Here's an example of how to use Table.Schema:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMlTSUTJUMlSK1SgmmJqfmJiRmlaRUZiSk5xfl5gCI6JUWJg
YUmBWZ2eV5QWAJ0YloigFAA==", BinaryEncoding.Base64)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table
[Column1 = _t, Column2 = _t])),
  schema = Table.Schema(Source)
in
  schema
```

Table.SelectColumns

Understanding the Table.SelectColumns Function

The Table.SelectColumns function is used to select specific columns from a table of data. It takes two arguments: the first argument is the table to select columns from, and the second argument is a list of column names to select. The function returns a new table that contains only the selected columns.

Here is an example of how the Table.SelectColumns function is used:

```
let
  Source = Excel.CurrentWorkbook(){[Name="Table1"]}[Content],
  SelectColumns = Table.SelectColumns(Source, {"Column1", "Column2"})
in
  SelectColumns
```

In this example, we are selecting only the “Column1” and “Column2” columns from the “Table1” table.

Understanding the M Code Behind Table.SelectColumns

The M code behind the Table.SelectColumns function is relatively simple. It uses the Table.SelectColumns function to select the desired columns and returns the resulting table. Here is the M code for the Table.SelectColumns function:

```
(table as table, columnsToSelect as list) =>
  let
    columns = List.Intersect(Table.ColumnNames(table), columnsToSelect),
    selectColumns = Table.SelectColumns(table, columns)
  in
    selectColumns
```

Table.SelectRows

What is Table.SelectRows?

The Table.SelectRows function in Power Query is used to filter rows in a table based on a condition. The function takes two arguments: the table to filter and the condition to apply. The condition is expressed as an expression that evaluates to true or false for each row in the table.

Here is an example of the Table.SelectRows function in action:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WMjQ3VNJRMgoy1DFRMjJQ1VZJrYzwvyc9JrYxCwA=",
BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [#"Column 1" =
_t, #"Column 2" = _t]),
  FilteredRows = Table.SelectRows(Source, each [#"Column 1"] > 2)
in
  FilteredRows
```

In this example, we have a table with two columns, “Column 1” and “Column 2”. We use the Table.FromRows function to create a table from some JSON data. We then use the Table.SelectRows function to filter the table so that only rows where the value in “Column 1” is greater than 2 are returned.

Understanding the M Code

The M code behind the Table.SelectRows function is relatively simple. Here is the M code that is generated when we use the function:

```
Table.SelectRows = (table as table, condition as function) as table =>
  let
    filteredTable = Table.Select(table, condition)
Table.SelectRowsWithErrors
```

What is Table.SelectRowsWithErrors?

The Table.SelectRowsWithErrors function is used to filter a table and return only the rows that contain errors. It takes a table as an input and returns a table with only the rows that have errors. This function is useful when you have a large dataset and want to quickly identify and fix errors.

Syntax of Table.SelectRowsWithErrors

The syntax of Table.SelectRowsWithErrors is as follows:

```
Table.SelectRowsWithErrors(table as table, optional errorHandling as nullable number) as table
```

The `table` parameter is the input table that you want to filter. The `errorHandling` parameter is optional and specifies how errors should be handled. If you don't specify anything, the default value is 1, which means that errors will be replaced with null values.

How Table.SelectRowsWithErrors Works

Table.SelectRowsWithErrors works by using the Table.TransformRows function to iterate through each row of the input table and check for errors. If a row contains an error, it is added to a new table that is returned as the output.

The function checks for errors in each column of the table. If a column contains an error, the entire row is considered to contain an error. Errors can occur in various forms, such as #VALUE!, #REF!, #DIV/0!, #NAME?, and #NUM!.

How to Use Table.SelectRowsWithErrors

To use Table.SelectRowsWithErrors, follow these steps:

1. Open Power Query and create a new blank query.
2. Go to the Home tab and select "Advanced Editor".
3. In the editor, enter the following code:

```
let
  Source = [Enter your data source here],
Table.SingleRow
```

In this article, we will explore the M code behind the Table.SingleRow function and how it can be used to enhance your data transformation and cleansing efforts.

What is the Table.SingleRow Function?

The Table.SingleRow function is a Power Query M function that returns a table containing a single row from the input table. It is often used in conjunction with other functions in Power Query to extract specific data from a table.

The syntax for the Table.SingleRow function is as follows:

```
Table.SingleRow(table as table)
```

Where `table` is the input table from which you want to extract a single row.

Understanding the M Code Behind Table.SingleRow

The M code behind the Table.SingleRow function is relatively simple. When you apply the function to a table, Power Query creates a new table that contains only the first row of the original table.

Here is the M code behind the Table.SingleRow function:

```
let
  Source = table,
  FirstRow = Table.FirstN(Source, 1),
  Output = FirstRow{0}
in
  Output
```

The code starts by setting the input table as the `Source` variable. Next, the `FirstRow` variable is created by using the Table.FirstN

Table.Skip

What is Power Query M?

Power Query M is a functional, case-sensitive language used to query and transform data in Power Query. It is based on Microsoft's Functional Language (MFL) and is used to write the M code that is executed in Power Query. The M code is used to build queries that extract, transform, and load data from various sources. Power Query M includes a wide range of functions, including Table.Skip, which we'll be discussing in this article.

What is the Table.Skip function?

The Table.Skip function is a Power Query M function that allows users to skip a specified number of rows in a table. This function is useful when working with large datasets and you only want to focus on a portion of the data. By using the Table.Skip function, you can easily skip a certain number of rows in a table and focus on the data that is relevant to your analysis.

The M Code Behind the Table.Skip Function

The M code behind the Table.Skip function is relatively simple. To use the Table.Skip function, you need to specify the number of rows that you want to skip. The syntax for the Table.Skip function is as follows:

```
Table.Skip(table as table, count as number) as table
```

In this syntax, the "table" parameter is the input table that you want to skip rows from, and the "count" parameter is the number of rows that you want to skip. The output of the Table.Skip function is a new table that contains the remaining rows after skipping the specified number of rows.

Here's an example of the M code behind the Table.Skip function:

```
let
  Source = Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45W8ixLLdJRMgZwNlKSIrJBYEgWUAA==", BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text) meta [Serialized.Text = true]) in type table [#"Column 1" = _t]),
Table.Sort
```

Table.Sort is a function that sorts a table based on one or more columns. In this article, we will explore the M code behind Table.Sort and how you can use it to sort your data.

Understanding Table.Sort

Table.Sort takes two arguments: the table you want to sort and a list of columns and sort direction. You can use the optional third argument to specify a custom sorting function.

Here is the basic syntax of Table.Sort:

```
Table.Sort(table as table, sortColumns as list, optional comparer as nullable function) as table
```

Let's explore each argument in more detail:

table

The table argument is the table you want to sort. It can be any table in your data model. You can reference a table by its name, like this:

```
Table.Sort(MyTable, {{"Column1", Order.Ascending}})
```

sortColumns

The sortColumns argument is a list of columns and sort directions. You can specify one or more columns to sort by, in the order you want. Each column is represented by a list of two elements: the column name and the sort direction (either Order.Ascending or Order.Descending).

```
Table.Sort(MyTable, {{"Column1", Order.Ascending}, {"Column2", Order.Descending}})
```

Table.Split

What is the Table.Split function?

The Table.Split function is used to split a table column into multiple columns based on a delimiter. The syntax for the Table.Split function is as follows:

- The 'table' argument is the table that contains the column you want to split.
- The 'column' argument is the name of the column you want to split.
- The 'separator' argument is the delimiter that separates the values in the column.
- The 'newColumnNames' argument is a list of new column names you want to give to the split columns. If this argument is not specified, Power Query will automatically name the columns as Column1, Column2, and so on.

How does the Table.Split function work?

The Table.Split function works by splitting a column into multiple columns based on a delimiter. Let's take an example to understand how it works. Consider the following table:

Name	Age	Profession
John, Doe	35	Engineer
Jane, Smith	28	Accountant
Michael, Wong	43	Data Analyst

Suppose we want to split the 'Name' column into two separate columns, 'First Name' and 'Last Name', based on the comma delimiter. We can use the Table.Split function to achieve this. The M code for this operation would be:

– The first argument '#"Previous Step"' represents the previous step in the Power Query Editor.
– The second argument "Name" is the name of the column we want to split.
– The third argument "," is the delimiter used to split the values in the column.
– The fourth argument {"First Name", "Last Name"} is the list of new column names we want to assign to the split columns.

When we execute this M code, we get the following table:

Table.SplitAt

What is the Table.SplitAt Function?

The Table.SplitAt function in Power Query is used to split a table into two parts based on a given delimiter. The delimiter can be a column index or a column name. The function returns two tables – the first table contains the columns before the delimiter, and the second table contains the columns after the delimiter.

Understanding the M Code Behind the Table.SplitAt Function

The M code behind the Table.SplitAt function is quite simple. Let's take a look at the code:

```
let
  SplitTable = Table.SplitAt(Source, Index)
in
  SplitTable
```

In the code above, 'Source' refers to the table that needs to be split, and 'Index' refers to the delimiter column index or name. The Table.SplitAt function takes in these two parameters and returns two tables – the first table contains the columns before the delimiter, and the second table contains the columns after the delimiter.

Using the Table.SplitAt Function

Using the Table.SplitAt function is easy. Let's take an example to understand how it works.

Suppose we have a table 'SalesData' that contains the following columns – 'Product', 'Category', 'Sales', 'Profit', and 'Date'. We want to split this table into two parts based on the 'Profit' column.

To do this, we can use the following M code:

```
let
  Source = SalesData,
  Index = Table.ColumnIndex(Source, "Profit"),
Table.SplitColumn
```

Understanding the Table.SplitColumn Function

The Table.SplitColumn function in Power Query is used to split a column in a table into multiple columns. This function takes in three arguments: the table to be split, the name of the column to be split, and the split options.

When applying the Table.SplitColumn function, Power Query creates a new table with additional columns based on the split options. The original column is removed from the table, and the new columns are added to the right of the original table.

The M Code Behind Table.SplitColumn

Behind the scenes, the M code that powers the Table.SplitColumn function is relatively straightforward. When calling the function, Power Query generates the following M code:

```
= Table.SplitColumn(Source, ColumnName, Splitter, SplitterPattern)
```

Let’s go through each of these arguments in more detail.

Source

The first argument, Source, refers to the table to be split. This argument should be a reference to the table or a step that generates the table.

ColumnName

The second argument, ColumnName, refers to the name of the column to be split. This argument should be a string that matches the exact name of the column.

Splitter

The third argument, Splitter, refers to the delimiter used to split the column. This argument should be a text value that matches the exact delimiter used in the column. If there is no delimiter, you can use null.

SplitterPattern

The last argument, SplitterPattern, refers to the pattern to be used when splitting the column. This argument should be a string that matches the pattern in the column. If there is no pattern, you can use null.

Examples of Using Table.SplitColumn

Table.ToColumns

Understanding the M Code

Before we dive into examples of Table.ToColumns, let's take a moment to understand the M code behind this function. When you use Table.ToColumns, Power Query generates M code that takes the table you want to convert and transforms it into a list of columns. The M code generated looks something like this:

let
  Source =
, Columns = Table.ColumnNames(Source), Output = List.Transform(Columns, each Table.Column(Source, _))in OutputHere's a brief overview of what each part of this code does:– The `Source` variable contains the table you want to convert.– The `Columns` variable contains a list of column names from the table.– The `Output` variable uses the `List.Transform` function to loop through each column in the `Columns` list and extract the values from that column in the `Source` table.The end result is a list of columns that can be used for further data manipulation.Example UsageNow that we have a basic understanding of the M code behind Table.ToColumns, let's explore some practical examples of how it can be used.Example 1: Splitting a Column into Multiple ColumnsOne common use case for Table.ToColumns is to split a column in a table into multiple columns. For example, let's say you have a table with a column called "Full Name" that contains both the first and last name of each person. You can use Table.ToColumns to split this column into two separate columns containing just the first and last names.Here's the M code to do this:let Source =

Table.ToList

What is the Table.ToList function?

The Table.ToList function is a Power Query M function that is used to convert a table into a list. In other words, it takes the contents of a table and returns a list of records, where each record represents a row in the original table. The syntax for the Table.ToList function is as follows:

```
Table.ToList(table as table) as list
```

The function takes a single parameter, which is the table to be converted. The result is a list of records, where each record contains a set of name/value pairs that correspond to the columns in the original table.

How does the Table.ToList function work?

The Table.ToList function works by iterating over the rows in the input table and appending each row to a list. The resulting list is a collection of records, where each record represents a row in the original table. The name/value pairs in each record correspond to the column names and cell values in the original table.

Here's an example of how the Table.ToList function can be used:

```
let
  Source = Excel.CurrentWorkbook(){[Name="Table1"]}[Content],
  Result = Table.ToList(Source)
in
  Result
```

In this example, the Table.ToList function is applied to a table named "Table1" in the current workbook. The result is a list of records, where each record represents a row in the original table.

Table.ToRecords

What is Power Query M?

Power Query M is a language used to transform and manipulate data in Microsoft Excel and Power BI. It's a functional language that allows you to create custom functions, transform data, and perform complex calculations. Power Query M is used in the Power Query Editor, which is a tool used to extract, transform, and load data from various sources.

Understanding the Table.ToRecords Function

The Table.ToRecords function is used to convert a table into a list of records. This function takes a single parameter, which is the table that you want to convert. The resulting list contains one record for each row in the table, where each record contains the values of all the columns in that row.

Here's an example of how the Table.ToRecords function works:

```
let
  Source = Table.FromRows({{"John", 25},{"Sarah", 30},{"David", 40}}, {"Name", "Age"}),
  ToRecords = Table.ToRecords(Source)
in
  ToRecords
```

In this example, we have a table with two columns (Name and Age) and three rows. We use the Table.ToRecords function to convert this table into a list of records. The resulting list contains three records, where each record represents a row in the original table.

The M Code Behind the Table.ToRecords Function

The M code behind the Table.ToRecords function is relatively simple. Here's what it looks like:

```
(Table as table) as list =>
  List.Transform(Table.ToRows(Table), each Record.FromList(_, Table.ColumnNames(Table)))
```

Table.ToRows

Understanding Tables in Power Query

Before diving into the M code behind `Table.ToRows`, it is important to understand what a table is in Power Query. In Power Query, a table is a collection of rows and columns that represents data. Each row in a table represents a record, while each column represents a field or attribute of the record.

Tables in Power Query are created by importing data from various sources, such as Excel files, CSV files, databases, and web pages. Once a table is created, you can perform various operations on it, such as filtering, sorting, transforming, and aggregating data.

The M Code Behind Table.ToRows

The `Table.ToRows` function is a built-in M function in Power Query that takes a table as its argument and returns a list of rows. Each row in the list is itself a list of values from the original table.

Here is the M code behind the `Table.ToRows` function:

```
(Table as table) as list =>
  List.Transform(
    Table.ToList(Table.SelectColumns(Table, Table.ColumnNames(Table))),
    each Record.FieldValues(_)
  )
```

Let's break down this code and understand what each line does.

Line 1: The Function Signature

The first line of the code defines the function signature, which specifies the name of the function (`Table.ToRows`) and the type of its argument (`table`). In Power Query, function signatures are written in the format `FunctionName(ArgumentName as ArgumentType) as ReturnType`.

Line 2: Converting the Table to a List

The second line of the code uses the `Table.ToList` function to convert the input table into a list. The `Table.ToList` function takes a table as its argument and returns a list of records, where each record is a dictionary with the column names as keys and the row values

Table.TransformColumnNames

What is the Table.TransformColumnNames Function?

The Table.TransformColumnNames function in Power Query is used to rename the columns of a table. It takes a table as input and returns a new table with the modified column names. The function is part of the M language, which is the underlying language used by Power Query. M is a functional programming language that is used to describe data transformations. The Table.TransformColumnNames function is just one of many functions available in M that can be used to manipulate data.

How to use the Table.TransformColumnNames Function

The Table.TransformColumnNames function takes two arguments: the table to be modified and a function that specifies how to modify the column names. The function that modifies the column names takes a text value as input and returns a text value. The function can be defined inline, or it can be defined as a separate function and passed as an argument to the Table.TransformColumnNames function. Here is an example of how to use the Table.TransformColumnNames function to modify the column names of a table:

```
let
  Source = Table.FromRecords({
    [Column1 = 1, Column2 = "A"],
    [Column1 = 2, Column2 = "B"],
    [Column1 = 3, Column2 = "C"]
  }),
  NewColumnNames = Table.TransformColumnNames(
    Source,
    each "NewName_" & _
  )
in
  NewColumnNames
```

This code creates a table with two columns (Column1 and Column2) and three rows. It then uses the Table.TransformColumnNames

Table.TransformColumns

Understanding the Table.TransformColumns M function

Before we dive into the M code behind Table.TransformColumns, let's first understand what this function does. As mentioned earlier, this function allows you to apply transformations to specific columns within your table.

The syntax of the function is as follows:

```
Table.TransformColumns(table as table, transformations as list)
```

Here, the `table` parameter refers to the input table, and the `transformations` parameter refers to the list of transformations that you want to apply.

The `transformations` list is made up of one or more records. Each record contains two fields: `ColumnName` and `Transformation`. The `ColumnName` field specifies the name of the column that you want to transform, while the `Transformation` field specifies the transformation that you want to apply.

The M code behind Table.TransformColumns

Now that we have a basic understanding of Table.TransformColumns, let's take a look at the M code behind this function. The M code for the Table.TransformColumns function is as follows:

```
(Table as table, Transformations as list) =>
let
  ColumnNames = Table.ColumnNames(Table),
  TransformedColumns = List.Transform(Transformations, each TransformColumn(Table, _, _)),
  RemainingColumns = List.RemoveItems(ColumnNames, Transformations[ColumnName]),
  Output = Table.SelectColumns(Table, RemainingColumns) & TransformedColumns
in
  Output
```

Table.TransformColumnTypes

In this article, we will delve into the M code behind the Table.TransformColumnTypes function and explore some of its features.

Understanding the basics of the Table.TransformColumnTypes function

The Table.TransformColumnTypes function is used to modify the data type of one or more columns in a table. It takes two arguments: the first argument is the table to be modified, and the second argument is a list of column specifications.

Each column specification is a record that contains the name of the column and its new data type. For example, if we want to change the data type of the “Price” column in a table to Decimal, we would use the following code:

```
Table.TransformColumnTypes(
  SourceTable,
  {
    {"Price", type number}
  }
)
```

This code changes the data type of the “Price” column in the “SourceTable” to Decimal.

Changing multiple column data types

The Table.TransformColumnTypes function can be used to change the data type of multiple columns at once. To do this, we simply add more column specifications to the list.

For example, if we want to change the data type of both the “Price” and “Quantity” columns in a table to Decimal, we would use the following code:

```
Table.TransformColumnTypes(
  SourceTable,
  {
Table.TransformRows
```

Understanding Table.TransformRows

Before we dive into the M code behind Table.TransformRows, let's first take a moment to understand what this function does. Essentially, Table.TransformRows allows you to apply a function to each row of a table. The function you apply can modify the data in each row, or perform some other action based on the data in that row.

For example, let's say you have a table of customer orders, and you want to calculate the total cost of each order. With Table.TransformRows, you could apply a function to each row of the table that calculates the total cost based on the quantity and price of each item in the order.

The M Code Behind Table.TransformRows

So, what does the M code behind Table.TransformRows look like? Here's an example:

```
Table.TransformRows = (table as table, transform as function) as table =>
  let
    newColumns = List.Transform(Table.ColumnNames(table), each {_, type any}),
    transformedRows = Table.ToRows(Table.AddColumn(table, "Transformed", transform)),
    expandedRows = List.Combine(List.Transform(transformedRows, each Record.FieldValues(_[Transformed]))),
    expandedTable = Table.FromColumns(List.Transform(newColumns, each {_[0], List.Last(_[1])}), List.Transform(newColumns, each _[0])),
    removedColumns = Table.RemoveColumns(expandedTable, List.PositionOf(Table.ColumnNames(expandedTable), "Transformed"))
  in
    removedColumns
```

At first glance, this code may look a bit intimidating. But let's break it down step by step.

The first line defines the function signature. It takes two arguments – a table and a transform function – and returns a table.

Next, we use the List.Transform function to create a new list of columns for our transformed table. This list includes all of the existing

Table.Transpose

In this article, we will explore the M code behind the Table.Transpose function and learn how it works.

What is the Table.Transpose function?

The Table.Transpose function is a built-in function in Power Query that allows you to transpose a table. Transposing a table means that you switch the rows and columns, so that the rows become columns and the columns become rows.

For example, if you have a table with three columns and four rows, transposing the table will create a new table with four columns and three rows.

How to use the Table.Transpose function

Using the Table.Transpose function is simple. You can either use the graphical user interface (GUI) or write the M code directly.

To use the GUI, follow these steps:

1. Select the table you want to transpose.
2. Go to the Transform tab and click on the Transpose button.
3. A new table will be created with the rows and columns transposed.

To write the M code directly, you can use the following code:

```
= Table.Transpose(#"Previous Step")
```

Replace “Previous Step” with the name of the previous step in your query. This code will create a new step in your query that transposes the table.

The M code behind the Table.Transpose function

The M code behind the Table.Transpose function is relatively simple. When you use the Table.Transpose function, Power Query generates the following M code:

```
let
  Source = ,
Table.Unpivot
```

What is the Table.Unpivot Function?

The Table.Unpivot function is a part of the Power Query M language. It is used to transform wide tables into long tables by unpivoting columns. When you unpivot a table, you essentially take the data from the column headings and turn them into rows. This makes it easier to analyze and visualize the data.

To use the Table.Unpivot function, you need to have a table with at least two columns. One column should be the identifier column, and the other columns should be the data columns. The data columns should all be of the same data type.

How to Use the Table.Unpivot Function

To use the Table.Unpivot function, you need to follow these steps:

1. Select the table you want to unpivot.
2. Go to the Transform tab in the Power Query Editor.
3. Click on the Unpivot Columns button.
4. Select the columns you want to unpivot.
5. Click on OK.

Once you have clicked on OK, Power Query will generate the M code for the Table.Unpivot function. You can then edit the M code to customize the transformation to your needs.

The M Code Behind the Table.Unpivot Function

The M code behind the Table.Unpivot function is relatively simple. The function takes two arguments: the table to be unpivoted, and the column names to be unpivoted. Here is the basic syntax for the Table.Unpivot function:

```
Table.Unpivot(Table as table, ColumnNames as list)
```

The `Table` argument is the table you want to unpivot, and the `ColumnNames` argument is a list of the column names you want to unpivot. Here is an example of how to use the Table.Unpivot function:

```
Table.UnpivotOtherColumns
```

What is Table.UnpivotOtherColumns?

Table.UnpivotOtherColumns is a Power Query M function that allows users to unpivot columns in a table while keeping other columns intact. This function is particularly useful when you have a table with multiple columns, and you want to unpivot some of them but leave others as they are.

For example, let's say you have a table with the following columns:

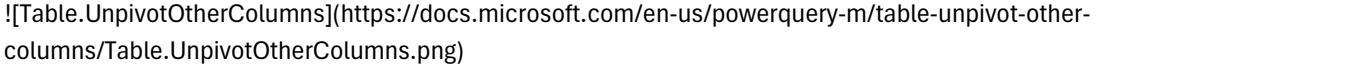
Name	Age	Address	City
——	——	———	——
John	25	123 Main St.	New York
Jane	30	456 Elm St.	Los Angeles
Bob	40	789 Oak St.	Chicago

If you wanted to unpivot the "Age", "Address", and "City" columns, but leave the "Name" column as it is, you could use the Table.UnpivotOtherColumns function.

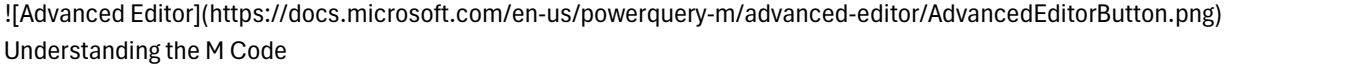
How to Use Table.UnpivotOtherColumns

To use the Table.UnpivotOtherColumns function, you need to have a table loaded into Power Query. Once you have your table loaded, follow these steps:

1. Select the columns you want to unpivot by clicking on their headers.
2. Right-click on one of the selected columns and choose "Unpivot Other Columns" from the context menu.



3. Power Query will generate the M code for you, which you can view by clicking on the "Advanced Editor" button on the "View" tab.



Understanding the M Code

Now that we know how to use Table.UnpivotOtherColumns, let's take a closer look at the M code it generates.

Here's an example of the M code generated by Table.UnpivotOtherColumns for our example table:

```
let
Table.View
```

One of the most powerful features of Power Query is the ability to write custom M code. M is the language used by Power Query to define transformations and data manipulation operations. By writing custom M code, you can extend the capabilities of Power Query and create complex data transformations that are not available out of the box.

In this article, we will explore the M code behind the Power Query M function Table.View. Table.View is a powerful function that lets you visualize the contents of a table in Power Query. We will examine the syntax and parameters of Table.View, and provide examples of how you can use it in your data transformation workflows.

Syntax of Table.View

The syntax of Table.View is straightforward:

```
Table.View(table as table, optional columns as list, optional sorting as list, optional ascending as list, optional maximumRowsToShow as nullable number) as table
```

The first parameter, `table`, is a required parameter that specifies the table you want to visualize. This can be a table loaded from a data source, or a table generated by a previous step in your transformation workflow.

The second parameter, `columns`, is an optional parameter that lets you specify which columns you want to include in the visualization. By default, all columns in the table are included.

The third parameter, `sorting`, is an optional parameter that lets you specify the sorting order of the table. This can be used to sort the table by one or more columns in ascending or descending order.

The fourth parameter, `ascending`, is an optional parameter that lets you specify the ascending order of the sorting. This should be a list with the same length as the `sorting` list, where each item in the list corresponds to the sorting order of the corresponding column.

The fifth parameter, `maximumRowsToShow`, is an optional parameter that lets you specify the maximum number of rows to include in the visualization. By default, all rows in the table are included.

Examples of Table.View

Let’s look at some examples of how you can use Table.View to visualize tables in Power Query.

Example 1: Visualizing a simple table

```
Table.ViewError
```

In this article, we will take a closer look at the M code behind the Table.ViewError function, and explore some of the ways in which it can be used to improve the accuracy and reliability of your data.

Understanding the Table.ViewError Function

The Table.ViewError function is a built-in function in Power Query that allows you to view and fix errors in your data. When you apply this function to a table, it will display a list of all the errors in that table, along with a description of the error type and the location of the error. For example, if you have a table that contains a column of dates, and some of those dates are in an incorrect format, you can use the Table.ViewError function to quickly identify and fix those errors.

The M Code Behind the Table.ViewError Function

The Table.ViewError function is actually a combination of several different M functions that work together to identify and display errors in your data. Here is a breakdown of the M code that makes up the Table.ViewError function:

Table.Buffer: This function is used to create a temporary copy of the table that is being analyzed. This copy is used to perform all of the error checks, so that the original table is not modified.

Table.SelectRows: This function is used to select only the rows in the table that contain errors. It does this by applying a custom error-checking function to each row in the table, and only selecting the rows that fail the check.

List.Transform: This function is used to transform the list of error rows into a more readable format. It does this by applying a custom formatting function to each row in the list, which converts the error data into a user-friendly format.

Table.FromRecords: This function is used to convert the formatted list of error rows into a new table, which is then displayed to the user.

Using the Table.ViewError Function

Now that we understand how the Table.ViewError function works, let's take a look at some of the ways in which it can be used to improve the accuracy and reliability of your data.

Identifying Data Entry Errors

One of the most common uses of the Table.ViewError function is to identify errors that were made during data entry. For example, if you have a table that contains customer names, and some of those names are misspelled or entered in the wrong format, the Table.ViewError function can help you identify and correct those errors.

Validating Data Formats

Another use of the Table.ViewError function is to validate data formats. For example, if you have a table that contains date values, you can use the Table.ViewError function to identify any values that are not in the correct date format. This can help you ensure that your

One of the functions that makes heavy use of M code is Table.ViewFunction. This function allows you to create a custom view of a table, which can be useful for tasks such as data cleaning, filtering, and formatting.

Understanding the Table.ViewFunction Function

The Table.ViewFunction function is used to create a custom view of a table. It takes two arguments: a table to create the view from, and a function that defines the view. The function takes a single argument, a row from the table, and returns a record that represents the transformed row.

Here's an example of a simple view function that takes a table of sales data and returns only the rows where the sales amount is greater than 1000:

```
let
  salesTable = #table(
    {"Region", "Product", "Sales"},
    {
      {"North", "Product A", 500},
      {"North", "Product B", 1500},
      {"South", "Product A", 750},
      {"South", "Product B", 2000}
    }
  ),
  filterSales = Table.ViewFunction(salesTable, each if [Sales] > 1000 then _ else null)
in
  filterSales
```

In this example, the Table.ViewFunction function is used to create a new table called `filterSales` that contains only the rows where the sales amount is greater than 1000. The view function used here is `each if [Sales] > 1000 then _ else null`, which returns the input row if

Tables.GetRelationships

How Tables.GetRelationships() Works

The Tables.GetRelationships() function is part of the M language syntax that is used in Power Query. This function is responsible for extracting all the relationships between tables in a workbook or a data model. The function takes a single argument, which is the name of the table or data model that you want to extract the relationships from. The function then returns a table that contains the following columns:

– Table: The name of the source table that contains the relationship
– Column: The name of the column in the source table that is part of the relationship
– Related Table: The name of the target table that is part of the relationship
– Related Column: The name of the column in the target table that is part of the relationship

Here is an example of how the Tables.GetRelationships() function works:

```
let
  Source = Excel.Workbook(File.Contents("C:MyWorkbook.xlsx"), null, true),
  Relationships = Source{[Name="Model"]}[Content]{0}[Tables]{0}[Tables.GetRelationships]()
in
  Relationships
```

In this example, the function extracts all the relationships from the table named “Model” in an Excel workbook located at “C:MyWorkbook.xlsx”. The result of the function is a table that contains the columns described above.

Usage of Tables.GetRelationships()

The Tables.GetRelationships() function is a very useful tool for managing and transforming data in Power Query. Here are some of the common use cases for this function:

1. Understanding the Data Model

When working with a complex data model in Power BI, it is often important to understand the relationships between tables. The Tables.GetRelationships() function can be used to extract all the relationships in a data model and present them in a clear and concise

Teradata.Database

Teradata.Database is a built-in function in Power Query that allows users to connect to Teradata databases and retrieve data. The syntax of the function is as follows:

Teradata.Database(server as text, database as text, options as record) as table

The function takes three parameters: the server name, the database name, and an optional record of options. The server and database parameters are required, while the options parameter is optional. Let's take a closer look at each of these parameters.

Server Parameter

The server parameter specifies the name of the Teradata database server that Power Query should connect to. This can be either the IP address or the fully qualified domain name of the server. If the server requires a specific port number, it can be specified by appending it to the server name using a comma, for example:

"myserver.example.com,1433"

Database Parameter

The database parameter specifies the name of the Teradata database that Power Query should retrieve data from. This is typically the name of the database that contains the data you want to analyze.

Options Parameter

The options parameter is an optional record that specifies additional options for the Teradata.Database function. The available options are:

– Query: A SQL query to execute against the Teradata database. This can be used to retrieve specific data or to perform data transformations within the query itself.

– CommandTimeout: The number of seconds to wait for a command to complete before timing out. This can be useful if you are working

Text.AfterDelimiter

What is Power Query M?

Power Query M is a functional language that is used to query and transform data in Power BI. It is a powerful tool that allows users to manipulate data in a variety of ways, including merging data, filtering data, and pivoting data. The language is designed to be easy to learn and can be used by users with little to no programming experience.

Understanding the Text.AfterDelimiter Function

The Text.AfterDelimiter function is designed to extract text that appears after a specific delimiter. The delimiter is defined as the character or sequence of characters that separates the text to be extracted from the rest of the text string.

The syntax for the Text.AfterDelimiter function is as follows:

```
Text.AfterDelimiter([Text], [Delimiter], [Occurrence])
```

Where:

– [Text]: The text string from which to extract the text.

– [Delimiter]: The character or sequence of characters that separates the text to be extracted from the rest of the text string.

– [Occurrence]: (Optional) The occurrence of the delimiter to use when extracting the text. The default value is 1.

The Text.AfterDelimiter function returns the text that appears after the specified delimiter. If the delimiter is not found, the function returns a null value. If the occurrence of the delimiter is not found, the function returns an error.

Examples of Using the Text.AfterDelimiter Function

Let's take a look at some examples of how the Text.AfterDelimiter function can be used to extract text from a variety of text strings.

Example 1: Extracting Text After a Single Character Delimiter

Suppose we have a text string that contains a person's full name, with the first and last names separated by a comma. We can use the Text.AfterDelimiter function to extract the last name from the text string.

```
let
Text.At
```

Understanding Text.At Function

The Text.At function is used to extract a character from a text string at a specified position. The function takes two arguments: the text string and the position of the character to be extracted. The position argument can be a number or a dynamic value.

The syntax of the Text.At function is as follows:

```
Text.At(text as text, position as number) as text
```

The function returns a single character as a string.

The M Code Behind the Text.At Function

The Text.At function is a simple function that can be easily implemented in M code. The following M code demonstrates the logic behind the Text.At function:

```
let
  TextAt = (text as text, position as number) =>
    let
      len = Text.Length(text),
      char = if position >= 0 and position <= len then Text.Range(text, position, 1) else ""
    in
      char
in
  TextAt
```

The M code defines a custom function called TextAt that takes two arguments: the text string and the position of the character to be

Text.BeforeDelimiter

The M code behind the Text.BeforeDelimiter function is what makes it possible to extract text from a string. The M language is a functional language used in Power Query to transform data. It is a powerful language that allows users to create complex transformations.

Understanding the Text.BeforeDelimiter Function

The Text.BeforeDelimiter function is used to extract text from a string before a delimiter character. For example, if we have a string “John Smith,CEO” and we want to extract the text before the comma, we can use the Text.BeforeDelimiter function. The function takes two arguments, the string and the delimiter character.

The syntax of the Text.BeforeDelimiter function is as follows:

Text.BeforeDelimiter(string as text, delimiter as text) as text

The function returns the text before the delimiter character. If the delimiter character is not found in the string, the function returns a blank value.

The M Code Behind Text.BeforeDelimiter

The M code behind the Text.BeforeDelimiter function is fairly simple. The function uses the Text.PositionOf function to find the position of the delimiter character in the string. It then uses the Text.Start function to extract the text before the delimiter character.

The M code for the Text.BeforeDelimiter function is as follows:

```
(Text as text, Delimiter as text) =>
let
  Pos = Text.PositionOf(Text, Delimiter),
  Result = if Pos = -1 then Text else Text.Start(Text, Pos)
in
  Result
```

In the code above, we can see that the function takes two parameters, Text and Delimiter. The Text.PositionOf function finds the position of the Delimiter character in the Text string. If the Delimiter character is not found, the Text.PositionOf function returns -1. The Text.BetweenDelimiters

Understanding the Text.BetweenDelimiters Function

Before we dive into the M code, it's important to understand how the Text.BetweenDelimiters function works. The Text.BetweenDelimiters function takes three arguments:

1. Text: The text you want to extract from.
2. Left Delimiter: The left delimiter that marks the beginning of the string you want to extract.
3. Right Delimiter: The right delimiter that marks the end of the string you want to extract.

The Text.BetweenDelimiters function then returns the text that's between the two specified delimiters. Let's take a look at an example to understand this better:

Suppose you have the following text: "Hello World! This is a sample text." You want to extract the text between the words "Hello" and "sample". You can use the Text.BetweenDelimiters function as follows:

```
= Text.BetweenDelimiters("Hello World! This is a sample text.", "Hello", "sample")
```

The above formula will return the following text: " World! This is a ".

Now that we understand how the function works, let's take a look at the M code that powers the Text.BetweenDelimiters function.

The M Code Behind the Text.BetweenDelimiters Function

The Text.BetweenDelimiters function is a built-in function in Power Query, which means its underlying M code is not visible to the user. However, we can use the "view native query" feature of Power Query to view the underlying M code.

To view the M code behind the Text.BetweenDelimiters function, we first need to create a sample query that uses the Text.BetweenDelimiters function. Let's create a simple query that uses the Text.BetweenDelimiters function:

1. Open Power Query and create a new query.
2. In the "Home" tab, click on "From Other Sources" and select "Blank Query".
3. In the "Query Editor" window, click on "View" and select "Advanced Editor".
4. In the "Advanced Editor" window, copy and paste the following M code:

```
Text.Combine
```

In this article, we will take a deep dive into the M code behind the Text.Combine function, exploring its syntax and capabilities, and providing examples of how it can be used to simplify data transformations in Power Query.

Syntax of the Text.Combine Function

The Text.Combine function is used to combine text values from multiple columns into a single column. It takes two arguments:

```
Text.Combine(list as list, optional separator as nullable text) as text
```

The first argument is a list of text values that need to be combined. The second argument is an optional separator that is used to separate the text values in the final output. If the separator argument is not specified, the default separator is a blank space.

Examples of Using the Text.Combine Function

Example 1 – Combining Two Columns

Suppose we have a table containing two columns – First Name and Last Name. We want to combine these two columns into a single column called Full Name. The M code to achieve this using the Text.Combine function is as follows:

```
= Table.AddColumn(Source, "Full Name", each Text.Combine({[First Name], [Last Name]}, " "))
```

In this example, the Text.Combine function is used to combine the values from the First Name and Last Name columns into a single column called Full Name. The curly braces { } are used to create a list of the text values that need to be combined. The separator argument (" ") is used to separate the First Name and Last Name values with a blank space.

Example 2 – Combining Multiple Columns

Suppose we have a table containing three columns – First Name, Middle Name, and Last Name. We want to combine these three columns into a single column called Full Name. The M code to achieve this using the Text.Combine function is as follows:

Text.Contains

In this article, we will explore the M code behind the Text.Contains function and how it can be used to perform complex text matching operations in Power Query.

Overview of Text.Contains Function

The Text.Contains function in Power Query takes two arguments: the text to search and the substring to search for. It returns a Boolean value of true if the substring is found in the text, and false otherwise.

The syntax for the Text.Contains function is as follows:

```
Text.Contains(text as nullable text, substring as text, optional comparer as nullable function) as logical
```

The first argument, `text`, is the text to search. It can be a column in a data table, a variable, or a literal string.

The second argument, `substring`, is the substring to search for. It can also be a column, a variable, or a literal string.

The optional third argument, `comparer`, is a function that specifies how the comparison should be made. It can be used to perform case-insensitive or culture-specific comparisons.

The M Code Behind Text.Contains

The M code behind the Text.Contains function is relatively simple, as it consists of just a few lines of code. Here's an example of how the function can be implemented in M code:

```
(text as nullable text, substring as text, optional comparer as nullable function) =>
  if text = null or substring = null then null
  else if comparer <> null then comparer(text, substring)
  else Text.Contains(text, substring)
```

The code defines a function that takes the same three arguments as the Text.Contains function. It then checks if either the `text` or

Text.End

Introduction to Text.End

The Text.End function is used to extract a specified number of characters from the end of a text string. The function takes two arguments: the text string to extract characters from, and the number of characters to extract. The syntax for the function is as follows:

```
Text.End(text as text, count as number) as text
```

For example, suppose we have a column of email addresses in a table, and we want to extract the domain name from each address. We could use the Text.End function to extract the last three characters of each address, which would give us the domain name:

```
= Table.AddColumn(PreviousStep, "Domain", each Text.End([Email], 3))
```

The M Code Behind Text.End

To understand the M code behind the Text.End function, let's take a look at a simple example. Suppose we want to extract the last three characters of the text string "hello". We can do this using the following M code:

```
let
  text = "hello",
  count = 3,
  result = Text.End(text, count)
in
  result
```

Text.EndsWith

Understanding Text.EndsWith

Text.EndsWith is a simple function that takes two parameters: a text value and a text string. It returns a Boolean value, which is true if the text value ends with the specified text string, and false otherwise. Here's an example of how to use the function:

```
= Text.EndsWith("Hello world", "world")
```

This will return true because the text value "Hello world" ends with the text string "world".

The M Code Behind Text.EndsWith

The M code behind the Text.EndsWith function is fairly simple. Here's the code:

```
(Text as nullable text, Ends as text) as logical =>
   if Text.End(Text, Text.Length(Ends)) = Ends then true else false
```

Let's break this down. The function takes two parameters: "Text", which is the text value we want to check, and "Ends", which is the text string we want to check for. The "as nullable text" part of the code means that the "Text" parameter can be null, which is useful in some cases.

The second part of the code, "if Text.End(Text, Text.Length(Ends)) = Ends then true else false", is where the magic happens. This code checks if the specified text value ends with the specified text string. Here's how it works:

1. The Text.End function takes two parameters: "Text" (the text value we want to check) and "Text.Length(Ends)" (the length of the text string we want to check for). This returns the last "n" characters of the text value, where "n" is the length of the specified text string.
2. The code then checks if the last "n" characters of the text value are equal to the specified text string. If they are, the function returns true. If they are not, the function returns false.

Using Text.EndsWith in Power Query

Text.From

In this article, we will explore the M code behind the Text.From function and learn how it works to convert different data types into text.

What is the Text.From Function?

The Text.From function is a Power Query M function that takes a value as input and returns its text representation. It is a versatile function that can handle values of various data types, such as numbers, dates, lists, tables, and more.

The Text.From function is typically used in data transformation scenarios where we need to convert non-text values into text for further processing or analysis. For instance, we may want to convert a numeric value into a text string to concatenate it with other text values or convert a date value into a specific text format.

Syntax of Text.From Function

The syntax of the Text.From function is straightforward. It takes a single argument, which is the value to be converted into text.

```
Text.From(value as any) as text
```

The argument “value” can be of any data type, and the function returns the text representation of the input value. The output is always a text string, even if the input is already a text string.

How Text.From Works

The Text.From function works by applying the appropriate conversion logic based on the input data type. It uses a set of built-in conversion rules to transform each data type into its corresponding text representation.

Let’s take a look at some examples of how Text.From works with different data types:

Converting Numbers into Text

When we pass a numeric value to the Text.From function, it converts it into a text string by default. For instance, the expression Text.From(123) returns “123”.

However, we can also specify a specific format for the output text string using the optional second argument of the Text.From function. For example, the expression Text.From(123, “0.00”) returns “123.00”.

Converting Dates into Text

When we pass a date value to the Text.From function, it converts it into a text string using the default date format (“yyyy-mm-dd”). For

Text.FromBinary

One function in M that comes in handy is the Text.FromBinary function. This function is used to convert binary data into text format. In this article, we will explore the M code behind the Text.FromBinary function and understand how it works.

Understanding Binary Data

Before we dive into the M code, let's first understand what binary data is. Binary data is a sequence of 0s and 1s that computers use to represent data. It is the most basic form of data storage used by computers. Binary data is used to store text, images, videos, and any other type of data that computers process.

The Text.FromBinary function

The Text.FromBinary function is used to convert binary data into text format. It takes in a binary value as an argument and returns a text value. The syntax for the Text.FromBinary function is as follows:

```
Text.FromBinary(binary as binary) as text
```

The binary argument is the binary data that needs to be converted, and the function returns the text representation of the binary data.

The M Code Behind the Text.FromBinary function

Now that we understand what the Text.FromBinary function does, let's take a look at the M code behind it. The M code for the Text.FromBinary function is as follows:

```
let
  Source = Binary.ToText(binary, BinaryEncoding.Base64)
in
  Source
```

The M code consists of a single expression that uses the Binary.ToText function to convert binary data into text format. The Binary.ToText

Text.InferNumberType

In this article, we will take a closer look at the M code behind the Text.InferNumberType function, and see how it can be used to improve data analysis.

What is the Text.InferNumberType function?

The Text.InferNumberType function is used to automatically detect the data type of a column in a table. This function is particularly useful when dealing with large datasets, as it can save a considerable amount of time that would otherwise be spent manually checking the data type of each column.

The Text.InferNumberType function works by analyzing the values in a column and determining their data type based on certain criteria. For example, if a column contains only integers, the Text.InferNumberType function will recognize it as an integer column.

How does the Text.InferNumberType function work?

The Text.InferNumberType function works by analyzing the values in a column and determining their data type based on certain criteria. The function considers the following criteria when determining the data type of a column:

– The presence of decimal points
– The presence of negative signs
– The presence of scientific notation
– The number of characters in a value

Based on these criteria, the Text.InferNumberType function can recognize the following data types:

– Integer
– Decimal
– Scientific
– Text

Let’s take a closer look at how the Text.InferNumberType function analyzes each of these data types.

Integer

An integer is a whole number without a decimal point. To be recognized as an integer column, the Text.InferNumberType function looks for a column where all of the values meet the following criteria:

– The value can be converted to a whole number without rounding
– The value does not contain any decimal points or scientific notation
– The value does not contain any negative signs

Text.Insert

Syntax of Text.Insert Function
The syntax of the Text.Insert function is as follows:

Text.Insert(text as nullable text, offset as number, insert as text) as nullable text

Here, `text` is the text in which you want to insert the substring, `offset` is the index at which you want to insert the substring, and `insert` is the substring that you want to insert.
Understanding the M Code Behind Text.Insert
To understand the M code behind the Text.Insert function, let's take a look at an example. Suppose we have the following table in Power Query:

Name	Age	Gender
John	30	Male
Emily	25	Female

Now, we want to insert the word "is" after the word "Name" in the column headers. To do this, we can use the following M code:

```
let
  Source =
Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText("i45WUtY0sgvK1UvOz0lV0lFyKzOzUwvOyC9JzE0uSS0q0gpKSix
JzU0tSrVTS1WwMgA", BinaryEncoding.Base64)), let _t = ((type text) meta [Serialized.Text = true]) in type table [#"Name" = _t, Age = _t,
Text.Length
```

However, while the Text.Length function is simple to use, its underlying M code is more complex. In this article, we will explore the M code behind Text.Length, and how it works to provide users with accurate character counts.

What is the M language?

Before we dive into the M code behind Text.Length, it's important to understand the M language itself. M is the language used by Power Query to manipulate and transform data. It is a functional language, which means that it relies on the input and output of functions to perform specific tasks.

While M may seem daunting at first, it is actually quite intuitive and easy to use once you become familiar with its syntax and structure. In fact, many of Power Query's most powerful functions, including Text.Length, are built using M code.

Understanding Text.Length

The Text.Length function is used to count the number of characters in a given text string. To use Text.Length, simply enter the text string you wish to count as the function's argument. For example:

```
=Text.Length("Hello, world!")
```

This will return the number of characters in the text string "Hello, world!" (which is 13, if you're counting).

But how does Text.Length actually work? Let's take a look at the M code behind the function to find out.

The M code behind Text.Length

The M code behind Text.Length is actually quite simple. Here is the code:

```
let
  Source = (#"Text" as text) =>
    Text.Length(#"Text")
in
  Source
```

Text.Lower

Text.Lower is a Power Query M function that takes a text input and returns the same text in lowercase format. In this article, we'll take a closer look at the M code behind the Text.Lower function and explore some examples of how it can be used.

Understanding the M Code Behind Text.Lower

The M code behind Text.Lower is relatively simple. Here's what the code looks like:

```
(Text as text) as text =>
   Text.Lower(Text)
```

Let's break this down a bit. The first line defines the function signature. It specifies that the function takes a single argument (Text) of type text and returns a value of type text.

The second line is the function body. It simply calls the built-in Text.Lower function and passes in the Text argument.

Examples of Using Text.Lower

Now that we understand how the Text.Lower function works, let's take a look at some examples of how it can be used.

Example 1: Converting Uppercase Text to Lowercase Text

Suppose we have a column in our data that contains names in all uppercase letters. We can use the Text.Lower function to convert these names to lowercase. Here's what the code would look like:

```
= Table.TransformColumns(
   Source,
   {{"Name", each Text.Lower(_), type text}}
)
```

This code uses the Table.TransformColumns function to apply the Text.Lower function to the "Name" column in the Source table. The

Text.Middle

Overview of the Text.Middle Function

The Text.Middle function is used to extract a specific number of characters from the middle of a text string. It takes three arguments:

– Text: The text string from which to extract the characters.

– Start: The position of the first character to extract.

– Number of characters: The number of characters to extract.

The syntax of the Text.Middle function is as follows:

= Text.Middle(Text, Start, Number of characters)

For example, if we have a text string “Lorem ipsum dolor sit amet”, and we want to extract 5 characters from the middle starting at position 7, we would use the following formula:

= Text.Middle("Lorem ipsum dolor sit amet", 7, 5)

This would return the text string “ipsum”.

Understanding the M Code Behind Text.Middle

The Text.Middle function is a built-in function in Power Query that is written in the M language. The M language is a functional programming language that is used to create custom functions and manipulate data in Power Query. Understanding the M code behind the Text.Middle function can help users to create more complex data transformations and custom functions.

The M code behind the Text.Middle function is as follows:

(Text as text, Start as number, Count as number) =>

Text.NewGuid

The Basics of GUIDs

Before we dive into the M code, let's first go over some basics about GUIDs. A GUID, or globally unique identifier, is a 128-bit value that is guaranteed to be unique across time and space. This means that even if two different systems generate GUIDs at the same time, the probability of those GUIDs being the same is extremely low.

GUIDs are often used in computer systems to create a unique identifier for each object or record. For example, if you have a table of customer data, you might use a GUID to create a unique identifier for each customer. This can be useful when you need to merge tables or track changes over time.

The M Code Behind Text.NewGuid

Now, let's take a look at the M code behind the Text.NewGuid function. In Power Query, you can open the Advanced Editor to view the M code for any query. Here's the M code for the Text.NewGuid function:

```
let
  Source = "",
  Guid = Guid.NewGuid(),
  TextGuid = Text.From(Guid)
in
  TextGuid
```

Let's break this code down line by line:

– The first line creates a variable called "Source" and sets it to an empty string. This variable is not used in the rest of the code, but it's included here for completeness.

– The second line creates a variable called "Guid" and sets it to the result of the Guid.NewGuid() function. This function is part of the .NET Framework and generates a new GUID each time it's called.

– The third line creates a variable called "TextGuid" and sets it to the result of the Text.From function, which converts the GUID to a text string.

Text.PadEnd

What is Text.PadEnd?

The Text.PadEnd function is used to add padding characters to the right side of a text string. This is useful when you want to align text columns or when you want to add a certain number of characters to the end of a string. The function takes three arguments:

– Text: The text string to which padding characters will be added.

– Length: The total length of the resulting string after padding.

– Character: The character to be used for padding.

For example, if we want to add 5 spaces to the end of the text string “hello”, we can use the following formula:

```
Text.PadEnd("hello", 10, " ")
```

This will result in the string “hello   ” (with 5 spaces added to the end).

The M Code Behind Text.PadEnd

The Text.PadEnd function is implemented in M code, which is the programming language used in Power Query. The M code behind the function is quite simple:

```
let
  PadEnd = (Text as text, Length as number, Character as text) as text =>
    let
      Padding = Text.Repeat(Character, Length - Text.Length(Text)),
      Result = Text & Padding
    in
      Result
in
  PadEnd
```

Text.PadStart

What is the Text.PadStart Function?

The Text.PadStart function is a built-in function in Power Query that is used to add leading characters to a text string to make it a specified length. It takes three arguments:

– Text: The text string to which leading characters will be added.

– Length: The desired length of the resulting text string.

– Character: The character to be used for padding.

The Text.PadStart function is particularly useful when working with data sets where values need to be standardized. For example, if you have a data set that includes product codes that are all supposed to be six characters long, but some are only four characters long, you can use the Text.PadStart function to add leading zeros to the shorter codes.

The M Code Behind Text.PadStart

The M code behind the Text.PadStart function is relatively simple. Here is an example of what the code might look like:

```
(Text as text, Length as number, Character as text) =>
let
   Padding = Character & Text,
   PaddingLength = List.Count(Padding),
   Result = if PaddingLength >= Length then Text else Text.PadStart(Padding, Length, Character)
in
   Result
```

The code takes the three arguments passed to the function and uses them to add padding to the text string. The first line of the code concatenates the padding character with the text string, creating a new string that is one character longer than the original. The next line counts the number of characters in the new string, which determines whether padding needs to be added or not. If the length of the new string is greater than or equal to the desired length, the function returns the original text string. If not, the Text.PadStart function is used to add the necessary padding characters to the beginning of the string to make it the desired length.

Text.PositionOf

What is Text.PositionOf?

The Text.PositionOf function is used to find the position of a substring in the given text. The syntax for this function is as follows:

Text.PositionOf(text as nullable text, substring as text, optional occurrence as nullable number) as nullable number

In this syntax, the first argument ‘text’ is the text in which we want to find the position of the substring. The second argument ‘substring’ is the text that we want to find in the given text. The third argument ‘occurrence’ is an optional argument that specifies the occurrence of the substring that we want to find. If the occurrence is not specified, the function returns the position of the first occurrence of the substring.

The M code behind Text.PositionOf

When we use the Text.PositionOf function in Power Query, it generates the following M code:

```
(text as nullable text, substring as text, optional occurrence as nullable number) =>
let
  pos = if occurrence = null then Text.PositionOf(text, substring) else Text.PositionOf(text, substring, occurrence)
in
  pos
```

This M code can be divided into two parts – the function definition and the function body.

Function definition

The function definition specifies the arguments of the function and their respective data types. In the case of Text.PositionOf, the function definition is as follows:

Text.PositionOfAny

One of the many functions available in the M language is Text.PositionOfAny. This function is used to find the first occurrence of any character from a given set of characters in a text string. In this article, we'll explore the M code that powers this function and how it can be used to transform data in Power Query.

Understanding the Text.PositionOfAny Function

Before diving into the M code behind the Text.PositionOfAny function, it's important to understand its basic syntax and functionality. The function takes two arguments: the first is the text string to be searched, and the second is a list of characters to search for.

For example, if we wanted to find the position of the first occurrence of either "a" or "b" in the text string "abcde", we would use the following formula:

```
Text.PositionOfAny("abcde", {"a", "b"})
```

This would return the value 1, since "a" is the first character in the list of search characters and appears at position 1 in the text string.

The M Code Behind Text.PositionOfAny

The M code behind the Text.PositionOfAny function is relatively simple. Here's the basic structure of the function:

```
(Text as text, List as list) as nullable number =>
let
   Positions = List.Transform(List, each Text.PositionOf(Text, _, Occurrence.First)),
   FirstPosition = List.Min(Positions)
in
   if FirstPosition = null then null else FirstPosition + 1
```

Let's break down each line of the code to understand how it works.

Text.Proper

Understanding the Text.Proper Function

Before we dive into the M code behind Text.Proper, let's first understand how the function works. The Text.Proper function takes a text value as input and converts it to a proper case format where the first letter of each word is capitalized, and the rest of the letters are in lowercase. For example, the input "JOHN DOE" would be converted to "John Doe".

The function works by first converting all letters in the input text to lowercase. It then searches for spaces in the text and converts the next letter to uppercase. This process is repeated for each space found in the text.

The M Code Behind Text.Proper

The M code behind the Text.Proper function is relatively simple. It consists of two main steps:

1. Convert the input text to lowercase
2. Capitalize the first letter of each word

Let's explore each of these steps in more detail.

Converting the Input Text to Lowercase

The first step in the Text.Proper function is to convert the input text to lowercase. This is done using the Text.Lower function. Here is the M code for this step:

```
let
  LowercaseText = Text.Lower(Text),
in
  LowercaseText
```

In this code, "Text" is the input parameter to the Text.Proper function. The Text.Lower function is used to convert the input text to lowercase and store it in the variable "LowercaseText". This variable is then used in the next step to capitalize the first letter of each word.

Capitalizing the First Letter of Each Word

The second step in the Text.Proper function is to capitalize the first letter of each word. This is done using the Text.Combine and Text.Range

Anatomy of the Text.Range Function

The Text.Range function takes three arguments: the text value, the starting position of the substring, and the length of the substring. Here is the basic syntax of the Text.Range function:

```
Text.Range(text as text, start as number, count as number) as text
```

– `text` is the text value from which you want to extract a substring.
– `start` is the starting position of the substring. The first character in the text value has a position of 0.
– `count` is the length of the substring.

For example, the following formula extracts the first three characters from the text value “Hello, World!”:

```
Text.Range("Hello, World!", 0, 3)
```

The result is “Hel”.

Advanced Features of the Text.Range Function

The Text.Range function has some advanced features that allow you to extract substrings based on more complex criteria.

Extracting a Substring based on a Delimiter

Often, you may want to extract a substring based on a delimiter, such as a comma or a space. You can achieve this by combining the Text.Range function with the Text.PositionOf function, which returns the position of a substring within a text value.

For example, the following formula extracts the first word from the text value “Hello, World!”:

```
Text.Range("Hello, World!", 0, Text.PositionOf("Hello, World!", ","))
```

Text.Remove

What is the Text.Remove Function?

The Text.Remove function is a Power Query M function that is used to remove a specified number of characters from a given text string. It takes two arguments: the text string to be modified and the number of characters to remove. Here is the syntax of the Text.Remove function:

```
Text.Remove(text as nullable text, count as number) as nullable text
```

The first argument, “text”, is the text string that you want to modify. The second argument, “count”, is the number of characters that you want to remove from the beginning of the text string. If the second argument is negative, the Text.Remove function will remove characters from the end of the text string.

How to Use the Text.Remove Function

Using the Text.Remove function is simple. First, you need to create a query in Power Query. Once you have your query, you can add the Text.Remove function to the query by following these steps:

1. Select the column that you want to modify in the query editor
2. Click on the “Add Column” tab in the ribbon
3. Click on the “Text” dropdown menu
4. Select the “Remove” option
5. Enter the number of characters to remove in the “Number of Characters” field

Here is an example of how to use the Text.Remove function to remove the first three characters from a text string:

```
= Table.AddColumn(#"Previous Step", "New Column", each Text.Remove([Old Column], 3))
```

This formula will create a new column called “New Column” that contains the original text string with the first three characters removed.

Text.RemoveRange

What is the Text.RemoveRange function?

The Text.RemoveRange function is a Power Query M function that removes a specified number of characters from a text string, starting from a specified position. It takes two arguments: the text string and the number of characters to remove.

The syntax of the Text.RemoveRange function is as follows:

```
Text.RemoveRange(text as nullable text, offset as number, count as number) as nullable text
```

The “text” argument is the text string from which characters are to be removed. The “offset” argument is the position from which the characters are to be removed. The “count” argument is the number of characters to be removed.

How to use the Text.RemoveRange function?

To use the Text.RemoveRange function, follow the steps below:

1. Load the data into Power Query.
2. Select the column that contains the text string.
3. Click on the “Add Column” tab.
4. Click on “Custom Column”.
5. In the “Custom Column” dialog box, enter a name for the new column.
6. In the “Custom Column” dialog box, enter the following formula:

```
Text.RemoveRange([column name], [offset], [count])
```

Replace “[column name]” with the name of the column that contains the text string, “[offset]” with the position from which the characters are to be removed, and “[count]” with the number of characters to be removed.

7. Click on “OK”. The new column will be added to the table.

Text.Repeat

Understanding the Text.Repeat Function

The Text.Repeat function is used to repeat a specific text value a certain number of times. The syntax for the Text.Repeat function is as follows:

```
Text.Repeat(text as nullable text, count as number) as nullable text
```

The function takes two arguments: ‘text’ and ‘count’. The ‘text’ argument is the value you want to repeat, and the ‘count’ argument specifies how many times you want to repeat it.

The M Code Behind Text.Repeat

Now that we understand the basic syntax of the Text.Repeat function, let’s take a look at the M code behind it. The M code for the Text.Repeat function is as follows:

```
(Text as nullable text, Count as number) =>
  let
    Source = List.Repeat({Text}, Count),
    #"Converted to Table" = Table.FromList(Source, Splitter.SplitByNothing(), {"Text"}),
    #"Changed Type" = Table.TransformColumnTypes(#"Converted to Table",{{"Text", type text}})
  in
    #"Changed Type"{0}[Text]
```

Let’s break this code down into smaller sections to understand what’s happening at each step.

Defining the Function Parameters

The first line of the code defines the function parameters. This is where we specify the ‘text’ and ‘count’ arguments for the Text.Repeat

Text.Replace

What is Power Query?

Power Query is a data transformation and cleaning tool that is included in Microsoft Excel and Power BI. It allows users to connect to various data sources, clean and transform data, and load it into Excel or Power BI for analysis. Power Query uses a functional language called M, which is used to create custom data transformations.

The Text.Replace Function

The Text.Replace function is used to replace a specific text value with another value. The syntax for the Text.Replace function is as follows:

Text.Replace(text as nullable text, old_text as nullable text, new_text as nullable text, optional comparer as nullable function) as nullable text

The parameters for the Text.Replace function are as follows:

– text: The text string that you want to replace a value in.
– old_text: The text value that you want to replace.
– new_text: The new text value that will replace the old text value.
– comparer: An optional function that can be used to specify the comparison mode for the search.

Examples

To better understand how the Text.Replace function works, let's look at some examples.

Example 1

Suppose we have a table that contains product names, and we want to replace the word "widgets" with "gadgets". We can use the Text.Replace function to accomplish this as follows:

= Table.ReplaceValue(#"PreviousStep", "widgets", "gadgets", Replacer.ReplaceText,{"Product Name"})

Text.ReplaceRange

How Does `Text.ReplaceRange` Work?

The `Text.ReplaceRange` function takes three arguments: the text string, the starting index of the range to replace, and the number of characters to replace. Here is the syntax of the function:

Text.ReplaceRange(text as nullable text, offset as number, count as number, newtext as text) as nullable text

Let’s say we have the following text string:

"This is a test string"

If we want to replace the word “test” with “sample”, we can use the `Text.ReplaceRange` function like this:

Text.ReplaceRange("This is a test string", 10, 4, "sample")

In this case, the starting index of the range we want to replace is 10 (which is the index of the first character of the word “test” in the text string), and we want to replace 4 characters (which is the length of the word “test”). The result of this function call would be:

"This is a sample string"

Text.Reverse

In this article, we will explore the M code behind the Text.Reverse function and learn how it can be used to manipulate text in Power Query.

Understanding the M Language

Before we dive into the M code behind Text.Reverse, let's first understand what the M language is. M is the formula language used in Power Query, which is a functional, case-sensitive, and highly expressive language that supports a wide range of data manipulation operations.

M code is created by using the Power Query Editor, which provides an intuitive interface for creating custom queries and transformations. M code is generated automatically as we create queries in the Editor, and we can also edit the code directly to create more complex transformations.

Power Query is designed to be a user-friendly tool that does not require extensive knowledge of coding, but understanding the M language can be helpful in creating more efficient and customized queries.

The Text.Reverse Function

The Text.Reverse function is a built-in function in Power Query that allows us to reverse the order of characters in a given text string. The syntax for this function is as follows:

```
Text.Reverse(text as nullable text) as nullable text
```

The function takes a text value as input and returns the same text value with the order of characters reversed. If the input value is null, the function returns null.

Let's take a look at an example to understand how the Text.Reverse function works:

```
Text.Reverse("hello world")
```

Text.Select

What is Text.Select?

The Text.Select function is used to extract a substring from a text value. It takes two arguments, the text value and a range. The range specifies the starting position and the length of the substring to extract. The Text.Select function is commonly used in data cleaning tasks where we need to extract specific parts of a text value.

M Code Behind Text.Select

The M code behind the Text.Select function is straightforward. Let's take a look at an example:

```
(Text as text, Start as number, Count as number) as text =>
let
  SubText = if Start > 0 then Text else "",
  SubText2 = if Count > 0 then Text.Range(SubText, Start, Count) else ""
in
  SubText2
```

The Text.Select function takes three parameters, Text, Start, and Count. The first parameter 'Text' is the input text value. The second parameter 'Start' is the starting position of the substring to extract. The third parameter 'Count' is the length of the substring to extract.

The Text.Select function starts with a 'let' statement, which creates a new variable called 'SubText.' The 'if' statement checks if the starting position is greater than zero. If it is, then the 'SubText' variable is set to the input text value. Otherwise, it is set to an empty string.

The next 'let' statement creates another variable called 'SubText2.' The 'if' statement checks if the length of the substring to extract is greater than zero. If it is, then the 'SubText2' variable is set to the substring extracted using the Text.Range function. Otherwise, it is set to an empty string.

Finally, the Text.Select function returns the 'SubText2' variable, which contains the extracted substring.

Examples

Let's take a look at some examples of how to use the Text.Select function in Power Query.

Text.Split

In this article, we will explore the M code behind the Text.Split function and how it works. We will also take a look at some tips and tricks for using Text.Split in your Power Query projects.

How Text.Split Works

The Text.Split function in Power Query M takes two arguments: the text string to split and the delimiter to use for splitting. The function returns an array of substrings that were separated by the delimiter.

For example, consider the following text string: “John,Smith,32”. If we want to split this string into three substrings (i.e., “John”, “Smith”, “32”), we can use the Text.Split function with a comma delimiter as follows:

```
Text.Split("John,Smith,32", ",")
```

This will return the following array of substrings:

```
{ "John", "Smith", "32" }
```

It is important to note that Text.Split is case-sensitive, which means that it will only split the string based on the exact delimiter that you specify. If you want to split a string based on multiple delimiters, you can use the Text.SplitAny function instead.

Tips and Tricks for Using Text.Split

Here are some tips and tricks for using Text.Split in your Power Query projects:

Trim Whitespace

When splitting a string, you may encounter whitespace characters (e.g., spaces, tabs, newlines) before or after the delimiter. To remove these whitespace characters, you can use the Text.Trim function in combination with Text.Split.

For example, consider the following text string: “John, Smith ,32”. If we split this string using a comma delimiter, we will get the following array of substrings:

Text.SplitAny

Understanding Text.SplitAny

The Text.SplitAny function takes two arguments: the text string to split and a list of delimiters to use. The function splits the text string into an array of sub-strings based on any of the delimiters in the list. For example, the following code splits the text string “apple,banana,pear” into an array of three sub-strings:

```
let
  textString = "apple,banana,pear",
  delimiterList = {",", ";"}, // Delimiter list can be a single delimiter or a list of delimiters
  result = Text.SplitAny(textString, delimiterList)
in
  result
```

The resulting array contains three sub-strings: “apple”, “banana”, and “pear”.

The M Code Behind Text.SplitAny

The M code behind the Text.SplitAny function is relatively simple. The function takes the text string to split and the list of delimiters as arguments, then iterates over each character in the text string, checking to see if it matches any of the delimiters. If a delimiter is found, the current sub-string is added to the output array and the next sub-string is started.

Here is the M code for the Text.SplitAny function:

```
(Text as text, Delimiters as list) as list =>
  let
    SplitByDelimiters = (Text as text, Delimiters as list) as list =>
      let
        DelimiterCount = List.Count(Delimiters),
Text.Start
```

Syntax
The syntax for the Text.Start function is as follows:

```
Text.Start(text as text, count as number) as text
```

The first argument, `text`, is the text string from which you want to extract characters. The second argument, `count`, is the number of characters you want to extract from the beginning of the text string.
Example
Let's say we have a table named `Sales` with a column named `Product Name`. We want to create a new column named `Short Name` that contains the first two characters of the product name. We can use the Text.Start function to achieve this as follows:

```
= Table.AddColumn(Sales, "Short Name", each Text.Start([Product Name], 2))
```

This will create a new column named `Short Name` in the `Sales` table with the first two characters of the `Product Name` column.
How it Works
The Text.Start function takes two arguments: `text` and `count`. The `text` argument is the text string from which you want to extract characters, and the `count` argument is the number of characters you want to extract from the beginning of the text string.
When the Text.Start function is called, it returns a new text string that contains the first `count` characters of the `text` argument.
The Text.Start function is a powerful tool in the M language that can be used to extract a specified number of characters from the beginning of a text string. It is a useful function for data transformation and manipulation tasks in Power Query. When used in conjunction with other M functions, it can help you quickly and easily transform your data to meet your needs.

Text.StartsWith

What is Power Query?

Power Query is a data transformation and cleansing tool that allows users to extract, transform, and load data from various sources. It is a part of Microsoft's Power BI suite of products and is also available as an add-in for Excel. Power Query uses a functional language called M to perform data transformations.

Understanding the `Text.StartsWith` function

The `Text.StartsWith` function is used to check if a text value starts with a specific substring. The function takes two arguments: the text value to check and the substring to search for. If the text value starts with the specified substring, the function returns `True`. Otherwise, it returns `False`.

The syntax for the `Text.StartsWith` function is as follows:

```
Text.StartsWith(text as nullable text, substring as text) as logical
```

Here, `text` is the text value to check, and `substring` is the substring to search for.

The M Code Behind `Text.StartsWith`

The M code behind the `Text.StartsWith` function is quite simple. In fact, it is just a single line of code. Here is the M code for the `Text.StartsWith` function:

```
(text as nullable text, substring as text) => Text.Start(text, Text.Length(substring)) = substring
```

The M code defines an anonymous function that takes two arguments (`text` and `substring`) and returns a logical value (`True` or `False`). The function uses two other M functions: `Text.Start` and `Text.Length`.

The `Text.Start` function takes two arguments: the text value to extract a substring from, and the number of characters to extract. In this case, we want to extract the first `n` characters from the text value, where `n` is the length of the substring we are searching for.

Text.ToBinary

What is Text.ToBinary?

Before we dive into the M code behind Text.ToBinary, let's take a moment to understand what this function does. Text.ToBinary is a Power Query M function that converts a text string into a binary representation. This binary representation can then be used for various purposes such as encryption, compression, or storage.

The M Code Behind Text.ToBinary

The M code behind Text.ToBinary is relatively simple. Here is the code:

```
(Text as text, optional Encoding as nullable number) as binary =>
Binary.FromText(Text, Encoding)
```

As you can see, the Text.ToBinary function takes two arguments: Text and Encoding. The Text argument is the text string that you want to convert into a binary representation. The Encoding argument is optional and specifies the encoding to use for the conversion. If the Encoding argument is not provided, the function will use the default encoding for the current locale.

The core of the Text.ToBinary function is the Binary.FromText function. This function takes the text string and converts it into a binary representation. The optional Encoding argument specifies the encoding to use for the conversion. If the Encoding argument is not provided, the function will use the default encoding for the current locale.

Using Text.ToBinary

Using Text.ToBinary is straightforward. Here is an example:

```
let
  textString = "Hello, world!",
  binaryRepresentation = Text.ToBinary(textString)
in
  binaryRepresentation
```

Text.ToList

One of the most useful M functions in Power Query is Text.ToList, which allows users to split a text string into a list of characters. This can be incredibly helpful when working with data that needs to be broken down into individual items, such as email addresses or product SKUs.

How to Use Text.ToList

The syntax for Text.ToList is simple:

```
Text.ToList(text as nullable text) as list
```

The function takes a single argument, which is the text string that you want to split into a list. For example, if you have a column of email addresses that you want to split into individual characters, you would use the following formula:

```
Text.ToList([Email])
```

This will create a list of characters for each email address in your data set.

Understanding the M Code

While the Text.ToList function is easy to use, understanding the M code behind it can be helpful for more advanced users who want to create more complex transformations. Here's a breakdown of the M code for Text.ToList:

```
(text as nullable text) as list =>
  if text = null then {} else List.Transform(Text.ToList(text), each _)
```

Text.Trim

Understanding Text.Trim

Before we dive into the M code behind `Text.Trim`, let's take a moment to understand what this function does. When you apply `Text.Trim` to a text value, it removes any spaces, tabs, or line breaks at the beginning or end of the value. For example, if you had a text value that looked like this:

```
 Hello World
```

Applying `Text.Trim` would result in:

```
Hello World
```

This can be useful for cleaning up text values that might contain extra spaces or other whitespace characters that could cause issues with further analysis or manipulation.

The M Code Behind Text.Trim

Now let's take a closer look at the M code behind `Text.Trim`. Here's the basic syntax for the function:

```
Text.Trim(text as nullable text, optional trimChars as nullable text) as text
```

The first argument, `text`, is the text value that you want to trim. The second argument, `trimChars`, is an optional argument that allows you to specify additional characters that should be trimmed from the text value. By default, `Text.Trim` only removes spaces, tabs, and line breaks.

Text.TrimEnd

One of the most commonly used functions in M is Text.TrimEnd. This function allows you to remove a specific character or set of characters from the end of a string. In this article, we will explore the M code behind the Text.TrimEnd function and how you can use it to manipulate your data.

Understanding Text.TrimEnd Function

The Text.TrimEnd function is used to remove a specific set of characters from the end of a string. This function takes two arguments, the first is the string that you want to remove characters from, and the second is the set of characters that you want to remove.

For example, if you want to remove the letter “a” from the end of the string “banana,” you would use the Text.TrimEnd function as follows:

```
Text.TrimEnd("banana", "a")
```

The output of this function would be the string “banan.”

The M Code Behind Text.TrimEnd

The M code behind the Text.TrimEnd function is quite simple. When you use the Text.TrimEnd function, Power Query generates the following M code:

```
(Text as text, Optional TrimChars as nullable text) as text =>
let
  TrimCharsValue = if TrimChars = null then {" "} else Text.ToList(TrimChars),
  LastNonTrimChar = List.PositionOf(List.Reverse(Text.ToList(Text)), each not List.Contains(TrimCharsValue, _), 0),
  TrimmedText = Text.Start(Text, LastNonTrimChar+1)
in
  TrimmedText
```

Text.TrimStart

Understanding the M Language

To understand the M code behind the Text.TrimStart function, we first need to understand the M language. M is the language used by Power Query to manipulate data, and it's a functional language that uses expressions and functions to transform data.

M is a case-sensitive language, which means that Text.TrimStart and text.trimstart are not the same thing. It's also a strongly typed language, which means that all values are assigned a specific data type (such as text, number, or date) and can only be manipulated using functions that operate on that data type.

The Text.TrimStart Function

Now let's take a closer look at the Text.TrimStart function itself. The function takes two arguments: the text string to be trimmed, and the character or set of characters to be removed from the beginning of the string.

Here's an example of how the function works:

```
Text.TrimStart("   hello world", " ")
```

In this example, we're passing in the string "   hello world" (with three spaces at the beginning) and the character " " (a single space). The function will remove the leading spaces and return the string "hello world".

The M Code Behind Text.TrimStart

So what does the M code behind Text.TrimStart look like? Here's the code:

```
(text as text, optional trimCharacter as text) =>
  let
    trimChars = if trimCharacter = null then {" "} else {trimCharacter},
    trimStart = Text.TrimStart(text, trimChars)
  in
    trimStart
Text.Upper
```

In this article, we will explore the M code behind the Text.Upper function and how it can be used to manipulate text data.

What is Text.Upper?

Text.Upper is a function in Power Query that is used to convert text to uppercase. It takes a text value as an input and returns the same text value in uppercase. For example, if the input text is “hello world”, the output of Text.Upper will be “HELLO WORLD”.

The syntax for using Text.Upper is as follows:

```
Text.Upper(text as nullable text) as nullable text
```

Understanding the M Code Behind Text.Upper

The M code behind Text.Upper is relatively simple. It consists of a single line of code that makes use of the Text.Upper function:

```
(Text.Upper([Text]))
```

This code takes an input text value and passes it to the Text.Upper function. The Text.Upper function then converts the text to uppercase and returns the result.

Let’s break down this code into its individual components:

– The outer parentheses indicate that we are creating a new expression.

– The Text.Upper function is enclosed in parentheses, indicating that it is a function call.

– The [Text] parameter is the input text value that will be converted to uppercase.

Using Text.Upper in Power Query

Now that we understand the M code behind Text.Upper, let’s take a look at how it can be used in Power Query.

Suppose we have a table of customer data that includes a column for the customer’s first name. We want to create a new column that contains the first name in uppercase. Here’s how we can do it:

Time.EndOfHour

Understanding Power Query M Function

Power Query M function is a language used for data transformation and analysis. It is used in Microsoft Power BI, Power Query, and Excel. The M language is designed to be user-friendly and easy to read, but it can also be complex and powerful.

In Power Query, M functions are used to transform data. The functions can be simple, such as adding columns or filtering data, or complex, such as grouping data and creating custom functions. The Time.EndOfHour function is one of the many built-in functions in Power Query M that can be used to manipulate time-based data.

Syntax of Time.EndOfHour

The syntax of Time.EndOfHour is simple and easy to understand. The function takes one argument, which is a datetime value. The function returns the last second of the current hour.

The syntax of the Time.EndOfHour function is as follows:

```
Time.EndOfHour(dateTime as any) as datetime
```

The argument dateTime is any datetime value.

M Code Behind Time.EndOfHour

To understand the M code behind the Time.EndOfHour function, let's first understand the concept of datetime in Power Query. A datetime value in Power Query is a combination of a date and a time. The date and time are represented as numbers in Power Query. The date is represented as the number of days after December 30, 1899, while the time is represented as the decimal part of a day. For example, noon is represented as 0.5 in Power Query.

The M code behind the Time.EndOfHour function is simple. It takes the datetime value as an argument and converts it to the end of the hour. Here is the M code behind the Time.EndOfHour function:

```
(dateTime as any) => DateTime.DateAdd(DateTime.LocalNow(), "hour", DateTime.Hour(dateTime)+1)-#duration(0,0,1,0)
```

Time.From

What is M code?

M code is the programming language used in Power Query to transform data. It is a functional language that is used to define queries, which are then executed against data sources. M code is similar to other functional programming languages, such as F# and Haskell, and it is designed to be easy to read and understand.

How does the Time.From function work?

The Time.From function takes a datetime value as its input and returns a time value. The datetime value can be any valid datetime value, such as a date and time value, or a datetimezone value. The time value that is returned by the function represents the time component of the input datetime value.

The M code behind the Time.From function is relatively simple. Here is the code:

```
(Time as datetime) => Time.TimeOfDay
```

The function takes a single argument, which is a datetime value, and returns the TimeOfDay property of the datetime value. The TimeOfDay property is a time value that represents the time component of the datetime value.

Examples

Let's take a look at some examples of how the Time.From function works. Suppose we have a table of datetime values, and we want to extract the time component of each value. We can use the Time.From function to achieve this:

```
let
  Source = Table.FromRows({{#datetime(2022, 1, 1, 12, 30, 0)}, {#datetime(2022, 1, 1, 14, 45, 0)}, {#datetime(2022, 1, 1, 16, 0, 0)}}),
  #"Renamed Columns" = Table.RenameColumns(Source,{{"Column1", "DateTime"}}),
  #"Added Custom" = Table.AddColumn(#"Renamed Columns", "Time", each Time.From([DateTime]))
in
  #"Added Custom"
```

Time.FromText

Understanding the Time.FromText Function

The Time.FromText function is a built-in Power Query function that allows users to convert text strings into time values. The function takes a single argument, which is the text string that contains the time data. The function then applies a series of transformations to the text string to extract the relevant time data, and returns a time value that can be used in calculations and analysis.

The Time.FromText function is particularly useful for working with time data that is stored in non-standard formats, or that needs to be extracted from other data sources. For example, if you have a dataset that contains time data in a variety of formats, the Time.FromText function can be used to standardize the data and convert it into a consistent format.

The M Code Behind the Time.FromText Function

The Time.FromText function is based on a series of M code transformations that are applied to the input text string. These transformations are designed to extract the relevant time data from the input string, and convert it into a time value that can be used for analysis.

The first step in the Time.FromText function is to extract the hour, minute and second values from the input text string. This is done using a series of text manipulation functions, which extract the relevant substrings from the input string. Once the hour, minute and second values have been extracted, they are converted into decimal values, and added together to create a time value.

The Time.FromText function also includes a number of error-handling features, which ensure that the function returns a valid time value even if the input text string contains errors or is in an unexpected format. For example, if the input string contains a time value that is greater than 24 hours, the function will automatically adjust the time value to the correct format.

Using the Time.FromText Function

The Time.FromText function can be used in a variety of ways to extract and manipulate time data. For example, the function can be used to convert time data from a variety of formats into a consistent format that can be used for analysis. The function can also be used to extract time data from other data sources, such as text files or databases.

To use the Time.FromText function, simply pass the input text string as a parameter to the function. The function will then apply the relevant transformations and return a time value that can be used for analysis. The time value can be used in calculations and analysis just like any other time value, and can be formatted and displayed in a variety of ways.

The Time.FromText function is a powerful tool for working with time data in Power Query. By understanding the M code behind the function, users can gain a deeper understanding of how the function works, and how it can be used to extract and manipulate time data. Whether you are working with time data in a variety of formats, or need to extract time data from other data sources, the Time.FromText

Time.Hour

What is the Time.Hour function?

The Time.Hour function is used to extract the hour component from a given time value. The function takes a time value as input and returns the hour component as an integer. For example, if the input time value is 10:30:45 AM, the Time.Hour function will return 10.

Syntax of the Time.Hour function

The syntax of the Time.Hour function is as follows:

```
Time.Hour(dateTime as any) as any
```

The function takes a single parameter, dateTime, which represents the input time value. The parameter can be of any data type that can be converted to a time value. The function returns the hour component of the input time value as an integer.

Understanding the M code behind the Time.Hour function

The M code behind the Time.Hour function is relatively straightforward. The function simply extracts the hour component from the input time value using the DateTime.LocalNow() function and the DateTime.TimeOfDay property. Here's the M code for the Time.Hour function:

```
(Time as any) =>
let
  Source = DateTime.LocalNow(),
  TimeOfDay = Time - DateTime.Date(Source),
  Hour = TimeOfDay / #duration(0, 1, 0, 0),
in
  Hour
```

Time.Minute

One of the most commonly used functions in Power Query is the Time.Minute function. This function is used to extract the minute component from a given time value. In this article, we will take a closer look at the M code behind this function and explore some practical examples of how it can be used.

Understanding the Time.Minute Function

Before we dive into the M code behind the Time.Minute function, let's first take a closer look at how this function works. The Time.Minute function takes a time value as its input and returns the minute component of that time value. For example, if the input time value is 3:15 PM, the Time.Minute function will return 15.

Here is the basic syntax of the Time.Minute function:

```
Time.Minute(time as any) as any
```

The 'time' argument is the input time value that you want to extract the minute component from. This argument can be any valid time value, such as a text string or a datetime value.

The M Code Behind Time.Minute

Now that we have a basic understanding of how the Time.Minute function works, let's take a closer look at the M code behind this function. Here is the M code that is used to define the Time.Minute function:

```
(Time) =>
   Duration.Minutes(Time.TimeOfDay)
```

This code defines a lambda function that takes a 'Time' argument and returns the minute component of that time value. The 'Time.TimeOfDay' property is used to extract the time component of the input 'Time' value, and the 'Duration.Minutes' function is used to extract the minute component from that time value.

Time.Second

The Time.Second function is a part of the M language, which is used in Power Query to create custom functions and manipulate data. In this article, we will explore the M code behind the Time.Second function and how it works.

The Syntax of Time.Second Function

The Time.Second function has a simple syntax. It takes a time value as an argument and returns the number of seconds since midnight of that time. Here is the syntax of the Time.Second function:

```
Time.Second(time as any) as any
```

The “time” argument can be any valid time value, including a date/time value, a time value, or a datetimezone value.

How Time.Second Works

The Time.Second function works by converting the given time value to the number of seconds since midnight. Here is the M code behind the Time.Second function:

```
(Time.Hour(time)  3600) + (Time.Minute(time)  60) + Time.Second(time)
```

As you can see, the M code first calculates the number of seconds in the hours component of the time value by multiplying it by 3600. It then calculates the number of seconds in the minutes component of the time value by multiplying it by 60. Finally, it adds the number of seconds in the seconds component of the time value.

The result is the total number of seconds since midnight of the given time value.

Examples of Using Time.Second Function

Let’s look at some examples of using the Time.Second function in Power Query.

Example 1: Using Time.Second with a Time Value

Suppose we have a time value of 10:30:15 AM. We can use the Time.Second function to find the number of seconds since midnight of

Time.StartOfHour

One such function is the Time.StartOfHour function, which returns the start of the hour for a given time value. In this article, we will take a closer look at the M code behind the Time.StartOfHour function, and understand how it works.

Understanding the Time.StartOfHour Function

Before we dive into the M code behind the Time.StartOfHour function, let's first understand what the function does. The Time.StartOfHour function takes a time value as input, and returns the start of the hour for that time value.

For example, if we have a time value of 9:30 AM, the Time.StartOfHour function would return 9:00 AM. Similarly, if we have a time value of 2:45 PM, the function would return 2:00 PM.

Now that we understand what the Time.StartOfHour function does, let's take a closer look at the M code behind the function.

The M Code Behind the Time.StartOfHour Function

The M code behind the Time.StartOfHour function is relatively simple. Here is the code for the function:

```
(Time as datetime) =>
Time.FromText(Text.Start(Text.From(Time), "yyyy-MM-ddThh"))
```

Let's break down this code and understand what each part does.

The Input Parameter

The first part of the code defines the input parameter for the function:

```
(Time as datetime) =>
```

The input parameter for the Time.StartOfHour function is a datetime value, which represents a specific date and time.

Converting the Time Value to Text

The next part of the code converts the input time value to text:

```
Time.ToRecord
```

In this article, we will take a closer look at the M code behind the Time.ToRecord function and how it can be used in Power Query.

Understanding the Time.ToRecord Function

The Time.ToRecord function is a M function that is used to convert a time value to a record. It takes a single parameter, a time value, and returns a record that contains the hour, minute, and second values of the time.

Here is an example of how the Time.ToRecord function can be used in Power Query:

```
let
  timeValue = #time(12, 30, 0),
  recordValue = Time.ToRecord(timeValue)
in
  recordValue
```

The above M code will convert the time value “12:30:00” to a record that contains the hour, minute, and second values of the time.

The M Code Behind the Time.ToRecord Function

To understand the M code behind the Time.ToRecord function, we need to take a closer look at how the function is defined in Power Query.

Here is the M code for the Time.ToRecord function:

```
(Time as any) as record =>
  let
    hour = Time.Hour(Time),
    minute = Time.Minute(Time),
    second = Time.Second(Time)
  in
Time.ToText
```

Understanding the Time.ToText Function

Before we dive into the M code behind the Time.ToText function, it is important to understand how this function works. The Time.ToText function takes two arguments: a time value and a format string. The time value is the value that you want to convert into a text string, and the format string is the string that specifies how you want the time value to be displayed.

For example, if you have a time value of 12:30:00 PM and you want to display it in the format of “hh:mm:ss tt”, you can use the following formula:

```
= Time.ToText(#time(12,30,0),"hh:mm:ss tt")
```

This formula will return the text string “12:30:00 PM”, which represents the time value in the specified format.

The M Code Behind Time.ToText

Now that we have a basic understanding of how the Time.ToText function works, let’s take a closer look at the M code behind this function. The M code for the Time.ToText function is as follows:

```
(Time as any, optional Format as nullable text) as nullable text =>
  let
    TimeText = Text.From(Time) & "Z",
    DateTime = DateTimeZone.FromText(TimeText),
    Result = if (Format = null) then DateTime.ToText() else DateTime.ToText(Format)
  in
    Result
```

Let’s break down this code into its individual components.

Transform operations

Understanding the M Function in Power Query

The M function is a functional programming language that is used in Power Query to transform data. It is a case-sensitive language that uses functions, expressions, and operators to manipulate data. The M language is used to create formulas that can be used to perform complex transformations on data.

The M code is written in the Advanced Editor of Power Query. To access the Advanced Editor, click on the View tab, then click on Advanced Editor. This will open a window where you can write and edit M code.

In Power Query, there are two types of M functions: built-in functions and custom functions. Built-in functions are pre-defined functions that are available in Power Query, while custom functions are user-defined functions that are created using the M language.

Transform Operations in Power Query

Transform operations in Power Query are used to manipulate data in various ways. The following are some of the common transform operations in Power Query:

Filtering Data

Filtering data in Power Query is done using the Filter Rows transformation. This transformation allows you to filter data based on a specific condition. For example, you can filter data to show only records where the value in a particular column is greater than a certain value.

Sorting Data

Sorting data in Power Query is done using the Sort transformation. This transformation allows you to sort data based on one or more columns in the data set. For example, you can sort data based on the date column or the customer name column.

Grouping Data

Grouping data in Power Query is done using the Group By transformation. This transformation allows you to group data by one or more columns in the data set. For example, you can group data by the customer name column to show the total sales for each customer.

Aggregating Data

Aggregating data in Power Query is done using the Aggregate transformation. This transformation allows you to perform calculations on grouped data. For example, you can calculate the total sales for each customer group.

Pivot and Unpivot Data

Pivoting and unpivoting data in Power Query is done using the Pivot and Unpivot transformations. These transformations allow you to reshape data from a wide format to a long format or vice versa. For example, you can pivot data to show sales by product category or

Trigonometry

What is Trigonometry?

Trigonometry is a branch of mathematics that deals with the relationships between the sides and angles of triangles. It is used to solve problems in fields such as physics, engineering, and navigation. Trigonometric functions, such as sine, cosine, and tangent, are used to calculate these relationships.

The M Code Behind the Trigonometry Function

In Power Query, the Trigonometry function is used to calculate trigonometric values of angles in radians. The function takes two arguments: the angle and the type of trigonometric function to be calculated. The angle must be in radians, and the type of function can be one of the following: Sine, Cosine, Tangent, Cotangent, Secant, or Cosecant.

The M code behind the Trigonometry function is based on the mathematical formulas for calculating trigonometric values. For example, the formula for calculating the sine of an angle is sin(x) = opposite/hypotenuse, where x is the angle in radians, and opposite and hypotenuse are the lengths of the two sides of a right triangle.

In M code, the formula for calculating the sine of an angle is:

```
let
   sine = (x) => Number.Round(Number.Sin(x), 14),
   result = sine(angle)
in
   result
```

In this code, the sine function takes the angle in radians as its input and calculates the sine of that angle using the Number.Sin function. The result is then rounded to 14 decimal places using the Number.Round function.

The M code for the other trigonometric functions is similar, with slight variations based on the mathematical formulas for each function.

Using the Trigonometry Function in Power Query

To use the Trigonometry function in Power Query, you must first create a query that contains a column with the angle in radians. You can then use the Trigonometry function to calculate the trigonometric value of that angle.

Type.AddTableKey

Among the many M functions available in Power Query, the Type.AddTableKey function is one of the most important and widely used. In this article, we will explore the M code behind the Type.AddTableKey function and show how it can be used to create effective data transformations.

What is the Type.AddTableKey function?

The Type.AddTableKey function is a Power Query M function that is used to create a table key for a given table. A table key is a unique identifier for each row in a table that helps to ensure data integrity and accuracy. It is usually created by combining one or more columns in the table to create a unique value for each row.

The Type.AddTableKey function takes two arguments: the table to which the key will be added, and the list of column names that will be used to create the key. Here is an example of how the function is used:

```
let
  Source = Table.FromRecords({
    [ID = 1, Name = "John", Age = 30],
    [ID = 2, Name = "Jane", Age = 25],
    [ID = 3, Name = "Bob", Age = 40]
  }),
  AddKey = Table.AddTableKey(Source, {"ID"})
in
  AddKey
```

In this example, we create a table called “Source” with three columns: ID, Name, and Age. We then use the Type.AddTableKey function to add a key to the table based on the ID column. The resulting table will have an additional column called “__Key” that contains the unique identifier for each row.

How does the Type.AddTableKey function work?

The Type.AddTableKey function works by creating a new column in the table that contains a unique identifier for each row based on the

Type.ClosedRecord

Understanding the Type.ClosedRecord Function

The Type.ClosedRecord function is used to define a record type with a fixed set of fields. It is useful when you want to ensure that your data follows a predefined structure. For example, if you are working with financial data, you may want to ensure that each record contains fields for Date, Amount, and Description.

Here is an example of how to define a closed record type using the Type.ClosedRecord function:

```
Type.ClosedRecord [
  Field1 = type,
  Field2 = type,
  ...
]
```

In this example, Field1 and Field2 are the names of the fields in the record type, and type is the data type of each field. You can define as many fields as you need, separated by commas.

The M Code Behind the Type.ClosedRecord Function

The M code behind the Type.ClosedRecord function is relatively straightforward. When you define a closed record type, Power Query generates a record type value that contains the names and data types of each field.

Here is an example of the M code behind the closed record type we defined earlier:

```
#shared
let
  typeClosedRecord = Type.ClosedRecord [
    Field1 = type,
    Field2 = type
Type.Facets
```

What is Type.Facets?

Type.Facets is a Power Query M function that is used to apply specific constraints to a column. These constraints are referred to as facets. The function takes two arguments: the first argument is the column, and the second argument is a record that defines the facets to apply. The facets that can be applied include:

– Maximum: Specifies the maximum value of the column.
– Minimum: Specifies the minimum value of the column.
– Nullable: Specifies whether the column can contain null values.
– Unique: Specifies whether the column can contain duplicate values.
– Items: Specifies the data type of the items in a list column.
– TextEncoding: Specifies the text encoding of a text column.

M Code Behind Type.Facets

To understand the M code behind Type.Facets, we need to take a closer look at the syntax of the function. The syntax of the Type.Facets function is as follows:

```
Type.Facets(column as any, facets as record) as any
```

The first argument of the function is the column that we want to apply the facets to. The second argument is a record that defines the facets to apply. The record consists of one or more key-value pairs, where the key is the name of the facet, and the value is the value of the facet. For example, to apply a maximum constraint to a column, we use the following syntax:

```
Type.Facets(column, [Maximum = 100])
```

This will apply a maximum constraint of 100 to the column. If the column contains a value greater than 100, an error will be generated.

Type.ForFunction

In this article, we will explore the M code behind the Power Query M function Type.ForFunction. This function is commonly used to check if a given value is a function or not. Let's dive in!

What is Type.ForFunction?

Before we dive into the M code behind Type.ForFunction, let's understand what this function does. Type.ForFunction is a Power Query M function that checks if a given value is a function or not. If the given value is a function, it returns a function type. If the value is not a function, it returns null.

The syntax for Type.ForFunction is as follows:

```
Type.ForFunction(function as any) as nullable type
```

Here, the function parameter is the value that needs to be checked for its type. The return type is nullable, which means it can return a value or null.

The M Code Behind Type.ForFunction

The M code behind Type.ForFunction is relatively simple. Let's break it down step by step.

```
(Type.FunctionType(_, _) meta [IsType = true])(function)
```

The first part of this code, `(Type.FunctionType(_, _)`, creates a function type using the Type.FunctionType M function. This function takes two parameters, which are used to describe the function type. The first parameter is the list of function parameter types, and the second parameter is the return type of the function.

In our case, we are using a wildcard (_) for both the function parameter types and the return type. This is because we want to check if the given value is a function, without knowing the exact parameter types or return type.

The next part of the code, `meta [IsType = true])`, adds metadata to the function type. This metadata is used to mark the function type

Type.ForRecord

Introduction to Type.ForRecord Function

The Type.ForRecord function is used to create a record type value in M. A record is a collection of named values, where each value can be of a different type. For example, a record can contain a name (text), age (number), and address (text) fields. The Type.ForRecord function takes a list of field names and types as arguments and returns a record type value.

The syntax of the Type.ForRecord function is as follows:

```
Type.ForRecord({
  [FieldName1 = FieldType1],
  [FieldName2 = FieldType2],
  ...
})
```

The field names are optional, and if they are not provided, the fields are automatically named as Field1, Field2, and so on. The field types can be any valid M data types, such as text, number, logical, or record.

Understanding the M Code Behind Type.ForRecord

The M code behind the Type.ForRecord function is relatively simple. It creates a record type value based on the list of field names and types provided as arguments. Let's take a look at an example:

```
Type.ForRecord({
  [Name = Text.Type],
  [Age = Int32.Type],
  [Address = Text.Type]
})
```

Type.FunctionParameters

Understanding Type.FunctionParameters

Before we dive into the M code behind the Type.FunctionParameters function, let's first understand what it does. The Type.FunctionParameters function takes a function as input and returns a record that describes the parameters of the function. The record includes information such as the name, data type, and optional/default values of each parameter.

For example, let's say we have a function called "AddNumbers" that takes two parameters, "number1" and "number2". We can use the Type.FunctionParameters function to get information about these parameters as follows:

```
let
  func = (number1 as number, number2 as number) => number1 + number2,
  paramInfo = Type.FunctionParameters(func)
in
  paramInfo
```

The result of this query would be a record with two fields, "Parameters" and "ReturnType". The Parameters field would be another record with two fields, "number1" and "number2". Each of these fields would describe the corresponding parameter, including its name, data type, and any optional/default values.

The M Code Behind Type.FunctionParameters

Now that we understand what the Type.FunctionParameters function does, let's take a closer look at the M code behind it. Here is the M code for the Type.FunctionParameters function:

```
(type as type) =>
  let
    func = Value.Metadata(type){0},
    parameters = func[Parameters],
Type.FunctionRequiredParameters
```

Overview of the Type.FunctionRequiredParameters Function

Before we delve into the M code behind the Type.FunctionRequiredParameters function, let's first understand what this function does. The Type.FunctionRequiredParameters function is used to retrieve a list of required parameters for a given function. This function takes a single argument, which is the function to retrieve the required parameters for. The function returns a list of required parameters for the given function.

Syntax of the Type.FunctionRequiredParameters Function

The syntax of the Type.FunctionRequiredParameters function is as follows:

```
Type.FunctionRequiredParameters(function as any) as list
```

The function argument is the function for which to retrieve the required parameters. This argument can be a reference to a function, or it can be a function literal. The function returns a list of required parameters for the given function.

Example Usage of the Type.FunctionRequiredParameters Function

Now that we've covered the syntax of the Type.FunctionRequiredParameters function, let's explore an example usage of this function. Suppose we have a function named "MyFunction" that takes two required parameters:

```
let
  MyFunction = (param1 as any, param2 as any) => param1 + param2,
  RequiredParams = Type.FunctionRequiredParameters(MyFunction)
in
  RequiredParams
```

In this example, we define a function named "MyFunction" that takes two required parameters. We then use the Type.FunctionReturn

In this article, we'll take a closer look at the M code behind the Type.FunctionReturn function, exploring its syntax, behavior, and applications. We'll also provide some practical examples of how this function can be used to enhance your data analysis and visualization workflows.

Understanding the Syntax of Type.FunctionReturn

Before we dive into the specific applications of Type.FunctionReturn, it's important to have a solid understanding of its syntax. At its most basic level, Type.FunctionReturn is a type annotation function that allows you to specify the data type that a function should return.

The syntax for Type.FunctionReturn looks like this:

```
Type.FunctionReturn(returnType as type)
```

In this syntax, `returnType` is a required argument that specifies the data type that the function should return. This data type can be any valid Power Query data type, including text, numbers, dates, lists, tables, and more.

Enhancing Data Analysis with Type.FunctionReturn

One of the most powerful applications of the Type.FunctionReturn function is in enhancing your data analysis workflows. By specifying the data type of the output from a particular function, you can ensure that your data is properly structured and formatted for further analysis and visualization.

For example, let's say that you have a dataset that contains information about sales transactions. You want to calculate the average sales revenue for each salesperson, and then visualize this data in a bar chart.

To do this, you might create a custom function that calculates the average sales revenue for a given salesperson. You can use the Type.FunctionReturn function to specify that this function should return a decimal number, like this:

```
let
  averageSales = (salesperson as text) =>
Type.Is
```

In this article, we will explore the M code behind the Type.Is function and how it can be used in Power Query.

What is the Type.Is Function?

The Type.Is function is a built-in function in Power Query that allows users to check the data type of a column or value in a table. The function takes two arguments: the first argument is the column or value to check, and the second argument is the data type to check for. The syntax of the Type.Is function is as follows:

```
Type.Is(value as any, type as type) as logical
```

The first argument, value, can be any column or value in the table. The second argument, type, is a data type that you want to check for. The function returns a logical value of true or false, depending on whether the value matches the specified data type.

How to Use the Type.Is Function

To use the Type.Is function, follow these steps:

1. Open Power Query and connect to your data source.
2. Select the column or value that you want to check.
3. Click on the Add Column tab in the ribbon.
4. Click on the Custom Column button.
5. In the Custom Column dialog box, enter a name for the new column.
6. In the Custom Column dialog box, enter the following formula:

```
= Type.Is([ColumnName], type)
```

Replace [ColumnName] with the name of the column that you want to check, and replace type with the data type that you want to check for.

Type.IsNullable

Understanding the Type.IsNullable Function

The Type.IsNullable function is used to identify whether a value in a column can be null or not. It takes a parameter that represents the type of the value, and returns a Boolean value indicating whether the value can be null or not. The syntax of the Type.IsNullable function is as follows:

```
Type.IsNullable(type as type) as logical
```

The ‘type’ parameter represents the data type of the value, and the function returns true if the type can be null, and false if the type cannot be null.

The M Code Behind the Type.IsNullable Function

The Type.IsNullable function is implemented using M code, which is a functional programming language used by Power Query. The M code behind the Type.IsNullable function is as follows:

```
(type) =>
  let
    nullableTypes = {
      type nullable any,
      type nullable binary,
      type nullable date,
      type nullable datetime,
      type nullable datetimezone,
      type nullable duration,
      type nullable time,
      type nullable text
Type.IsOpenRecord
```

In this article, we will explore the M code behind the Type.IsOpenRecord function in Power Query and how it can be used to simplify data transformations.

Understanding the Type.IsOpenRecord Function

The Type.IsOpenRecord function is one of the many M functions available in Power Query. It is used to check if a given record is an open record. An open record is a record that can contain additional fields beyond those explicitly defined in its type definition.

The syntax for the Type.IsOpenRecord function is as follows:

```
Type.IsOpenRecord(record as record) as logical
```

The function takes a record as its input and returns a logical value that indicates whether the record is open or not. If the record is an open record, the function will return true. Otherwise, it will return false.

The M Code Behind the Type.IsOpenRecord Function

To understand the M code behind the Type.IsOpenRecord function, let us consider an example. Suppose we have a table named “Sales” with the following data:

Product	Sales
———	——-
A	100
B	200
C	300

We can use the following M code to convert the table to a record:

```
let
  Source = Excel.CurrentWorkbook(){[Name="Sales"]}[Content],
  #"Changed Type" = Table.TransformColumnTypes(Source,{{"Product", type text}, {"Sales", Int64.Type}}),
```

Type.ListItem

Understanding the Type.ListItem Function

The Type.ListItem function is used to extract a single item from a list. It takes two arguments: the list and the index of the item you want to extract. For example, if you have a list of numbers and you want to extract the third item in the list, you would use the following code:

```
List.Select({1, 2, 3, 4, 5}, 2)
```

This would return the value 3, which is the third item in the list. The Type.ListItem function is a shorthand way of writing this code. The equivalent code using the Type.ListItem function would be:

```
{1, 2, 3, 4, 5}{2}
```

This code does the same thing as the previous code, but it is shorter and easier to read. The Type.ListItem function is especially useful when you are working with large lists or when you need to extract multiple items from a list.

The M Code Behind the Type.ListItem Function

The M code behind the Type.ListItem function is relatively simple. The function is defined as follows:

```
Type.ListItem(list as list, index as number) as any =>
   List.PositionOf(list, index, 0)
```

The function takes two arguments: the list and the index of the item you want to extract. It then uses the List.PositionOf function to extract the item from the list. The List.PositionOf function returns the position of the item in the list, and the Type.ListItem function

Type.NonNullable

What is Type.NonNullable?

Type.NonNullable is a function in Power Query that returns the non-nullable version of a given type. In simpler terms, it returns the type of a value with nullability information removed. For example, if you have a column of data that contains null values, you can use Type.NonNullable to create a new column that removes those null values.

The M Code Behind Type.NonNullable

The M code behind Type.NonNullable is fairly simple. Here's what it looks like:

```
(Type as nullable type) as type =>
  if Type.IsNullable(Type) then
    Type.NonNullable(Type)
  else
    Type
```

Let's break this down. The first line defines the function and takes a single parameter, Type, which is a nullable type. The second line checks whether the type is nullable using the Type.IsNullable function. If it is, it uses Type.NonNullable to return the non-nullable version of the type. If it's not nullable, the function simply returns the original type.

Examples of Type.NonNullable in Action

Now that we know what Type.NonNullable does and have seen the M code behind it, let's take a look at some examples of how it can be used.

Example 1: Removing Null Values from a Column

Suppose you have a table with a column named "Salary" that contains both numeric values and null values. If you want to create a new column that removes the null values, you can use Type.NonNullable like this:

```
= Table.AddColumn(Source, "Salary Non-Nullable", each Type.NonNullable(Value.Type([Salary])))
```

Type.OpenRecord

Introduction to Record Types

Before we dive into the M code behind the Type.OpenRecord function, let's first understand what a record type is. A record is a data type that consists of a collection of named fields, where each field has a specified data type. For example, a record type for a person might include fields such as "Name", "Age", and "Address". Each field in the record has a specific data type, such as "text", "number", or "date".

Record types are useful for organizing and structuring data in a consistent and meaningful way. They allow you to group related fields together and enforce a consistent structure for your data. In Power Query, record types are often used to represent rows of data from a table or query.

The Type.OpenRecord Function

The Type.OpenRecord function is used to create a record type from a list of field names and types. The syntax for the function is as follows:

```
Type.OpenRecord([Field1 = Type1, Field2 = Type2, ...])
```

In this syntax, `Field1`, `Field2`, and so on represent the names of the fields in the record, and `Type1`, `Type2`, and so on represent the data types of the fields.

Here's an example of how to use the Type.OpenRecord function to create a record type for a person:

```
Type.OpenRecord([
   Name = Text.Type,
   Age = Int32.Type,
   Address = Text.Type
])
```

Type.RecordFields

What is a Record in Power Query?

In Power Query, a record is a data structure that consists of a set of named values. It is similar to a database record or a dictionary in other programming languages. Each value in a record is associated with a unique name, which is referred to as a field. For example, the following record has two fields: Name and Age.

```
[
  Name = "John",
  Age = 30
]
```

Records are often used to represent data in a structured way, especially when dealing with complex data structures such as JSON or XML documents.

How to Use the Type.RecordFields Function

The Type.RecordFields function is used to extract the fields of a record and return them as a list of field names. The function takes a record as its argument and returns a list of text values.

```
Type.RecordFields(record as record) as list
```

For example, if we have a record that looks like this:

```
[
  Name = "John",
```

Type.ReplaceFacets

What is Type.ReplaceFacets?

Type.ReplaceFacets is a Power Query M function that is used to modify the data type of a column. It is used when you need to change the data type of a column, but also want to apply additional constraints or facets to the new data type. For example, you may want to change a column from a number to a whole number, or from a text string to a date.

Understanding the Syntax

The syntax for Type.ReplaceFacets is as follows:

```
Type.ReplaceFacets(type, facets)
```

The first argument, “type”, specifies the data type you want to change to. This can be any valid data type in Power Query, such as “number”, “text”, or “date”. The second argument, “facets”, is an optional parameter that allows you to specify additional constraints on the new data type. These constraints can include things like minimum or maximum values, precision, or scale.

Examples

To better understand how Type.ReplaceFacets works, let’s explore some examples.

Example 1: Changing a Column to a Whole Number

Suppose you have a column of data that contains decimal values, but you want to convert it to a whole number. You can use Type.ReplaceFacets to accomplish this. Here’s the M code:

```
= Table.TransformColumnTypes(
   #"Previous Step",
   {
      {"Decimal Column", Type.ReplaceFacets(type nullable number, [IsWholeNumber=true])}
   }
)
```

Type.ReplaceTableKeys

Understanding the Basics of Type.ReplaceTableKeys

Before we dive into the M code behind Type.ReplaceTableKeys, let's first take a step back and explore what this function does. Put simply, Type.ReplaceTableKeys is a Power Query M function that allows you to replace the keys in one table with the keys from another table.

This function can be incredibly useful for a variety of data manipulation tasks. For example, let's say you have two tables – one containing customer data, and another containing order data. If you want to merge these tables based on customer ID, you can use Type.ReplaceTableKeys to replace the customer ID key in the order table with the customer ID key in the customer table. This will allow you to merge the two tables based on this common key field.

Taking a Closer Look at the M Code

Now that we understand the basics of Type.ReplaceTableKeys, let's dive into the M code behind this powerful function. To begin, let's take a look at the basic syntax of this function:

```
Type.ReplaceTableKeys(table as table, keyReplacements as list) as table
```

As you can see, this function takes two arguments – the table you want to replace the keys in, and a list of key replacements. Let's take a closer look at each of these arguments.

The Table Argument

The first argument, `table`, is simply the table that you want to replace the keys in. This can be any table, as long as it has one or more columns that can be used as keys.

The keyReplacements Argument

The second argument, `keyReplacements`, is where things start to get a bit more interesting. This argument is a list of key replacements, where each replacement consists of two columns – the original key and the replacement key.

Here's an example of what a keyReplacements list might look like:

```
Type.TableColumn
```

In this article, we will take a closer look at the M code behind the Type.TableColumn function in Power Query.

What is Type.TableColumn?

Type.TableColumn is a M function used to determine the data type of a specific column in a table. It takes in two arguments: a table, and a column name or index.

The function returns a record with the following fields:

– Type: the data type of the column (e.g. text, number, date)
– Culture: the culture used for formatting the column
– DefaultValue: the default value of the column
– IsNullable: whether the column allows null values
– Precision: the precision for numeric columns
– Scale: the scale for numeric columns

How Does Type.TableColumn Work?

To use Type.TableColumn, you first need to create a reference to the table in Power Query. This can be done by selecting the table in the workbook, and clicking on the “From Table/Range” button in the “Data” tab.

Once you have a reference to the table, you can use the Type.TableColumn function to determine the data type of a specific column. Here is an example:

```
let
  Source = Excel.CurrentWorkbook(){[Name="MyTable"]}[Content],
  ColumnType = Type.TableColumn(Source, "MyColumn")
in
  ColumnType
```

In this example, we first create a reference to the “MyTable” table using the Excel.CurrentWorkbook function. We then pass this reference, along with the name of the column we want to check (“MyColumn”), to the Type.TableColumn function.

Type.TableKeys

Syntax
The Type.TableKeys function has the following syntax:

```
Type.TableKeys(table as table) as list
```

The table parameter is the input table for which you want to extract the keys. The function returns a list of records, where each record represents a key of the input table. The record contains two fields: the first is the name of the key, and the second is a list of the columns that make up the key.
Parameters
The Type.TableKeys function has only one parameter, which is the input table. The input table must be a Table object, which is a data type in Power Query that represents a collection of rows and columns. The input table can be obtained from a variety of sources, such as a file, a database, or a web page.
Use Cases
The Type.TableKeys function is commonly used in the following scenarios:
Joining Tables
When you have two or more tables that share one or more keys, you can use the keys to join the tables. To do this, you need to extract the keys of each table using the Type.TableKeys function, and then use the Merge Queries function to join the tables based on the keys. For example, suppose you have two tables: Customers and Orders. Both tables have a column called “Customer ID”, which is the key. To join the tables, you can use the following M code:

```
let
  Source = Table.NestedJoin(
    Customers,
    {"Customer ID"},
Type.TableRow
```

Introduction to Type.TableRow

Type.TableRow is a built-in M function in Power Query that is used to create a table row type. A table row type is a data type that describes the structure of a single row in a table. It is commonly used in Power Query to define the expected data types of columns in a table.

The Type.TableRow function takes a record type as its argument and returns a table row type. A record type is a data type that describes a collection of fields, each with a name and a data type. The table row type returned by Type.TableRow has the same fields as the input record type, but with the addition of an index field.

Here is an example of how to use the Type.TableRow function:

```
let
  Source = #table(
    type table [Name = text, Age = number],
    {
      {"John", 25},
      {"Jane", 30}
    }
  ),
  RowType = type table [Name = text, Age = number, Index = number],
  Transform = Table.TransformColumnTypes(Source, RowType)
in
  Transform
```

In this example, we start with a table called Source that has two columns: Name and Age. We then define a row type called RowType that includes the Name and Age fields, as well as an Index field. Finally, we use the Table.TransformColumnTypes function to convert the Source table to the RowType table.

Type.TableSchema

In this article, we'll take a closer look at the M code behind the Type.TableSchema function, explore its syntax and usage, and provide examples of how it can be used in Power Query.

Understanding the Type.TableSchema Function

The Type.TableSchema function is used to define the schema of a table in Power Query. It is a type function that returns a table schema value, which is used to describe the structure of the table. The syntax of the Type.TableSchema function is as follows:

```
Type.TableSchema(
   columns as list,
   optional primaryKeyColumns as any,
   optional navigationProperties as any,
   optional isNullable as nullable logical,
   optional isOpen as nullable logical
) as table schema
```

Let's break down the parameters of the Type.TableSchema function:

– Columns: A list of columns that define the structure of the table. Each column is defined using the following syntax:

```
   [Name = "column name", Type = type, Optional = optional, DefaultValue = defaultValue]
```

   – Name: The name of the column.
   – Type: The data type of the column.
   – Optional: A logical value that specifies whether the column is optional. If set to true, the column can be null or missing in the table.
   – DefaultValue: The default value of the column if it is missing.

Type.Union

One of the functions in Power Query that is commonly used for data transformation is the Type.Union function. The Type.Union function is used to combine tables with the same schema into a single table. In this article, we will explore the M code behind the Type.Union function and how it can be used to transform data.

Understanding Type.Union

The Type.Union function is used to combine tables with the same schema into a single table. It is a powerful function that can be used to transform and reshape data. The Type.Union function takes two or more tables as input and returns a single table that contains all the columns from the input tables.

The input tables must have the same schema, which means that they must have the same columns with the same data types. If the input tables have different schemas, the Type.Union function will return an error.

Here is the basic syntax for the Type.Union function:

```
Type.Union(table1, table2, ..., tableN)
```

In this syntax, `table1`, `table2`, and `tableN` are the input tables that will be combined into a single table.

The M Code Behind Type.Union

The M code behind the Type.Union function is relatively straightforward. The function simply combines the input tables into a single table by appending the rows of each table to the output table.

Here is the M code behind the Type.Union function:

```
let
  Source = Type.Union(table1, table2, ..., tableN)
in
  Source
```

Uri.BuildQueryString

The `Uri.BuildQueryString` function takes a record as input, and returns a string that represents the query string. The input record can contain any number of fields, and each field represents a parameter that is included in the query string. Here is an example of how the `Uri.BuildQueryString` function can be used:

```
let
  params = [
    param1 = "value1",
    param2 = "value2",
    param3 = "value3"
  ],
  queryString = Uri.BuildQueryString(params)
in
  queryString
```

In this example, a record is created that contains three fields, and the `Uri.BuildQueryString` function is called with this record as input. The resulting string represents the query string `param1=value1&param2=value2&param3=value3`.

Understanding the M Code

To understand how the `Uri.BuildQueryString` function works, it is necessary to look at the M code that powers it. Here is the M code for the `Uri.BuildQueryString` function:

```
let
  BuildQueryString = (params as record) =>
    let
      paramList = Record.ToList(params),
Uri.Combine
```

In this article, we will explore the M code behind the Uri.Combine function, how it works, and some examples of how it can be used in Power Query.

What is the Uri.Combine function?

The Uri.Combine function is a Power Query M function that allows users to combine two or more URIs into a single URI. URIs are used to identify resources on the internet, such as web pages, images, or videos.

The Uri.Combine function takes two or more URIs as arguments and combines them into a single URI. It is similar to the CONCATENATE function in Excel, but for URIs.

How does the Uri.Combine function work?

The Uri.Combine function works by taking two or more URIs as arguments and concatenating them into a single URI. The function also handles any necessary formatting, such as adding a forward slash (/) between URIs if needed.

For example, if we have two URIs, “https://www.example.com” and “/products”, we can use the Uri.Combine function to combine them into a single URI:

```
Uri.Combine("https://www.example.com", "/products")
```

This will return the following URI:

```
"https://www.example.com/products"
```

Examples of using the Uri.Combine function

The Uri.Combine function can be used in a variety of ways in Power Query. Here are some examples of how it can be used:

Combining a base URL with a relative URL

In some cases, we may have a base URL and a relative URL that we want to combine into a single URI. For example, we may have a list of

Uri.EscapeDataString

What is Uri.EscapeDataString?

Before we dive into the M code behind this function, let's first understand what Uri.EscapeDataString is. This function is used to escape special characters in a URL before sending it to a web server. It takes a string as input and returns the escaped string as output. For example, if you have a URL that contains spaces, you can use the Uri.EscapeDataString function to replace the spaces with %20 so that the URL can be properly interpreted by the web server.

The M Code Behind Uri.EscapeDataString

Now that we understand what Uri.EscapeDataString is, let's take a look at the M code behind this function. The M code for Uri.EscapeDataString is relatively simple and consists of just a few lines of code:

```
(text as text) as text =>
 Text.Replace(
  Text.Replace(
   Text.Replace(
    Text.Replace(
     Text.Replace(
      Text.Replace(
       Text.Replace(
        Text.Replace(
         Text.Replace(
          Text.Replace(
           Text.Replace(
            text, " ", "%20"),
           "!", "%21"),
          "'", "%27"),
         "(", "%28"),
        ")", "%29"),
       "", "%2A"),
```

Uri.Parts

Understanding Uniform Resource Identifiers (URIs)

Before we delve into the M code behind Uri.Parts, let's take a moment to understand what a uniform resource identifier (URI) is. Simply put, a URI is a string of characters that identifies a resource on the internet. This resource can be a web page, an image, a video, or any other type of data that can be accessed through a web browser.

A URI consists of several components, including the scheme, authority, path, query, and fragment. The scheme is the protocol used to access the resource (e.g., http, https, ftp). The authority is the domain name or IP address of the server hosting the resource. The path is the location of the resource on the server. The query is any additional parameters passed to the server. The fragment is an optional component that specifies a specific location within the resource.

The Power Query M function Uri.Parts

The Power Query M function Uri.Parts allows users to parse a URI into its various components. The function takes a single parameter, which is the URI to be parsed. The function then returns a record that contains the different components of the URI as separate fields.

The record returned by Uri.Parts contains the following fields:

– Scheme: The protocol used to access the resource (e.g., http, https, ftp)
– Host: The domain name or IP address of the server hosting the resource
– Path: The location of the resource on the server
– Query: Any additional parameters passed to the server
– Fragment: An optional component that specifies a specific location within the resource

The M Code Behind Uri.Parts

The M code behind Uri.Parts is relatively simple. The function is defined as follows:

```
let
   uri = Text.Combine({uri, ""}),
   scheme = Text.BeforeDelimiter(uri, "://"),
   authority = Text.AfterDelimiter(uri, "://"),
   path = Text.AfterDelimiter(authority, "/"),
   query = if Text.Contains(path, "?") then Text.AfterDelimiter(path, "?") else "",
Value.Add
```

At its core, Value.Add is a simple function that takes two arguments: the column or table to which the value will be added, and the value to add. The function then returns a new column or table with the value added. Here is the basic syntax for using Value.Add:

```
Value.Add(table, column, value)
```

Where `table` is the table to which the value will be added, `column` is the name of the column to which the value will be added, and `value` is the value to add.

But what's really going on behind the scenes when you use Value.Add? Let's take a closer look at the M code that powers this function.

The M Code Behind Value.Add

Value.Add is implemented in M code, which is the language used by Power Query to define data transformations. The M code for Value.Add is relatively simple:

```
(table as table, column as text, value as any) as table =>
  let
    addValue = (row) => Record.AddField(row, column, value),
    newTable = Table.TransformRows(table, addValue)
  in
    newTable
```

Let's break this code down into its component parts.

The Function Signature

The first line of the code defines the function signature:

```
Value.Alternates
```

Introduction to the M code behind Value.Alternates

The M code is the backbone of Power Query. It is a functional programming language used to define data transformations. Each function in Power Query has its M code behind it. The Value.Alternates function, for instance, has the following M code:

```
(Value as any, Alternates as list) =>
let
  AlternatingFunction =
    (index as number) =>
      if List.Contains(Alternates, Value) then
        Alternates{index mod List.Count(Alternates)}
      else
        Value,
  Result = AlternatingFunction
in
  Result
```

This code takes two parameters: Value and Alternates. Value is the original value in the column, while Alternates is a list of alternative values. The function uses a nested function called AlternatingFunction to replace the original value with an alternate value.

Practical applications of Value.Alternates

Value.Alternates can be used in various scenarios. Here are some examples:

Example 1: Replacing missing values with an alternate value

In some cases, a dataset may have missing values. For instance, a sales dataset may have missing values in the “Region” column. Value.Alternates can be used to replace the missing values with an alternate value. Consider the following code:

```
Value.As
```

Understanding the Value.As function

The Value.As function in Power Query is used to convert a value to a specific data type. The function syntax is as follows:

```
Value.As(expression as any, type as type)
```

The expression parameter is the value that you want to convert, and the type parameter is the data type that you want to convert the value to.

For example, if you have a text value that you want to convert to a number, you can use the Value.As function as follows:

```
Value.As("123", type number)
```

This will convert the text value “123” to a number.

The M code behind Value.As

The M code behind the Value.As function is fairly simple. Let’s take a look at the M code for the example we used above:

```
Value.As("123", type number)
```

In M code, this would look like:

```
Value.As("123", type number)
```

Value.Compare

One of the essential functions in the M language is the Value.Compare function. This function is used to compare the values of two arguments and return a result based on the comparison. In this article, we will take a closer look at the M code behind the Power Query M function Value.Compare.

What is Value.Compare?

The Value.Compare function is used to compare two values and return a result based on the comparison. The function takes two arguments, value1 and value2, and returns a result that indicates the relationship between the two values. The result can be one of the following:

– Less than (-1)
– Equal to (0)
– Greater than (1)

The syntax of the Value.Compare function is as follows:

```
Value.Compare(value1, value2)
```

Understanding the M Code Behind Value.Compare

The M code behind the Value.Compare function is straightforward and consists of a series of if-else statements. The code compares the two values passed as arguments and returns the appropriate result based on the comparison.

Here is the M code behind the Value.Compare function:

```
let
  Value.Compare = (value1 as any, value2 as any) as number =>
    if value1 < value2 then
      -1
    else if value1 = value2 then
Value.Divide
```

The Value.Divide function is used to divide the values in two columns. It takes two arguments: the numerator and the denominator. For example, if we have a table with two columns – Sales and Cost of Goods Sold – we could use the Value.Divide function to calculate the gross profit margin.

But have you ever wondered what the M code behind the Value.Divide function looks like? In this article, we will take a deep dive into the M code that powers the Value.Divide function.

The Syntax of the Value.Divide Function

Before we dive into the M code, let's first take a look at the syntax of the Value.Divide function.

```
Value.Divide(numerator as any, denominator as any, optional precision as nullable number) as nullable number
```

The Value.Divide function takes three arguments: the numerator, the denominator, and an optional precision. The numerator and denominator can be of any type, while the precision must be a nullable number. The function returns a nullable number.

The M Code Behind the Value.Divide Function

Now that we know the syntax of the function, let's take a look at the M code that powers it.

```
let
  divide = (numerator, denominator, optional precision) =>
    if denominator = 0 or numerator = null or denominator = null then null
    else if precision = null then numerator / denominator
    else Number.Round(numerator / denominator, precision),
  result = Table.AddColumn(Source, NewColumnName, each divide([Numerator], [Denominator], null))
in
  result
```

Value.Equals

What is the Value.Equals Function?

Before diving into the M code behind Value.Equals, let's first discuss what the function does. The Value.Equals function compares the values of two columns or variables in Power Query and returns a Boolean value indicating whether or not they are equal. The function takes two arguments: the first argument is the first column or variable to compare, and the second argument is the second column or variable to compare.

Here is an example of how the Value.Equals function can be used in a Power Query formula:

```
= Table.AddColumn(Source, "Equal", each Value.Equals([Column1], [Column2]))
```

In this example, we are adding a new column to our source table called "Equal". The value in this column will be determined by the result of the Value.Equals function, which compares the values in Column1 and Column2.

The M Code Behind Value.Equals

Now that we understand what the Value.Equals function does, let's take a look at the M code behind it. The M code for Value.Equals looks like this:

```
(Value1 as any, Value2 as any) as logical =>
    if Value1 is Value2 then true else false
```

Let's break down what each line of this code does:

– `(Value1 as any, Value2 as any) as logical =>`: This line of code defines the function and sets the two arguments that the function takes (Value1 and Value2). The `as any` part means that the function can take any data type as an argument. The `as logical` part means that the function will return a Boolean value.

– `if Value1 is Value2 then true else false`: This line of code performs the actual comparison between Value1 and Value2. If they are

Value.Expression

What is the Value.Expression function?

The Value.Expression function in Power Query is a dynamic M function that converts a string value into a dynamic expression. This means that you can use it to dynamically generate M expressions at runtime. The function takes a single argument, which is a string value that contains the M expression you want to evaluate. The Value.Expression function is useful for scenarios where you need to dynamically generate M expressions based on user input or other runtime conditions.

How does the Value.Expression function work?

The Value.Expression function works by taking the string value passed to it and evaluating it as an M expression. It then returns the result of the expression evaluation. The M language is a functional programming language that is used to define Power Query transformations. M expressions are evaluated from left to right, and they can include other M functions, operators, and values. The Value.Expression function allows you to generate M expressions on the fly, which can be a powerful tool in your Power Query arsenal.

How to use the Value.Expression function in Power Query

Using the Value.Expression function in Power Query is straightforward. You can use it in any step of your query where you need to dynamically generate an M expression. To use the function, you simply need to pass a string value containing the M expression you want to evaluate as an argument to the function. Here is an example:

```
let
  Source = Excel.CurrentWorkbook(){[Name="MyTable"]}[Content],
  DynamicColumn = Value.Expression("Table.AddColumn(Source, ""NewColumn"", each [Column1] + [Column2])"),
  #"DynamicColumnAdded" = DynamicColumn
in
  #"DynamicColumnAdded"
```

In this example, we are using the Value.Expression function to dynamically generate a new column in our table. The function takes a string value containing the M expression we want to evaluate as an argument. We then use the Table.AddColumn function to add the new column to our table. The result of the expression evaluation is returned by the function and assigned to the DynamicColumn

Value.Firewall

Understanding Value.Firewall

Before we dive into the M code behind Value.Firewall, let's first understand what this function does. Value.Firewall is used to protect data sources in Power Query. It allows you to specify which data sources are trusted and which ones are not. When a user attempts to access a data source that is not trusted, Power Query will block the request and display an error message.

The purpose of Value.Firewall is to prevent unauthorized access to your data. By default, Power Query allows all data sources to be accessed. This can be a security risk, especially when dealing with sensitive data. Value.Firewall allows you to control which data sources can be accessed and who can access them.

The M code behind Value.Firewall

Now that we understand what Value.Firewall does, let's take a look at the M code behind this function. The Value.Firewall function is actually a combination of two M functions: Value.Native and Security.Firewall.

Value.Native is used to evaluate a value within a specified context. This function is used to evaluate the results of a query step. Security.Firewall, on the other hand, is used to check if a data source is trusted or not.

The Value.Firewall function combines these two functions to evaluate a query step and check if the data source is trusted before returning the result. If the data source is not trusted, an error is returned instead of the actual data.

Here is an example of the M code behind the Value.Firewall function:

```
let
  Source = Excel.Workbook(File.Contents("C:Data.xlsx"), null, true),
  Sheet1_Sheet = Source{[Item="Sheet1",Kind="Sheet"]}[Data],
  #"Promoted Headers" = Table.PromoteHeaders(Sheet1_Sheet, [PromoteAllScalars=true]),
  #"Changed Type" = Table.TransformColumnTypes(#"Promoted Headers",{{"Name", type text}, {"Age", Int64.Type}})
in
  #"Changed Type"
```

This code reads an Excel file, selects the Sheet1 sheet, promotes the header information, and changes the data types. However, this

Value.FromText

Understanding Value.FromText Function

The Value.FromText function is used to convert a text value to a number, date, or time value. It takes one argument, which is the text value that needs to be converted. The function returns the value in the desired data type based on the format of the text value.

The syntax for the Value.FromText function is as follows:

```
Value.FromText(text as text, optional culture as nullable text) as any
```

The `text` parameter is mandatory and represents the text value that needs to be converted. The `culture` parameter is optional and represents the culture-specific information that can be used to convert the text value.

M Code Behind Value.FromText Function

The M code behind the Value.FromText function is as follows:

```
let
  Source = (text as text, optional culture as nullable text) =>
      let
        defaultCulture = "en-US",
        cultureInfo = if culture <> null then Culture.FromName(culture) else Culture.FromName(defaultCulture),
        result = if cultureInfo = null then Number.FromText(text)
            else try Number.FromText(text, cultureInfo) otherwise null
      in
        result
in
  Source
```

Value.Is

What is the Value.Is function?

The Value.Is function is used to check whether a value is of a certain type. This function takes two arguments: the value you want to check and the type you want to check for. It returns a boolean value: True if the value is of the specified type, False if it is not.

For example, the following formula checks whether the value in column “Value” is a number:

```
= Table.AddColumn(#"PreviousStep", "IsNumber", each Value.Is([Value], type number))
```

If the value in the “Value” column is a number, the “IsNumber” column will contain a True value. If the value is not a number, the “IsNumber” column will contain a False value.

How the Value.Is function works

The Value.Is function is actually a wrapper function that calls other functions depending on the type you want to check for. These functions are called type functions.

For example, if you want to check whether a value is a number, the Value.Is function calls the Type.IsNumber function. If you want to check whether a value is a text string, the Value.Is function calls the Type.IsText function.

The M code behind the Value.Is function is actually quite simple. Here is the code for the Value.Is function:

```
(Value as any, Type as type) => try Type.Value(Value) otherwise false
```

Let’s break down this code:

– The function takes two arguments: Value and Type.

– The “as any” syntax means that the Value argument can be of any data type.

– The “as type” syntax means that the Type argument must be a data type.

– The function uses the try otherwise syntax to attempt to call the Type.Value function on the Value argument. If this call succeeds, the

Value.Lineage

What is Value.Lineage?

Value.Lineage is a M function in Power Query that returns a table containing the lineage of a particular value within a column or table. The lineage shows the path that the value took from its original source to its current location. This can be useful when you are working with large datasets and need to trace the origin of a particular value.

How to Use Value.Lineage

To use Value.Lineage, you first need to select the column or table that contains the value you want to trace. Then, you need to call the Value.Lineage function and pass in the value you want to trace as a parameter. The function will then return a table containing the lineage of the value.

Here is an example of how to use Value.Lineage:

```
let
  Source = Excel.CurrentWorkbook(){[Name="Sales"]}[Content],
  ValueToTrace = "500",
  Lineage = Value.Lineage(Source[Amount], ValueToTrace)
in
  Lineage
```

In this example, we are selecting the Sales table from the current workbook and passing in the value “500” as the value to trace. We then call the Value.Lineage function and pass in the Sales[Amount] column as the source column. The function will return a table containing the lineage of the value “500” within the Sales[Amount] column.

The M Code Behind Value.Lineage

Now that we know how to use Value.Lineage, let’s take a closer look at the M code behind the function. The M code for Value.Lineage is as follows:

Value.Metadata

Understanding the Value.Metadata Function

Before we dive into the M code behind the Value.Metadata function, let's first understand what this function does. Simply put, Value.Metadata is a function that returns metadata information about a given value. This value can be a column, a table, or even a cell in a table.

The metadata information returned by this function includes details about the data type of the value, its length, and other properties such as its culture and format settings. This information can be extremely useful when working with large datasets or when trying to understand the structure of a data source.

The M Code Behind the Value.Metadata Function

The M code behind the Value.Metadata function is relatively simple and easy to understand. The function takes a single parameter, which is the value for which metadata information is required. The M code for the Value.Metadata function is as follows:

```
(Value as any) as record =>
  let
    metadata = Value.Metadata(Value)
  in
    metadata
```

As you can see, the function takes the input value and passes it to the Value.Metadata function. The output of the Value.Metadata function is then returned as the result of the Value.Metadata function.

Use Cases for the Value.Metadata Function

Now that we understand the M code behind the Value.Metadata function, let's explore some of its use cases. Here are a few scenarios where the Value.Metadata function can be particularly useful:

1. Understanding the Structure of a Data Source

When working with a large dataset, it can be challenging to understand the structure of the data source. The Value.Metadata function can be used to extract metadata information about the columns and tables in the dataset, helping users to better understand the

Value.Multiply

Understanding the Value.Multiply Function

The Value.Multiply function is a simple but powerful function that allows users to multiply two values together. It takes two arguments: Value1 and Value2. The function then multiplies these two values together and returns the result. The syntax for the function is as follows:

```
Value.Multiply(Value1, Value2)
```

Using the Value.Multiply Function in Power Query

The Value.Multiply function can be used in a variety of ways to manipulate data in Power Query. Here are some examples:

Example 1: Multiplying Two Columns Together

Suppose we have a table with two columns, Price and Quantity, and we want to create a new column that calculates the total cost of each item. We can use the Value.Multiply function to do this. Here's the M code:

```
= Table.AddColumn(#"PreviousStep", "TotalCost", each Value.Multiply([Price], [Quantity]))
```

This code adds a new column called "TotalCost" to the table and calculates the total cost by multiplying the Price and Quantity columns together.

Example 2: Multiplying a Column by a Fixed Value

Suppose we have a table with a column called "Weight" and we want to convert the values in this column from pounds to kilograms. We know that 1 pound is equal to 0.453592 kilograms. We can use the Value.Multiply function to do this. Here's the M code:

```
= Table.TransformColumns(#"PreviousStep",{{"Weight", each Value.Multiply(_, 0.453592)}})
```

Value.NativeQuery

The M code behind the Value.NativeQuery function is what makes it possible to run SQL queries inside Power Query. In this article, we will explore the M code behind this function and how it works.

Understanding the Value.NativeQuery Function

Before diving into the M code behind the Value.NativeQuery function, it is essential to understand what this function does. The Value.NativeQuery function is used to execute native SQL queries against a given data source. This function takes in three parameters:

1. Connection string: This parameter specifies the connection string for the data source that you want to query. The connection string should be in a format that is recognized by the data source.
2. Native query: This parameter specifies the SQL query that you want to execute against the data source.
3. Options record: This parameter is an optional record that specifies additional options for the query. For example, you can specify the query timeout, the command timeout, and the isolation level.

Once you have provided these three parameters, the Value.NativeQuery function will execute the SQL query against the data source and return the results as a table.

The M Code Behind Value.NativeQuery

The M code behind the Value.NativeQuery function is what makes it possible to run SQL queries inside Power Query. This code is essentially a wrapper around the SQL Server Native Client (SNAC) API, which is a library of functions that provides a way to interact with SQL Server databases.

The M code behind the Value.NativeQuery function uses the SNAC API to establish a connection to the data source using the connection string provided. Once the connection is established, the SQL query is executed against the data source using the Execute method provided by the SNAC API.

The results of the query are then returned as a table, which can be further manipulated using the various functions provided by Power Query. The M code behind the Value.NativeQuery function is designed to be flexible and can handle a wide range of SQL queries.

Best Practices for Using Value.NativeQuery

While the Value.NativeQuery function is a powerful tool, it is essential to use it correctly to get the most out of it. Here are some best practices to keep in mind when using this function:

1. Use parameterized queries: When writing SQL queries, it is essential to use parameterized queries instead of concatenating values into the query string. Parameterized queries help prevent SQL injection attacks and improve query performance.
2. Limit the number of rows returned: When executing SQL queries against a large data source, it is easy to return a large number of

Value.NullableEquals

In this article, we will take a deep dive into the M code behind the `Value.NullableEquals` function, understand how it works, and explore some use cases.

Understanding the `Value.NullableEquals` Function

The `Value.NullableEquals` function takes two arguments and returns a boolean indicating whether the values are equal. The function can handle null values, which makes it particularly useful.

The syntax for the function is as follows:

```
Value.NullableEquals(value1 as any, value2 as any) as logical
```

The `value1` and `value2` arguments can be any value. If either of the arguments is null, the function returns false. However, if both arguments are null, the function returns true.

Let's take a look at some examples to understand how the function works.

Example 1: Comparing Two Non-Null Values

Suppose we want to compare two non-null values, 10 and 20. We can use the `Value.NullableEquals` function as follows:

```
Value.NullableEquals(10, 20)
```

The function will return `false` because the two values are not equal.

Example 2: Comparing a Null Value with a Non-Null Value

Suppose we want to compare a null value with a non-null value, say null and 10. We can use the `Value.NullableEquals` function as follows:

```
Value.Optimize
```

Understanding Value.Optimize

Before we dive into the M code behind Value.Optimize, let's first understand what this function does. Value.Optimize is used to optimize the data types of columns in a table. When you load data into Power Query, it tries to automatically detect the data types for each column. However, this detection is not always accurate, and sometimes results in slower query performance.

This is where Value.Optimize comes in. It allows you to optimize the data types of columns in a table to improve query performance. For example, if you have a column with values that are all integers, but Power Query detects it as a decimal column, you can use Value.Optimize to change the data type to integer and improve query performance.

The M Code Behind Value.Optimize

Now that we understand what Value.Optimize does, let's take a look at the M code behind this function. Here is the M code for Value.Optimize:

```
let
  Source = #table( type table [ Column1 = text, Column2 = text ], {} ),
  Optimize = (table as table) as table =>
    let
      types = Table.TransformColumnTypes(table, List.Transform(Table.ColumnNames(table), each {_, type any})),
      optimizedTypes = Table.TransformColumns(types, List.Transform(Table.ColumnNames(types), each {_,
Value.Type(Text.Combine(List.Sort(List.Distinct(Table.Column(table, _))))})))
    in
      Table.TransformColumnTypes(table, List.Transform(Table.ColumnNames(optimizedTypes), each {_,
optimizedTypes{_}[Column1]}))
    in
  Optimize
```

As you can see, the M code for Value.Optimize is quite complex. However, let's break it down step-by-step to help you understand what

Value.RemoveMetadata

In this article, we will explore the M code behind the Value.RemoveMetadata function, how it works, and how it can be used to improve data processing in Power Query.

Understanding Metadata in Power Query

Before we dive into the M code behind the Value.RemoveMetadata function, it's important to understand what metadata is and how it affects data processing in Power Query.

Metadata refers to information that describes the structure, content, or context of data. In Power Query, metadata is often generated automatically when importing data from different sources, such as Excel spreadsheets, CSV files, or databases. This metadata can include column names, data types, formatting, and other attributes that are used to define the structure of the data table.

While metadata can be useful for understanding and manipulating data, it can also add unnecessary overhead and complexity to data processing in Power Query. For example, metadata can slow down queries, increase file size, and make it harder to perform certain operations on the data.

To address these issues, Power Query provides the Value.RemoveMetadata function, which allows you to selectively remove metadata from data tables, while preserving the underlying data values.

The Syntax of Value.RemoveMetadata

The Value.RemoveMetadata function is a simple yet powerful M function that takes a data table as its input, and returns a new data table with the same data values, but without any metadata.

The syntax of Value.RemoveMetadata is as follows:

```
Value.RemoveMetadata(data as any) as table
```

Here, "data" is the input data table, which can be of any type, such as table, list, or record. The function returns a new table with the same data values, but without any metadata.

For example, let's say we have a data table with the following metadata:

Value.ReplaceMetadata

What is Power Query M?

Power Query M is a functional programming language used in Power Query, a data transformation and data preparation tool in Microsoft Excel and Power BI. It is used to connect to various data sources, transform and shape data, and load it into an output destination. Power Query M is a powerful tool in data analytics, as it allows users to perform complex data transformations with ease.

Understanding Value.ReplaceMetadata

In Power Query M, metadata is data that describes other data. It provides information about the structure and content of data, such as column headers and data types. The function Value.ReplaceMetadata is used to replace the metadata of a value in M.

The syntax for Value.ReplaceMetadata is as follows:

```
Value.ReplaceMetadata(value as any, metadata as any) as any
```

The first parameter, `value`, is the value whose metadata needs to be replaced. The second parameter, `metadata`, is the new metadata that needs to be applied to the value. The function returns a new value with the updated metadata.

The M Code Behind Value.ReplaceMetadata

The M code behind Value.ReplaceMetadata is relatively simple. The function takes two parameters, `value` and `metadata`, and returns a new value with the updated metadata. Here is the M code for Value.ReplaceMetadata:

```
(Value as any, Metadata as any) =>
  let
    OriginalType = Value.Type,
    OriginalNullable = Value.IsNullable,
    OriginalKind = Value.Kind,
    OriginalMetadata = Value.Metadata,
    ReplaceType = if (Type.Is(Value.Type, type function)) then Value.Type else Metadata.Type,
Value.ReplaceType
```

Understanding the Value.ReplaceType Function

Before we dive into the M code behind Value.ReplaceType, let's briefly review how the function works. The syntax for Value.ReplaceType is as follows:

```
Value.ReplaceType(value as any, type as type) as any
```

The function takes two arguments: the value to be converted and the new data type. For example, if you wanted to convert a column containing text values to a numeric data type, you could use the following code:

```
Value.ReplaceType(#"PreviousStep", type number)
```

This would replace the data type of the column from text to numeric.

Breaking Down the M Code

Now that we understand how Value.ReplaceType works, let's take a closer look at the M code behind it. The code for Value.ReplaceType can be found in the following file:

```
C:Program FilesMicrosoft Power BI DesktopCustom ConnectorsMicrosoftPowerQueryFunctionsValue.m
```

The code for the function is as follows:

```
Value.Subtract
```

Understanding Value.Subtract

Before we dive into the code, let's first understand what the Value.Subtract function does. The function takes two values as inputs and subtracts the second value from the first. For example, if we have two values A and B, the function will return A-B.

Here's an example of how the function is used in Power Query:

```
= Value.Subtract(10,5)
```

In this example, the function will return 5, as 5 is subtracted from 10.

The M Code Behind Value.Subtract

Now let's take a look at the M code behind this function. The Value.Subtract function is defined as follows:

```
(Value1 as any, Value2 as any) as any =>
   Value1 - Value2
```

Let's break this code down and understand what it does.

The first line defines the function and its input parameters. In this case, the function takes two input parameters, Value1 and Value2, which can be any data type.

The second line uses the "=>" operator to define what the function does. In this case, the function subtracts Value2 from Value1 and returns the result.

It's worth noting that the Value.Subtract function is not limited to just numbers. It can be used to subtract any two values, regardless of their data type.

Using Value.Subtract in Power Query

Now that we understand the M code behind the Value.Subtract function, let's take a look at how it can be used in Power Query.

Value.Traits

What is Value.Traits Function?

The Value.Traits function is a part of the Power Query M language and is used to extract information about a column or a value from a table. It returns a record that contains various properties of the value such as its data type, precision, scale, and others. The function takes a single argument, which can be any value or column in a table.

Syntax

The syntax for the Value.Traits function is as follows:

```
Value.Traits(value as any) as record
```

The argument `value` can be any value or column in a table. The function returns a record that contains various properties of the value.

Properties of Value.Traits Function

The record returned by the Value.Traits function contains the following properties:

– `Kind`: The data type of the value, such as text, number, date, time, duration, or datetime.
– `Length`: The size of the value in bytes. This property is only applicable to text and binary values.
– `Precision`: The maximum number of digits that the value can hold. This property is only applicable to numeric values.
– `Scale`: The number of digits to the right of the decimal point. This property is only applicable to numeric values.
– `Nullable`: A boolean value that indicates whether the value can be null or not.
– `Values`: A list of distinct values in the column. This property is only applicable to columns.

How to Use Value.Traits Function?

The Value.Traits function can be used in various scenarios. Here are some examples:

Example 1: Check Data Type of a Column

Suppose you have a table that contains a column named “Age” and you want to check its data type. Here is how you can use the Value.Traits function:

```
Value.Type
```

Basic Syntax
The basic syntax of the Value.Type function is as follows:

```
Value.Type(expression as any) as type
```

Here, the `expression` parameter is the value for which we want to determine the data type, and the `type` parameter is the name of the data type that the expression represents.
For example, to determine the data type of a column named “StartDate” in a table named “SalesData”, we would use the following M code:

```
Value.Type(SalesData[StartDate])
```

This would return the name of the data type of the values in the “StartDate” column, such as “date” or “text”.
Supported Data Types
The Value.Type function supports a wide range of data types, including:
– `logical`: Boolean true/false values.
– `number`: Numeric values, including integers, decimals, and fractions.
– `date`: Dates and times.
– `duration`: Time durations.
– `text`: Text strings.
– `binary`: Binary data.
In addition, the Value.Type function can also return the following special data types:
– `list`: A list of values.

Value.VersionIdentity

What is the M language?

M is the programming language used by Power Query to perform data transformations. It is a functional language that is similar to other programming languages like F# and Haskell. M is used to define queries, functions, and expressions that are used to transform data.

The Value.VersionIdentity function

The Value.VersionIdentity function returns a record that contains information about the current version of the Power Query engine that is running. The record contains the following fields:

– BuildDate: The date that the engine was built.
– BuildNumber: The build number of the engine.
– FullVersion: The full version number of the engine.
– MajorVersion: The major version number of the engine.
– MinorVersion: The minor version number of the engine.
– Revision: The revision number of the engine.
– Version: The version number of the engine.

Here is an example of how to use the Value.VersionIdentity function in Power Query:

```
let
   version = Value.VersionIdentity(),
   buildDate = version[BuildDate],
   buildNumber = version[BuildNumber],
   fullVersion = version[FullVersion],
   majorVersion = version[MajorVersion],
   minorVersion = version[MinorVersion],
   revision = version[Revision],
   versionNumber = version[Version]
in
   versionNumber
```

Value.Versions

What is the Value.Versions Function?

The Value.Versions function is used to access all the values that a specific column in a table has had over time. For example, if you have a table with a column that contains the prices of a stock over a period of time, you can use the Value.Versions function to access all the previous prices of the stock.

The syntax for the Value.Versions function is as follows:

```
Value.Versions(table as table, column as text) as list
```

The table parameter is the table that you want to access, and the column parameter is the name of the column that you want to access the values from.

The M Code Behind Value.Versions

The M code behind the Value.Versions function is relatively simple. It starts by filtering the table to only include the rows where the value in the specified column is not null. It then groups the resulting table by the value in the specified column and creates a new column that contains a list of all the values in the specified column for each group.

Here is the M code behind the Value.Versions function:

```
(table as table, column as text) =>
let
   Source = Table.SelectRows(table, each [column] <> null),
   Grouped = Table.Group(Source, {column}, {{"Versions", each [column], type list}})
in
   Grouped[Versions]
```

Value.ViewError

Understanding the Value.ViewError Function

The Value.ViewError function is used to display detailed error messages when a query encounters an error. This function is particularly useful when you are working with complex queries that contain multiple transformations. When an error occurs, the Value.ViewError function is called, and a detailed error message is displayed.

The syntax for the Value.ViewError function is as follows:

```
Value.ViewError(error as any, optional message as nullable text)
```

The error parameter is mandatory and represents the error that occurred. The message parameter is optional and represents a custom message that can be displayed along with the error message.

Analyzing the M Code Behind Value.ViewError

Behind the scenes, the Value.ViewError function uses a combination of M functions to display the error message. Let's take a closer look at the M code behind the Value.ViewError function.

The Error Function

The Error function is a built-in M function that is used to generate an error. The syntax for the Error function is as follows:

```
Error(message as text, optional value as any) as any
```

The message parameter is mandatory and represents the error message. The value parameter is optional and represents a custom value that can be associated with the error.

The Error function is used in combination with the Try...Otherwise statement to detect and handle errors in Power Query.

The Try...Otherwise Statement

The Try...Otherwise statement is used to detect and handle errors in Power Query. The syntax for the Try...Otherwise statement is as

```
Value.ViewFunction
```

Value.ViewFunction is a versatile function that allows you to view and modify the underlying M code behind other Power Query functions. This can be incredibly useful when troubleshooting issues with a particular function or simply gaining a better understanding of how Power Query works.

In this article, we will dive into the M code behind Value.ViewFunction and explore how it can be used to gain deeper insights into Power Query.

What is the M Code?

Before we dive into the M code behind Value.ViewFunction, it is essential to understand what the M code is and how it works.

M code is the programming language used by Power Query to transform and cleanse data. It is a functional programming language that allows you to define functions and apply them to data sets. Power Query automatically generates M code when you use its graphical user interface to perform transformations.

However, M code is not limited to Power Query. You can generate M code in other applications, such as Power BI, Excel, and Visual Studio.

Understanding Value.ViewFunction

Value.ViewFunction is a function that allows you to view and modify the underlying M code of other Power Query functions. It takes one argument, which is the function you want to view. The function you want to view must be surrounded by quotes.

The syntax for Value.ViewFunction is as follows:

```
Value.ViewFunction("function_name")
```

For example, if you want to view the underlying M code for the Table.SelectColumns function, you would use the following syntax:

```
Value.ViewFunction("Table.SelectColumns")
```

Variable.Value

What is Variable.Value?

Variable.Value is an M function that returns the value of the specified variable. It is often used to store intermediate values that are needed for further calculations. Using Variable.Value helps to avoid duplicating computations and improve the performance of your queries.

For example, suppose you have a table that contains sales data for different products. You want to calculate the total sales for each product. Instead of calculating the total for each product separately, you can use Variable.Value to store the intermediate result and use it to calculate the grand total.

Syntax of Variable.Value

The syntax of Variable.Value is straightforward. Here is how it looks like:

```
Variable.Value(variableName)
```

where variableName is the name of the variable whose value you want to retrieve.

How to use Variable.Value in Power Query

To use Variable.Value in Power Query, you need to create a new variable and assign a value to it. You can then use Variable.Value to retrieve the value of the variable and use it in further calculations.

Here is an example of how to use Variable.Value in Power Query to calculate the total sales for each product:

1. Open Power Query and load the sales data into a new query.
2. Create a new variable by clicking on the ‘New Source’ button on the ‘Home’ tab and selecting ‘Blank Query’.
3. In the ‘Query Editor’, click on ‘View’ and select ‘Advanced Editor’.
4. In the ‘Advanced Editor’, enter the following M code:

```
let
  Source = SalesData,
Web.BrowserContents
```

What is the Web.BrowserContents function?

The Web.BrowserContents function is a built-in M function in Power Query that allows users to retrieve the HTML contents of a webpage. It is one of several functions available in Power Query for web scraping, which is the process of extracting data from websites. The function takes a URL as input and returns the HTML contents of the webpage as a text string.

How does the Web.BrowserContents function work?

The Web.BrowserContents function uses the Internet Explorer browser engine to retrieve the contents of a webpage. It works by creating an instance of the InternetExplorer.Application COM object, navigating to the specified URL, and then retrieving the HTML contents of the page using the Document.Body.outerHTML property. The function also supports authentication and cookie handling, allowing users to retrieve contents from pages that require authentication or have session cookies.

Using the Web.BrowserContents function

To use the Web.BrowserContents function in Power Query, you need to create a new query and enter the URL of the webpage you want to retrieve data from. The following example shows how to use the function to retrieve the title and content of a Wikipedia page:

```
let
  Source = Web.BrowserContents("https://en.wikipedia.org/wiki/Power_Query"),
  #"Converted to Table" = Html.Table(Source, {{"Title", "title"}, {"Content", "p"}}, {"Title", "Content"})
in
  #"Converted to Table"
```

In this example, we first create a new query and use the Web.BrowserContents function to retrieve the HTML contents of the Wikipedia page for Power Query. We then use the Html.Table function to parse the contents into a table format, with columns for the page title and content.

Limitations of the Web.BrowserContents function

While the Web.BrowserContents function is a useful tool for web scraping, it has a number of limitations. One of the main limitations is that it relies on the Internet Explorer browser engine, which is not the most modern or efficient browser engine. This can result in slow

Web.Contents

In this article, we will take a closer look at the M code behind the Web.Contents function and explore some of the ways in which it can be used to extract data from websites.

Understanding the Web.Contents Function

The Web.Contents function is used to retrieve data from a website using a URL. The function takes two arguments – the URL of the website to retrieve data from and an optional set of options that can be used to customize the request.

The options argument is an optional record that can be used to specify a variety of settings for the request. For example, the options record can be used to specify custom headers, cookies, and user agent strings. It can also be used to specify the encoding to use for the response data.

When the Web.Contents function is called, it sends a request to the specified URL and waits for a response. Once a response is received, the function returns a binary value that represents the response data.

Using Web.Contents to Extract Data

One of the most common use cases for the Web.Contents function is to extract data from websites. To do this, users can use the Web.Contents function in conjunction with other Power Query M functions, such as Xml.Document or Json.Document, to parse the response data.

For example, let's say we want to extract data from the following website:

https://jsonplaceholder.typicode.com/users/1

To do this, we can use the following M code:

```
let
  url = "https://jsonplaceholder.typicode.com/users/1",
  options = [Headers=[#"Content-Type"="application/json"]],
  source = Web.Contents(url, options),
```

Web.Headers

One of the most widely used functions in Power Query is the Web.Headers function. This function is used to retrieve the headers of a web page, which can be useful when working with web data sources. In this article, we will take a closer look at the M code behind the Web.Headers function, and explore its various use cases.

Understanding the Web.Headers function

The Web.Headers function is a built-in function in Power Query that is used to retrieve the headers of a web page. Headers are pieces of information that are sent between a client and a server as part of an HTTP request and response. These headers contain metadata about the data being transferred, such as the content type, encoding, and language.

The Web.Headers function takes a single argument, which is the URL of the web page whose headers you want to retrieve. Here is an example of how to use the Web.Headers function in Power Query:

```
let
  Source = Web.Headers("https://www.example.com"),
  Headers = Record.FieldNames(Source{0})
in
  Headers
```

In this example, we first use the Web.Headers function to retrieve the headers of the web page at the URL https://www.example.com. We then extract the field names from the first record in the resulting table, which gives us a list of the headers.

Use cases for the Web.Headers function

The Web.Headers function can be used for a variety of purposes in Power Query. Here are some common use cases:

1. Retrieving metadata about a web page

The Web.Headers function can be used to retrieve metadata about a web page, such as its content type, encoding, and language. This information can be useful when working with web data sources, as it can help you understand the structure of the data and how to work with it in Power Query.

2. Checking the status of a web page

Web.Page

In order to fully understand and utilize the Web.Page function, it is important to have a working knowledge of the M language, which is the language used by Power Query to perform data transformations. In this article, we will dive into the M code behind the Web.Page function and explore its various components and functionalities.

Understanding the Web.Page Function

The Web.Page function is a built-in function within Power Query that allows users to extract data from web pages. It is a powerful tool that can be used to scrape large amounts of data from a website and transform it into a usable format.

To use the Web.Page function, users must first specify the URL of the web page they wish to extract data from. They can then specify which elements of the page they want to extract, such as tables, lists, or specific HTML tags.

The Web.Page function returns a table containing the extracted data, which can then be further transformed and analyzed within Power Query.

Breaking Down the M Code

The M language is used to write the code that instructs Power Query on how to extract and transform data. Let's break down the M code behind the Web.Page function to gain a better understanding of how it works.

First, we need to specify the URL of the web page we want to extract data from. This is done using the following code:

```
let
  Source = Web.Page(Web.Contents("https://www.example.com"))
in
  Source
```

In this code, we use the Web.Contents function to retrieve the contents of the web page we want to extract data from. We then pass this content to the Web.Page function, which extracts the desired elements and returns a table.

Next, we can specify which elements we want to extract from the web page. This is done using the following code:

```
WebAction.Request
```

In this article, we will explore the M code behind the WebAction.Request function and how it can be used to retrieve data from the web.

Understanding WebAction.Request

The WebAction.Request function is used to make HTTP requests to web pages, web services, and APIs. The function takes several parameters, including the URL of the web page or service, the HTTP method to use (GET, POST, PUT, DELETE), and any headers or query parameters to include in the request.

Here is an example of the WebAction.Request function in action:

```
let
  url = "https://www.example.com/api/data",
  options = [
    Headers = [
      #"Content-Type"="application/json"
    ],
    Query = [
      limit = 100
    ]
  ],
  result = WebAction.Request(url, options)
in
  result
```

In this example, we are making a GET request to the URL “https://www.example.com/api/data” with a query parameter of “limit=100”. We are also including a header with the content type of “application/json”. The result of the request is stored in the variable “result”.

Understanding the M Code

Behind the scenes, the WebAction.Request function generates M code that handles the HTTP request and response. The M code is a Xml.Document

The Xml.Document function in Power Query is a powerful tool that can be used to extract data from XML documents. It takes an XML document as input and returns a table that represents the data in the XML document. In this article, we will take a closer look at the M code behind the Xml.Document function and how it can be used to extract data from XML documents.

Understanding the Xml.Document Function

The Xml.Document function in Power Query takes an XML document as input and returns a table that represents the data in the XML document. The input to the function can be in the form of a file path, a URL or a text string. The function parses the XML document and returns a table with columns that correspond to the elements in the XML document.

The table returned by the Xml.Document function can be further transformed using other Power Query functions such as Table.SelectColumns, Table.RenameColumns, and Table.TransformColumns. This makes it possible to extract only the required data from the XML document and transform it into a format that is suitable for analysis.

The M Code Behind Xml.Document

The M code behind the Xml.Document function is a series of steps that are executed by Power Query to extract data from the XML document. The M code is generated automatically by Power Query when the Xml.Document function is used.

Here is an example of the M code that is generated by Power Query when the Xml.Document function is used:

```
let
  Source = Xml.Document(File.Contents("C:UsersJohnDocumentsexample.xml")),
  #"Converted to Table" = Record.ToTable(Source)
in
  #"Converted to Table"
```

The M code starts with the ‘let’ keyword which is used to define a variable called ‘Source’. The ‘Source’ variable contains the result of calling the Xml.Document function with the file path to the XML document as input.

The next line of the M code uses the ‘Record.ToTable’ function to convert the ‘Source’ variable into a table. This table contains columns that correspond to the elements in the XML document.

Xml.Tables

In this article, we will explore the M code behind the Xml.Tables function and how it can be used to extract and transform data.

What is XML?

XML stands for eXtensible Markup Language. It is a markup language that is used to store and transport data. XML files are structured in a hierarchical format and can be quite complex. Each element in an XML file has a name and contains data or other elements.

Understanding the Xml.Tables Function

The Xml.Tables function is a built-in function in Power Query that allows you to extract data from an XML file and convert it into a table format. This function takes a single argument, which is the URL or file path of the XML file.

Let's take a look at an example of how the Xml.Tables function works:

```
let
  Source = Xml.Tables("https://www.example.com/data.xml"),
  #"Expanded Table" = Table.ExpandTableColumn(Source, "Table", {"Column1", "Column2", "Column3"}, {"Column1", "Column2", "Column3"})
in
  #"Expanded Table"
```

In this example, we are using the Xml.Tables function to extract data from an XML file located at https://www.example.com/data.xml. The resulting table will contain three columns, named Column1, Column2, and Column3.

The M Code Behind Xml.Tables

The M code behind the Xml.Tables function is what makes it possible to extract and transform data from an XML file. Let's take a look at the M code for the example we just used:

```
let
  Source = Xml.Tables("https://www.example.com/data.xml"),
```