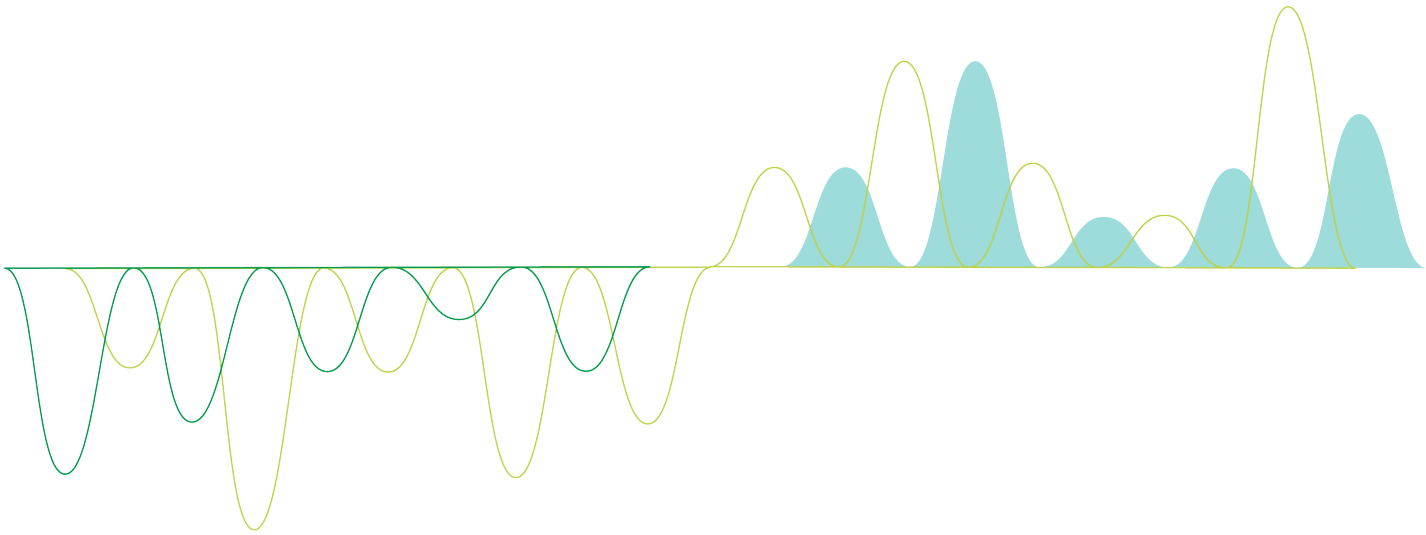
Script syntax and chart functions

Qlik Sense®
August 2022



Contents

1 What is Qlik Sense?

Qlik Sense is a platform for data analysis. With Qlik Sense you can analyze data and make data discoveries on your own. You can share knowledge and analyze data in groups and across organizations. Qlik Sense lets you ask and answer your own questions and follow your own paths to insight. Qlik Sense enables you and your colleagues to reach decisions collaboratively.

1.1 What can you do in Qlik Sense?

Most Business Intelligence (BI) products can help you answer questions that are understood in advance. But what about your follow-up questions? The ones that come after someone reads your report or sees your visualization? With the Qlik Sense associative experience, you can answer question after question after question, moving along your own path to insight. With Qlik Sense you can explore your data freely, with just clicks, learning at each step along the way and coming up with next steps based on earlier findings.

1.2 How does Qlik Sense work?

Qlik Sense generates views of information on the fly for you. Qlik Sense does not require predefined and static reports or you being dependent on other users – you just click and learn. Every time you click, Qlik Sense instantly responds, updating every Qlik Sense visualization and view in the app with a newly calculated set of data and visualizations specific to your selections.

The app model

Instead of deploying and managing huge business applications, you can create your own Qlik Sense apps that you can reuse, modify and share with others. The app model helps you ask and answer the next question on your own, without having to go back to an expert for a new report or visualization.

The associative experience

Qlik Sense automatically manages all the relationships in the data and presents information to you using a **green/white/gray** metaphor. Selections are highlighted in green, associated data is represented in white, and excluded (unassociated) data appears in gray. This instant feedback enables you to think of new questions and continue to explore and discover.

Collaboration and mobility

Qlik Sense further enables you to collaborate with colleagues no matter when and where they are located. All Qlik Sense capabilities, including the associative experience and collaboration, are available on mobile devices. With Qlik Sense, you can ask and answer your questions and follow-up questions, with your colleagues, wherever you are.

1.3 How can you deploy Qlik Sense?

There are two versions of Qlik Sense to deploy, Qlik Sense Desktop and Qlik Sense Enterprise.

Qlik Sense Desktop

This is an easy-to-install single user version that is typically installed on a local computer.

Qlik Sense Enterprise

This version is used to deploy Qlik Sense sites. A site is a collection of one or more server machines connected to a common logical repository or central node.

1.4 How to administer and manage a Qlik Sense site

With the Qlik Management Console you can configure, manage and monitor Qlik Sense sites in an easy and intuitive way. You can manage licenses, access and security rules, configure nodes and data source connections and synchronize content and users among many other activities and resources.

1.5 Extend Qlik Sense and adapt it for your own purposes

Qlik Sense provides you with flexible APIs and SDKs to develop your own extensions and adapt and integrate Qlik Sense for different purposes, such as:

Building extensions and mashups

Here you can do web development using JavaScript to build extensions that are custom visualization in Qlik Sense apps, or you use a mashups APIs to build websites with Qlik Sense content.

Building clients

You can build clients in .NET and embed Qlik Sense objects in your own applications. You can also build native clients in any programming language that can handle WebSocket communication by using the Qlik Sense client protocol.

Building server tools

With service and user directory APIs you can build your own tool to administer and manage Qlik Sense sites.

Connecting to other data sources

Create Qlik Sense connectors to retrieve data from custom data sources.

2 Script syntax overview

2.1 Introduction to script syntax

In a script, the name of the data source, the names of the tables, and the names of the fields included in the logic are defined. Furthermore, the fields in the access rights definition are defined in the script. A script consists of a number of statements that are executed consecutively.

The Qlik Sense command line syntax and script syntax are described in a notation called Backus-Naur Formalism, or BNF code.

The first lines of code are already generated when a new Qlik Sense file is created. The default values of these number interpretation variables are derived from the regional settings of the OS.

The script consists of a number of script statements and keywords that are executed consecutively. All script statements must end with a semicolon, "**;**".

You can use expressions and functions in the **LOAD**-statements to transform the data that has been loaded.

For a table file with commas, tabs or semicolons as delimiters, a **LOAD**-statement may be used. By default a **LOAD**-statement will load all fields of the file.

General databases can be accessed through ODBC or OLE DBdatabase connectors. . Here standard SQL statements are used. The SQL syntax accepted differs between different ODBC drivers.

Additionally, you can access other data sources using custom connectors.

2.2 What is Backus-Naur formalism?

The Qlik Sense command line syntax and script syntax are described in a notation called Backus-Naur formalism, also known as BNF code.

The following table provides a list of symbols used in BNF code, with a description of how they are interpreted:

Symbols

Symbol	Description
|	Logical OR: the symbol on either side can be used.
( )	Parentheses defining precedence: used for structuring the BNF syntax.
[ ]	Square brackets: enclosed items are optional.
{ }	Braces: enclosed items may be repeated zero or more times.
Symbol	A non-terminal syntactic category, that: can be divided further into other symbols. For example, compounds of the above, other non-terminal symbols, text strings, and so on.
::=	Marks the beginning of a block that defines a symbol.
LOAD	A terminal symbol consisting of a text string. Should be written as it is into the script.

All terminal symbols are printed in a **bold face** font. For example, "(" should be interpreted as a parenthesis defining precedence, whereas "**(**" should be interpreted as a character to be printed in the script.

Example:

The description of the alias statement is:

alias fieldname **as** aliasname { , fieldname **as** aliasname}

This should be interpreted as the text string "alias", followed by an arbitrary field name, followed by the text string "as", followed by an arbitrary alias name. Any number of additional combinations of "fieldname as alias" may be given, separated by commas.

The following statements are correct:

```
alias a as first;
alias a as first, b as second;
alias a as first, b as second, c as third;
```

The following statements are not correct:

```
alias a as first b as second;
alias a as first { , b as second };
```

2 Script statements and keywords

The Qlik Sense script consists of a number of statements. A statement can be either a regular script statement or a script control statement. Certain statements can be preceded by prefixes.

Regular statements are typically used for manipulating data in one way or another. These statements may be written over any number of lines in the script and must always be terminated by a semicolon, ";".

Control statements are typically used for controlling the flow of the script execution. Each clause of a control statement must be kept inside one script line and may be terminated by a semicolon or the end-of-line.

Prefixes may be applied to applicable regular statements but never to control statements. The **when** and **unless** prefixes can however be used as suffixes to a few specific control statement clauses.

In the next subchapter, an alphabetical listing of all script statements, control statements and prefixes, are found.

All script keywords can be typed with any combination of lower case and upper case characters. Field and variable names used in the statements are however case sensitive.

2.3 Script control statements

The Qlik Sense script consists of a number of statements. A statement can be either a regular script statement or a script control statement.

Control statements are typically used for controlling the flow of the script execution. Each clause of a control statement must be kept inside one script line and may be terminated by semicolon or end-of-line.

Prefixes are never applied to control statements, with the exceptions of the prefixes **when** and **unless** which may be used with a few specific control statements.

All script keywords can be typed with any combination of lower case and upper case characters.

Script control statements overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Call

The **call** control statement calls a subroutine which must be defined by a previous **sub** statement.

```
Call name ( [ paramlist ])
```

Do..loop

The **do..loop** control statement is a script iteration construct which executes one or several statements until a logical condition is met.

```
Do..loop [ ( while | until ) condition ] [statements]
[exit do [ ( when | unless ) condition ] [statements]
loop [ ( while | until ) condition ]
```

Exit script

This control statement stops script execution. It may be inserted anywhere in the script.

```
Exit script[ (when | unless) condition ]
```

For each ..next

The **for each..next** control statement is a script iteration construct which executes one or several statements for each value in a comma separated list. The statements inside the loop enclosed by **for** and **next** will be executed for each value of the list.

```
For each..next var in list
[statements]
[exit for [ ( when | unless ) condition ]
[statements]
next [var]
```

For..next

The **for..next** control statement is a script iteration construct with a counter. The statements inside the loop enclosed by **for** and **next** will be executed for each value of the counter variable between specified low and high limits.

```
For..next counter = expr1 to expr2 [ stepexpr3 ]
[statements]
[exit for [ ( when | unless ) condition ]
[statements]
Next [counter]
```

If..then

The **if..then** control statement is a script selection construct forcing the script execution to follow different paths depending on one or several logical conditions.

> *Since the **if..then** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its four possible clauses (**if..then**, **elseif..then**, **else** and **end if**) must not cross a line boundary.*

```
If..then..elseif..else..end if condition then
  [ statements ]
{ elseif condition then
  [ statements ] }
[ else
  [ statements ] ]
end if
```

Sub

The **sub..end sub** control statement defines a subroutine which can be called upon from a **call** statement.

```
Sub..end sub name [ ( paramlist )] statements end sub
```

Switch

The **switch** control statement is a script selection construct forcing the script execution to follow different paths, depending on the value of an expression.

```
Switch..case..default..end switch expression {case valuelist [ statements ]}
[default statements] end switch
```

Call

The **call** control statement calls a subroutine which must be defined by a previous **sub** statement.

Syntax:

```
Call name ( [ paramlist ])
```

Arguments:

Arguments

Argument	Description
name	The name of the subroutine.
paramlist	A comma separated list of the actual parameters to be sent to the subroutine. Each item in the list may be a field name, a variable or an arbitrary expression.

The subroutine called by a **call** statement must be defined by a **sub** encountered earlier during script execution.

Parameters are copied into the subroutine and, if the parameter in the **call** statement is a variable and not an expression, copied back out again upon exiting the subroutine.

Limitations:

- Since the **call** statement is a control statement and as such is ended with either a semicolon or end-of-line, it must not cross a line boundary.
- When you define a subroutine with `Sub..end sub` inside a control statement, for example `if..then`, you can only call the subroutine from within the same control statement.

Example:

This example lists all Qlik related files in a folder and its subfolders, and stores file information in a table. It is assumed that you have created a data connection named Apps to the folder .

The DoDir subroutine is called with the reference to the folder, 'lib://Apps', as parameter. Inside the subroutine, there is a recursive call, `Call DoDir (Dir)`, that makes the function look for files recursively in subfolders.

```
sub DoDir (Root)
   For Each Ext in 'qvw', 'qvo', 'qvs', 'qvt', 'qvd', 'qvc', 'qvf'
      For Each File in filelist (Root&'\*.' &Ext)
          LOAD
             '$(File)' as Name,
             FileSize( '$(File)' ) as Size,
             FileTime( '$(File)' ) as FileTime
          autogenerate 1;
      Next File
   Next Ext
   For Each Dir in dirlist (Root&'\*' )
      Call DoDir (Dir)
   Next Dir
End Sub

Call DoDir ('lib://Apps')
```

Do..loop

The **do..loop** control statement is a script iteration construct which executes one or several statements until a logical condition is met.

Syntax:

```
Do [ ( while | until ) condition ] [statements]
[exit do [ ( when | unless ) condition ] [statements]
loop[ ( while | until ) condition ]
```

*Since the **do..loop** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its three possible clauses (**do**, **exit do** and **loop**) must not cross a line boundary.*

Arguments:

Arguments

Argument	Description
condition	A logical expression evaluating to True or False.
statements	Any group of one or more Qlik Sense script statements.
while / until	The **while** or **until** conditional clause must only appear once in any **do..loop** statement, i.e. either after **do** or after **loop**. Each condition is interpreted only the first time it is encountered but is evaluated for every time it encountered in the loop.
exit do	If an **exit do** clause is encountered inside the loop, the execution of the script will be transferred to the first statement after the **loop** clause denoting the end of the loop. An **exit do** clause can be made conditional by the optional use of a **when** or **unless** suffix.

Example:

```
// LOAD files file1.csv..file9.csv
```

```
Set a=1;
Do while a<10
LOAD * from file$(a).csv;
Let a=a+1;
Loop
```

End

The **End** script keyword is used to close **If**, **Sub** and **Switch** clauses.

Exit

The **Exit** script keyword is part of the **Exit Script** statement, but can also be used to exit **Do**, **For** or **Sub** clauses.

Exit script

This control statement stops script execution. It may be inserted anywhere in the script.

Syntax:

```
Exit Script [ (when | unless) condition ]
```

Since the **exit script** statement is a control statement and as such is ended with either a semicolon or end-of-line, it must not cross a line boundary.

Arguments:

Arguments

Argument	Description
condition	A logical expression evaluating to True or False.
when / unless	An **exit script** statement can be made conditional by the optional use of **when** or **unless** clause.

Examples:

```
//Exit script
Exit Script;

//Exit script when a condition is fulfilled
Exit Script when a=1
```

For..next

The **for..next** control statement is a script iteration construct with a counter. The statements inside the loop enclosed by **for** and **next** will be executed for each value of the counter variable between specified low and high limits.

Syntax:

```
For counter = expr1 to expr2 [ step expr3 ]
[statements]
[exit for [ ( when | unless ) condition ]
[statements]
Next [counter]
```

The expressions *expr1*, *expr2* and *expr3* are only evaluated the first time the loop is entered. The value of the counter variable may be changed by statements inside the loop, but this is not good programming practice.

If an **exit for** clause is encountered inside the loop, the execution of the script will be transferred to the first statement after the **next** clause denoting the end of the loop. An **exit for** clause can be made conditional by the optional use of a **when** or **unless** suffix.

> *Since the **for..next** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its three possible clauses (**for..to..step**, **exit for** and **next**) must not cross a line boundary.*

Arguments:

Arguments

Argument	Description
counter	A variable name. If *counter* is specified after **next** it must be the same variable name as the one found after the corresponding **for**.
expr1	An expression which determines the first value of the *counter* variable for which the loop should be executed.
expr2	An expression which determines the last value of the *counter* variable for which the loop should be executed.
expr3	An expression which determines the value indicating the increment of the *counter* variable each time the loop has been executed.
condition	a logical expression evaluating to True or False.
statements	Any group of one or more Qlik Sense script statements.

Example 1: Loading a sequence of files

```
// LOAD files file1.csv..file9.csv
for a=1 to 9

      LOAD * from file$(a).csv;

next
```

Example 2: Loading a random number of files

In this example, we assume there are data files *x1.csv, x3.csv, x5.csv, x7.csv* and *x9.csv*. Loading is stopped at a random point using the `if rand( )<0.5 then` condition.

```
for counter=1 to 9 step 2
      set filename=x$(counter).csv;
      if rand( )<0.5 then
            exit for unless counter=1
      end if
      LOAD a,b from $(filename);
next
```

For each..next

The **for each..next** control statement is a script iteration construct which executes one or several statements for each value in a comma separated list. The statements inside the loop enclosed by **for** and **next** will be executed for each value of the list.

Syntax:
Special syntax makes it possible to generate lists with file and directory names in the current directory.

```
for each var in list
[statements]
[exit for [ ( when | unless ) condition ]
[statements]
next [var]
```

Arguments:

Arguments

Argument	Description
var	A script variable name which will acquire a new value from list for each loop execution. If **var** is specified after **next** it must be the same variable name as the one found after the corresponding **for each**.

The value of the **var** variable may be changed by statements inside the loop, but this is not good programming practice.

If an **exit for** clause is encountered inside the loop, the execution of the script will be transferred to the first statement after the **next** clause denoting the end of the loop. An **exit for** clause can be made conditional by the optional use of a **when** or **unless** suffix.

*Since the **for each..next** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its three possible clauses (**for each**, **exit for** and **next**) must not cross a line boundary.*

Syntax:

```
list := item { , item }
item := constant | (expression) | filelist mask | dirlist mask |
fieldvaluelist mask
```

Arguments

Argument	Description
constant	Any number or string. Note that a string written directly in the script must be enclosed by single quotes. A string without single quotes will be interpreted as a variable, and the value of the variable will be used. Numbers do not need to be enclosed by single quotes.
expression	An arbitrary expression.
mask	A filename or folder name mask which may include any valid filename characters as well as the standard wildcard characters, ***** and **?**. You can use absolute file paths or lib:// paths.
condition	A logical expression evaluating to True or False.
statements	Any group of one or more Qlik Sense script statements.
filelist mask	This syntax produces a comma separated list of all files in the current directory matching the filename mask. *This argument supports only library connections in standard mode.*
dirlist mask	This syntax produces a comma separated list of all folders in the current folder matching the folder name mask. *This argument supports only library connections in standard mode.*
fieldvaluelist mask	This syntax iterates through the values of a field already loaded into Qlik Sense.

*The Qlik Web Storage Provider Connectors and other DataFiles connections do not support filter masks that use wildcard (***** and **?**) characters.*

Example 1: Loading a list of files

```
// LOAD the files 1.csv, 3.csv, 7.csv and xyz.csv
for each a in 1,3,7,'xyz'
   LOAD * from file$(a).csv;
next
```

Example 2: Creating a list of files on disk

This example loads a list of all Qlik Sense related files in a folder.

```
sub DoDir (Root)
   for each Ext in 'qvw', 'qva', 'qvo', 'qvs', 'qvc', 'qvf', 'qvd'

      for each File in filelist (Root&'/*.' &Ext)

         LOAD
            '$(File)' as Name,
            FileSize( '$(File)' ) as Size,
            FileTime( '$(File)' ) as FileTime
         autogenerate 1;

      next File

   next Ext
   for each Dir in dirlist (Root&'/*' )

      call DoDir (Dir)

   next Dir

end sub

call DoDir ('lib://DataFiles')
```

Example 3: Iterating through a the values of a field

This example iterates through the list of loaded values of FIELD and generates a new field, NEWFIELD. For each value of FIELD, two NEWFIELD records will be created.

```
load * inline [
FIELD
one
two
three
];

FOR Each a in FieldValueList('FIELD')
LOAD '$(a)' &'-'&RecNo() as NEWFIELD AutoGenerate 2;
NEXT a
```

The resulting table looks like this:

Example table

NEWFIELD
one-1
one-2
two-1
two-2
three-1
three-2

If..then..elseif..else..end if

The **if..then** control statement is a script selection construct forcing the script execution to follow different paths depending on one or several logical conditions.

Control statements are typically used to control the flow of the script execution. In a chart expression, use the **if** conditional function instead.

Syntax:

```
If  condition then
  [ statements ]
{ elseif condition then
  [ statements ] }
[ else
  [ statements ] ]
end if
```

Since the **if..then** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its four possible clauses (**if..then**, **elseif..then**, **else** and **end if**) must not cross a line boundary.

Arguments:

Arguments

Argument	Description
condition	A logical expression which can be evaluated as True or False.
statements	Any group of one or more Qlik Sense script statements.

Example 1:

```
if a=1 then

      LOAD * from abc.csv;
      SQL SELECT e, f, g from tab1;

end if
```

Example 2:

```
if a=1 then; drop table xyz; end if;
```

Example 3:

```
if x>0 then
      LOAD * from pos.csv;
elseif x<0 then
      LOAD * from neg.csv;
else
      LOAD * from zero.txt;
end if
```

Next

The **Next** script keyword is used to close **For** loops.

Sub..end sub

The **sub..end sub** control statement defines a subroutine which can be called upon from a **call** statement.

Syntax:

```
Sub name [ ( paramlist )] statements end sub
```

Arguments are copied into the subroutine and, if the corresponding actual parameter in the **call** statement is a variable name, copied back out again upon exiting the subroutine.

If a subroutine has more formal parameters than actual parameters passed by a **call** statement, the extra parameters will be initialized to NULL and can be used as local variables within the subroutine.

Arguments:

Arguments

Argument	Description
name	The name of the subroutine.
paramlist	A comma separated list of variable names for the formal parameters of the subroutine. These can be used as any variable inside the subroutine.
statements	Any group of one or more Qlik Sense script statements.

Limitations:

- Since the **sub** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its two clauses (**sub** and **end sub**) must not cross a line boundary.
- When you define a subroutine with `Sub..end sub` inside a control statement, for example `if..then`, you can only call the subroutine from within the same control statement.

Example 1:

```
Sub INCR (I,J)
I = I + 1
Exit Sub when I < 10
J = J + 1
End Sub
Call INCR (X,Y)
```

Example 2: - parameter transfer

```
Sub ParTrans (A,B,C)
A=A+1
B=B+1
C=C+1
End Sub
A=1
X=1
C=1
Call ParTrans (A, (X+1)*2)
```

The result of the above will be that locally, inside the subroutine, A will be initialized to 1, B will be initialized to 4 and C will be initialized to NULL.

When exiting the subroutine, the global variable A will get 2 as value (copied back from subroutine). The second actual parameter “(X+1)*2” will not be copied back since it is not a variable. Finally, the global variable C will not be affected by the subroutine call.

Switch..case..default..end switch

The **switch** control statement is a script selection construct forcing the script execution to follow different paths, depending on the value of an expression.

Syntax:

```
Switch expression {case valuelist [ statements ]} [default statements] end switch
```

*Since the **switch** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its four possible clauses (**switch**, **case**, **default** and **end switch**) must not cross a line boundary.*

Arguments:

Arguments

Argument	Description
expression	An arbitrary expression.
valuelist	A comma separated list of values with which the value of expression will be compared. Execution of the script will continue with the statements in the first group encountered with a value in valuelist equal to the value in expression. Each value in valuelist may be an arbitrary expression. If no match is found in any **case** clause, the statements under the **default** clause, if specified, will be executed.
statements	Any group of one or more Qlik Sense script statements.

Example:

```
Switch I
Case 1
LOAD '$(I): CASE 1' as case autogenerate 1;
Case 2
LOAD '$(I): CASE 2' as case autogenerate 1;
Default
LOAD '$(I): DEFAULT' as case autogenerate 1;
End Switch
```

To

The **To** script keyword is used in several script statements.

2.4 Script prefixes

Prefixes may be applied to applicable regular statements but never to control statements. The **when** and **unless** prefixes can however be used as suffixes to a few specific control statement clauses.

All script keywords can be typed with any combination of lower case and upper case characters. Field and variable names used in the statements are however case sensitive.

Script prefixes overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Add

The **Add** prefix can be added to any **LOAD** or **SELECT** statement in the script to specify that it should add records to another table. It also specifies that this statement should be run in a partial reload. The **Add** prefix can also be used in a **Map** statement.

```
Add [only] [Concatenate[(tablename )]] (loadstatement | selectstatement)
Add [ Only ] mapstatement
```

Buffer

QVD files can be created and maintained automatically via the **buffer** prefix. This prefix can be used on most **LOAD** and **SELECT** statements in script. It indicates that QVD files are used to cache/buffer the result of the statement.

```
Buffer[(option [ , option])] ( loadstatement | selectstatement )
option::= incremental | stale [after] amount [(days | hours)]
```

Concatenate

If two tables that are to be concatenated have different sets of fields, concatenation of two tables can still be forced with the **Concatenate** prefix.

```
Concatenate[ (tablename ) ] ( loadstatement | selectstatement )
```

Crosstable

The **crosstable** load prefix is used to transpose “cross table” or “pivot table” structured data. Data structured this way is commonly encountered when working with spreadsheet sources. The output and aim of the **crosstable** load prefix is to transpose such structures into a regular column-oriented table equivalent, as this structure is generally better suited for analysis in Qlik Sense.

```
Crosstable (attribute field name, data field name [ , n ] ) ( loadstatement |
selectstatement )
```

First

The **First** prefix to a **LOAD** or **SELECT (SQL)** statement is used for loading a set maximum number of records from a data source table.

```
First n( loadstatement | selectstatement )
```

Generic

The **Generic** load prefix allows for conversion of entity–attribute–value modeled data (EAV) into a traditional, normalized relational table structure. EAV modeling is alternatively referred to as "generic data modeling" or "open schema".

```
Generic ( loadstatement | selectstatement )
```

Hierarchy

The **hierarchy** prefix is used to transform a parent-child hierarchy table to a table that is useful in a Qlik Sense data model. It can be put in front of a **LOAD** or a **SELECT** statement and will use the result of the loading statement as input for a table transformation.

```
Hierarchy (NodeID, ParentID, NodeName, [ParentName], [PathSource],
[PathName], [PathDelimiter], [Depth])(loadstatement | selectstatement)
```

HierarchBelongsTo

This prefix is used to transform a parent-child hierarchy table to a table that is useful in a Qlik Sense data model. It can be put in front of a **LOAD** or a **SELECT** statement and will use the result of the loading statement as input for a table transformation.

```
HierarchyBelongsTo (NodeID, ParentID, NodeName, AncestorID, AncestorName,
[DepthDiff])(loadstatement | selectstatement)
```

Inner

The **join** and **keep** prefixes can be preceded by the prefix **inner**.

If used before **join** it specifies that an inner join should be used. The resulting table will thus only contain combinations of field values from the raw data tables where the linking field values are represented in both tables. If used before **keep**, it specifies that both raw data tables should be reduced to their common intersection before being stored in Qlik Sense.

.

```
Inner ( Join | Keep) [ (tablename) ](loadstatement |selectstatement )
```

IntervalMatch

The **IntervalMatch** prefix is used to create a table matching discrete numeric values to one or more numeric intervals, and optionally matching the values of one or several additional keys.

```
IntervalMatch (matchfield)(loadstatement | selectstatement )
IntervalMatch (matchfield,keyfield1 [ , keyfield2, ... keyfield5 ] )
(loadstatement | selectstatement )
```

Join

The **join** prefix joins the loaded table with an existing named table or the last previously created data table.

```
[Inner | Outer | Left | Right ] Join [ (tablename ) ]( loadstatement |
selectstatement )
```

Keep

The **keep** prefix is similar to the **join** prefix. Just as the **join** prefix, it compares the loaded table with an existing named table or the last previously created data table, but instead of joining the loaded table with an existing table, it has the effect of reducing one or both of the two tables before they are stored in Qlik Sense, based on the intersection of table data. The comparison made is equivalent to a natural join made over all the common fields, i.e. the same way as in a corresponding join. However, the two tables are not joined and will be kept in Qlik Sense as two separately named tables.

```
(Inner | Left | Right) Keep [(tablename ) ]( loadstatement | selectstatement
)
```

Left

The **Join** and **Keep** prefixes can be preceded by the prefix **left**.

If used before **join** it specifies that a left join should be used. The resulting table will only contain combinations of field values from the raw data tables where the linking field values are represented in the first table. If used before **keep**, it specifies that the second raw data table should be reduced to its common intersection with the first table, before being stored in Qlik Sense.

```
Left ( Join | Keep) [ (tablename) ](loadstatement |selectstatement )
```

Mapping

The **mapping** prefix is used to create a mapping table that can be used to, for example, replacing field values and field names during script execution.

```
Mapping  ( loadstatement | selectstatement )
```

Merge

The **Merge** prefix can be added to any **LOAD** or **SELECT** statement in the script to specify that the loaded table should be merged into another table. It also specifies that this statement should be run in a partial reload.

```
Merge [only] [(SequenceNoField [, SequenceNoVar])] On ListOfKeys [Concatenate
[(TableName)]] (loadstatement | selectstatement)
```

NoConcatenate

The **NoConcatenate** prefix forces two loaded tables with identical field sets to be treated as two separate internal tables, when they would otherwise be automatically concatenated.

```
NoConcatenate( loadstatement | selectstatement )
```

Outer

The explicit **Join** prefix can be preceded by the prefix **Outer** to specify an outer join. In an outer join, all combinations between the two tables are generated. The resulting table will thus contain combinations of field values from the raw data tables where the linking field values are represented in one or both tables. The **Outer** keyword is optional and is the default join type used when a join prefix is not specified.

```
Outer Join [ (tablename) ](loadstatement |selectstatement )
```

Partial reload

A full reload always starts by deleting all tables in the existing data model, and then runs the load script.

A *Partial reload (page 83)* will not do this. Instead it keeps all tables in the data model and then executes only **Load** and **Select** statements preceded by an **Add**, **Merge**, or **Replace** prefix. Other data tables are not affected by the command. The **only** argument denotes that the statement should be executed only during partial reloads, and should be disregarded during full reloads. The following table summarizes statement execution for partial and full reloads.

Replace

The **Replace** prefix can be added to any **LOAD** or **SELECT** statement in the script to specify that the loaded table should replace another table. It also specifies that this statement should be run in a partial reload. The **Replace** prefix can also be used in a **Map** statement.

```
Replace [only] [Concatenate[(tablename) ]] (loadstatement | selectstatement)
Replace [only] mapstatement
```

Right

The **Join** and **Keep** prefixes can be preceded by the prefix **right**.

If used before **join** it specifies that a right join should be used. The resulting table will only contain combinations of field values from the raw data tables where the linking field values are represented in the second table. If used before **keep**, it specifies that the first raw data table should be reduced to its common intersection with the second table, before being stored in Qlik Sense.

```
Right (Join | Keep) [(tablename)](loadstatement |selectstatement )
```

Sample

The **sample** prefix to a **LOAD** or **SELECT** statement is used for loading a random sample of records from the data source.

```
Sample p ( loadstatement | selectstatement )
```

Semantic

Tables containing relations between records can be loaded through a **semantic** prefix. This can for example be self-references within a table, where one record points to another, such as parent, belongs to, or predecessor.

```
Semantic ( loadstatement | selectstatement)
```

Unless

The **unless** prefix and suffix is used for creating a conditional clause which determines whether a statement or exit clause should be evaluated or not. It may be seen as a compact alternative to the full **if..end if** statement.

```
(Unless condition statement | exitstatement Unless condition )
```

When

The **when** prefix and suffix is used for creating a conditional clause which determines whether a statement or exit clause should be executed or not. It may be seen as a compact alternative to the full **if..end if** statement.

```
( When condition statement | exitstatement when condition )
```

Add

The **Add** prefix can be added to any **LOAD** or **SELECT** statement in the script to specify that it should add records to another table. It also specifies that this statement should be run in a partial reload. The **Add** prefix can also be used in a **Map** statement.

> *For partial reload to work properly, the app must be opened with data before a partial reload is triggered.*

Perform a partial reload using the **Reload** button. You can also use the Qlik Engine JSON API.

Syntax:

```
Add [only] [Concatenate[(tablename)]] (loadstatement | selectstatement)
```

```
Add [only] mapstatement
```

During a normal (non-partial) reload, the **Add LOAD** construction will work as a normal **LOAD** statement. Records will be generated and stored in a table.

If the **Concatenate** prefix is used, or if there exists a table with the same set of fields, the records will be appended to the relevant existing table. Otherwise, the **Add LOAD** construction will create a new table.

A partial reload will do the same. The only difference is that the **Add LOAD** construction will never create a new table. There always exists a relevant table from the previous script execution to which the records should be appended.

No check for duplicates is performed. Therefore, a statement using the **Add** prefix will often include either a distinct qualifier or a where clause guarding duplicates.

The **Add Map...Using** statement causes mapping to take place also during partial script execution.

Arguments:

Arguments

Argument	Description
only	An optional qualifier denoting that the statement should be executed only during partial reloads. It should be disregarded during normal (non-partial) reloads.

Examples and results:

Example	Result

`Tab1:`
`LOAD Name, Number FROM Persons.csv;`
`Add LOAD Name, Number FROM newPersons.csv;`
	During normal reload, data is loaded from *Persons.csv* and stored in the Qlik Sense table Tab1. Data from *NewPersons.csv* is then concatenated to the same Qlik Sense table.
	
	During partial reload, data is loaded from *NewPersons.csv* and appended to the Qlik Sense table Tab1. No check for duplicates is made.

`Tab1:`
`SQL SELECT Name, Number FROM Persons.csv;`
`Add LOAD Name, Number FROM NewPersons.csv where not exists(Name);`
	A check for duplicates is made by means of looking if Name exists in the previously loaded table data.
	
	During normal reload, data is loaded from *Persons.csv* and stored in the Qlik Sense table Tab1. Data from *NewPersons.csv* is then concatenated to the same Qlik Sense table.
	
	During partial reload, data is loaded from *NewPersons.csv* which is appended to the Qlik Sense table Tab1. A check for duplicates is made by means of seeing if Name exists in the previously loaded table data.

`Tab1:`
`LOAD Name, Number FROM Persons.csv;`
`Add Only LOAD Name, Number FROM NewPersons.csv where not exists(Name);`
	During normal reload, data is loaded from *Persons.csv* and stored in the Qlik Sense table Tab1. The statement loading *NewPersons.csv* is disregarded.
	
	During partial reload, data is loaded from *NewPersons.csv* which is appended to the Qlik Sense table Tab1. A check for duplicates is made by means of seeing if Name exists in the previously loaded table data.

Buffer

QVD files can be created and maintained automatically via the **buffer** prefix. This prefix can be used on most **LOAD** and **SELECT** statements in script. It indicates that QVD files are used to cache/buffer the result of the statement.

Syntax:

```
Buffer [(option [ , option])] ( loadstatement | selectstatement )
option::= incremental | stale [after] amount [(days | hours)]
```

If no option is used, the QVD buffer created by the first execution of the script will be used indefinitely.

The buffer file is stored in the *Buffers* sub-folder, typically *C:\ProgramData\Qlik\Sense\Engine\Buffers* (server installation) or *C:\Users\{user}\Documents\Qlik\Sense\Buffers* (Qlik Sense Desktop).

The name of the QVD file is a calculated name, a 160-bit hexadecimal hash of the entire following **LOAD** or **SELECT** statement and other discriminating info. This means that the QVD buffer will be rendered invalid by any change in the following **LOAD** or **SELECT** statement.

QVD buffers will normally be removed when no longer referenced anywhere throughout a complete script execution in the app that created it or when the app that created it no longer exists.

Arguments:

Arguments

Argument	Description
incremental	The incremental option enables the ability to read only part of an underlying file. Previous size of the file is stored in the XML header in the QVD file. This is particularly useful with log files. All records loaded at a previous occasion are read from the QVD file whereas the following new records are read from the original source and finally an updated QVD-file is created. The incremental option can only be used with **LOAD** statements and text files. Incremental load cannot be used where old data is changed or deleted.
stale [after] amount [(days | hours)]	amount is a number specifying the time period. Decimals may be used. The unit is assumed to be days if omitted. The stale after option is typically used with DB sources where there is no simple timestamp on the original data. Instead you specify how old the QVD snapshot can be to be used. A stale after clause simply states a time period from the creation time of the QVD buffer after which it will no longer be considered valid. Before that time the QVD buffer will be used as source for data and after that the original data source will be used. The QVD buffer file will then automatically be updated and a new period starts.

Limitations:

Numerous limitations exist, most notable is that there must be either a file **LOAD** or a **SELECT** statement at the core of any complex statement.

Example 1:

```
Buffer SELECT * from MyTable;
```

Example 2:

```
Buffer (stale after 7 days) SELECT * from MyTable;
```

Example 3:

```
Buffer (incremental) LOAD * from MyLog.log;
```

Concatenate

If two tables that are to be concatenated have different sets of fields, concatenation of two tables can still be forced with the **Concatenate** prefix. This statement forces concatenation with an existing named table or the latest previously created logical table.

A concatenation is in principle the same as the **SQL UNION** statement, but with two differences:

- The **Concatenate** prefix can be used no matter if the tables have identical field names or not.
- Identical records are not removed with the **Concatenate** prefix.

Arguments:

Arguments

Argument	Description
tablename	The name of the existing table.

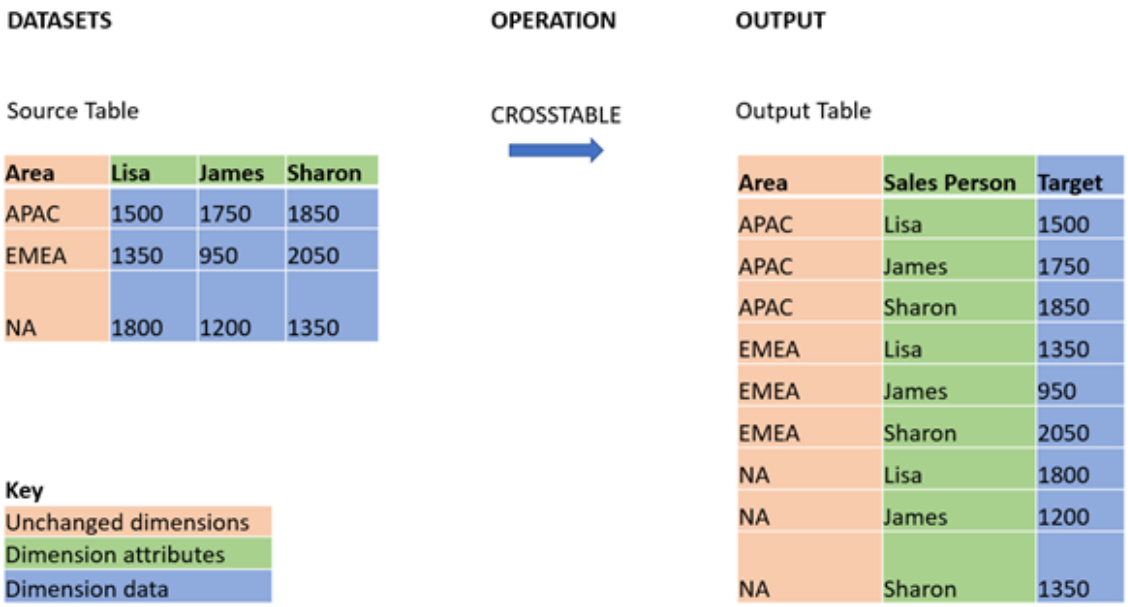
Example:

```
Concatenate LOAD * From file2.csv;
Concatenate SELECT * From table3;
tab1:
LOAD * From file1.csv;
tab2:
LOAD * From file2.csv;
.. .. ..
Concatenate (tab1) LOAD * From file3.csv;
```

Crosstable

The **crosstable** load prefix is used to transpose “cross table” or “pivot table” structured data. Data structured this way is commonly encountered when working with spreadsheet sources. The output and aim of the **crosstable** load prefix is to transpose such structures into a regular column-oriented table equivalent, as this structure is generally better suited for analysis in Qlik Sense.

Example of data structured as a crosstable and its equivalent structure after a crosstable transformation



Syntax:

```
crosstable (attribute field name, data field name [ , n ] ) ( loadstatement |
selectstatement )
```

Arguments

Argument	Description
attribute field name	The desired output field name describing the horizontally oriented dimension that is to be transposed (the header row).
data field name	The desired output field name which describes the horizontally oriented data of the dimension that is to be transposed (the matrix of data values beneath the header row).
n	The number of qualifier fields, or unchanged dimensions, preceding the table to be transformed to generic form. The default value is 1.

This scripting function is related to the following functions:

Related functions

Function	Interaction
Generic (page 50)	A transformation load prefix which takes an entity-attribute-value structured data set and transforms it into a regular relational table structure, separating each attribute encountered into a new field or column of data.

Example 1 – Transforming pivoted sales data (simple)

Load scripts and results

Overview

Open the Data load editor and add the first load script below to a new tab.

The first load script contains a dataset to which the `crosstable` script prefix will be applied later, with the section applying `crosstable` commented out. This means that comment syntax was used to disable this section in the load script.

The second load script is the same as the first, but with the application of `crosstable` uncommented (enabled by removing the comment syntax). The scripts are shown this way to highlight the value of this scripting function in transforming data.

First load script (function not applied)

```
tmpData:
//Crosstable (MonthText, Sales)
Load * inline [
Product, Jan 2021, Feb 2021, Mar 2021, Apr 2021, May 2021, Jun 2021
A, 100, 98, 103, 63, 108, 82
B, 284, 279, 297, 305, 294, 292
C, 50, 53, 50, 54, 49, 51];

//Final:
//Load Product,
//Date(Date#(MonthText,'MMM YYYY'),'MMM YYYY') as Month,
//Sales

//Resident tmpData;

//Drop Table tmpData;
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `Product`
- `Jan 2021`
- `Feb 2021`

- `Mar 2021`
- `Apr 2021`
- `May 2021`
- `Jun 2021`

Results table

Product	Jan 2021	Feb 2021	Mar 2021	Apr 2021	May 2021	Jun 2021
A	100	98	103	63	108	82
B	284	279	297	305	294	292
C	50	53	50	54	49	51

This script allows the creation of a crosstable with one column for each month and one row per product. In its current format, this data is not easy to analyze. It would be much better to have all numbers in one field and all months in another, in a three-column table. The next section explains how to do this transformation to the crosstable.

Second load script (function applied)

Uncomment the script by removing the `//`. The load script should look like this:

```
tmpData:
Crosstable (MonthText, Sales)
Load * inline [
Product, Jan 2021, Feb 2021, Mar 2021, Apr 2021, May 2021, Jun 2021
A, 100, 98, 103, 63, 108, 82
B, 284, 279, 297, 305, 294, 292
C, 50, 53, 50, 54, 49, 51];

Final:
Load Product,
Date(Date#(MonthText,'MMM YYYY'),'MMM YYYY') as Month,
Sales

Resident tmpData;

Drop Table tmpData;
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `Product`
- `Month`
- `Sales`

Results table

Product	Month	Sales
A	Jan 2021	100
A	Feb 2021	98
A	Mar 2021	103
A	Apr 2021	63
A	May 2021	108
A	Jun 2021	82
B	Jan 2021	284
B	Feb 2021	279
B	Mar 2021	297
B	Apr 2021	305
B	May 2021	294
B	Jun 2021	292
C	Jan 2021	50
C	Feb 2021	53
C	Mar 2021	50
C	Apr 2021	54
C	May 2021	49
C	Jun 2021	51

Once the script prefix has been applied, the crosstable is transformed into a straight table with one column for `Month` and another for `Sales`. This improves the readability of the data.

Example 2 – Transforming pivoted sales target data into a vertical table structure (intermediate)

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset which is loaded into a table named Targets.
- The `crosstable` load prefix, which transposes the pivoted sales person names into a field of its own,

labeled `Sales Person`.

- The associated sales target data, which is structured into a field called `Target`.

Load script

```
SalesTargets:
CROSSTABLE([Sales Person],Target,1)
LOAD
*
INLINE [
Area, Lisa, James, Sharon
APAC, 1500, 1750, 1850
EMEA, 1350, 950, 2050
NA, 1800, 1200, 1350
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `Area`
- `Sales Person`

Add this measure:

```
=Sum(Target)
```

Results table

Area	Sales Person	=Sum(Target)
APAC	James	1750
APAC	Lisa	1500
APAC	Sharon	1850
EMEA	James	950
EMEA	Lisa	1350
EMEA	Sharon	2050
NA	James	1200
NA	Lisa	1800
NA	Sharon	1350

If you want to replicate the display of data as the pivoted input table, you can create an equivalent pivot table in a sheet.

Do the following:

1. Copy and paste the table you have just created into the sheet.
2. Drag the **Pivot table** chart object on top of the newly created table copy. Select **Convert**.
3. Click ✔ **Done editing**.
4. Drag the `Sales Person` field from the vertical column shelf to the horizontal column shelf.

The following table shows the data in its initial table form, as it is displayed in Qlik Sense:

Original results table, as shown in Qlik Sense

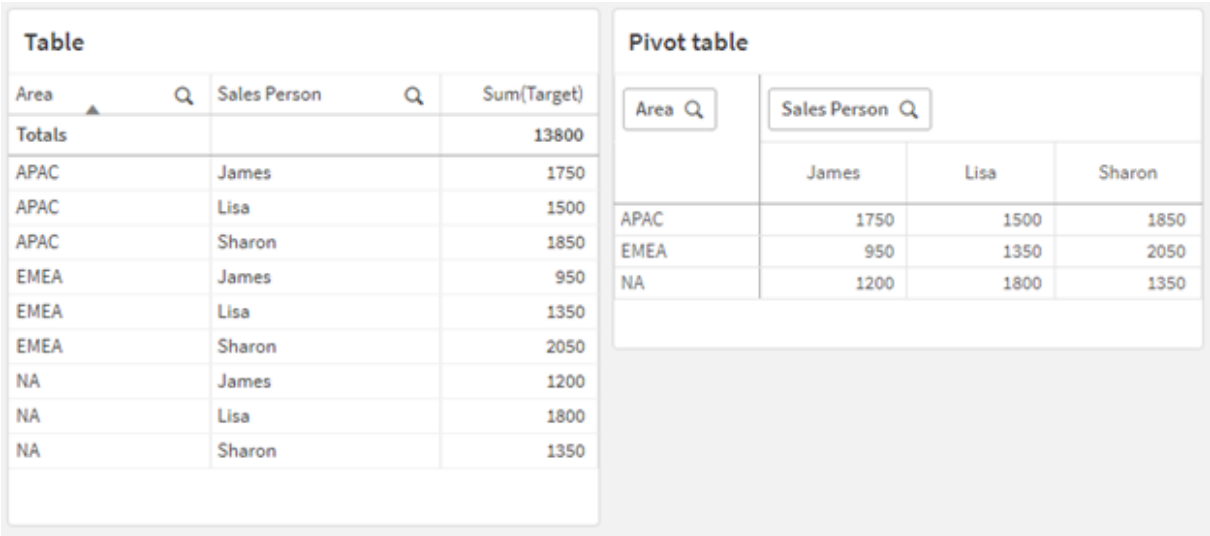
Area	Sales Person	=Sum(Target)
Totals	-	13800
APAC	James	1750
APAC	Lisa	1500
APAC	Sharon	1850
EMEA	James	950
EMEA	Lisa	1350
EMEA	Sharon	2050
NA	James	1200
NA	Lisa	1800
NA	Sharon	1350

The equivalent pivot table looks similar to the following, with the column for each sales person's name being contained within the larger row for `Sales Person`:

Equivalent pivot table with the `Sales Person` field pivoted horizontally

Area	James	Lisa	Sharon
APAC	1750	1500	1850
EMEA	950	1350	2050
NA	1350	1350	1350

Example of data displayed as a table and an equivalent pivot table with the `Sales Person` field pivoted horizontally



Table

Area	Sales Person	Sum(Target)
Totals		13800
APAC	James	1750
APAC	Lisa	1500
APAC	Sharon	1850
EMEA	James	950
EMEA	Lisa	1350
EMEA	Sharon	2050
NA	James	1200
NA	Lisa	1800
NA	Sharon	1350

Pivot table

Area	Sales Person		
	James	Lisa	Sharon
APAC	1750	1500	1850
EMEA	950	1350	2050
NA	1200	1800	1350

Example 3 – Transforming pivoted sales and target data into a vertical table structure (advanced)

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset representing sales and targets data, organized by area and month of the year. This is loaded into a table called `SalesAndTargets`.
- The `crosstable` load prefix. This is used to unpivot the `Month Year` dimension into a dedicated field, as well as to transpose the matrix of sales and target amounts into a dedicated field called `Amount`.
- A conversion of the `Month Year` field from text to a proper date, using the text-to-date conversion function `date#`. This date-converted `Month Year` field is joined back onto the `SalesAndTarget` table via a `Join` load prefix.

Load script

```
SalesAndTargets:
CROSSTABLE(MonthYearAsText,Amount,2)
LOAD
        *
INLINE [
Area    Type    Jan-22  Feb-22  Mar-22  Apr-22  May-22  Jun-22  Jul-22  Aug-22  Sep-22  Oct-22  Nov-22  Dec-22
APAC    Target  425     425     425     425     425     425     425     425     425     425     425     425
APAC    Actual  435     434     397     404     458     447     413     458     385     421     448     397
EMEA    Target  362.5   362.5   362.5   362.5   362.5   362.5   362.5   362.5   362.5   362.5   362.5   362.5
EMEA    Actual  363.5   359.5   337.5   361.5   341.5   337.5   379.5   352.5   327.5   337.5   360.5   334.5
```

```
NA     Target  375     375     375     375     375     375     375     375     375     375     375     375
NA     Actual  378     415     363     356     403     343     401     365     393     340     360     405
] (delimiter is '\t');

tmp:
LOAD DISTINCT MonthYearAsText,date#(MonthYearAsText,'MMM-YY') AS [Month Year]
RESIDENT SalesAndTargets;

JOIN (SalesAndTargets)
LOAD * RESIDENT tmp;

DROP TABLE tmp;
DROP FIELD MonthYearAsText;
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `Area`
- `Month Year`

Create the following measure, with the label `Actual`:

```
=Sum({<Type={'Actual'}>} Amount)
```

Also create this measure, with the label `Target`:

```
=Sum({<Type={'Target'}>} Amount)
```

Results table (cropped)

Area	Month Year	Actual	Target
APAC	Jan-22	435	425
APAC	Feb-22	434	425
APAC	Mar-22	397	425
APAC	Apr-22	404	425
APAC	May-22	458	425
APAC	Jun-22	447	425
APAC	Jul-22	413	425
APAC	Aug-22	458	425
APAC	Sep-22	385	425
APAC	Oct-22	421	425
APAC	Nov-22	448	425
APAC	Dec-22	397	425

Area	Month Year	Actual	Target
EMEA	Jan-22	363.5	362.5
EMEA	Feb-22	359.5	362.5

If you wish to replicate the display of data as the pivoted input table, you can create an equivalent pivot table in a sheet.

Do the following:

1. Copy and paste the table you have just created into the sheet.
2. Drag the **Pivot table** chart object on top of the newly created table copy. Select **Convert**.
3. Click ✔ **Done editing**.
4. Drag the `Month Year` field from the vertical column shelf to the horizontal column shelf.
5. Drag the `values` item from the horizontal column shelf to the vertical column shelf.

The following table shows the data in its initial table form, as it is displayed in Qlik Sense:

Original results table (cropped), as shown in Qlik Sense

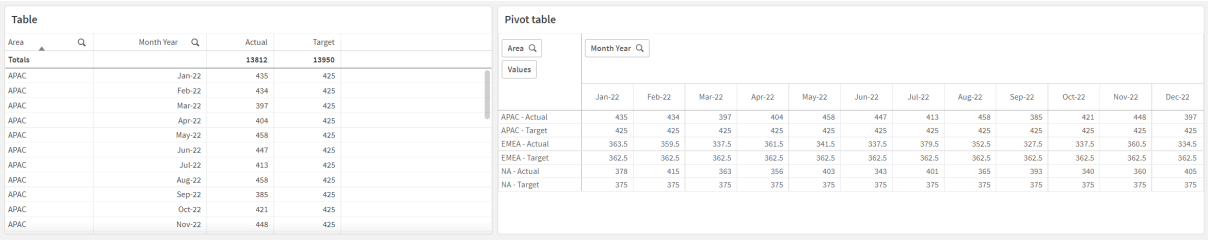
Area	Month Year	Actual	Target
Totals	-	13812	13950
APAC	Jan-22	435	425
APAC	Feb-22	434	425
APAC	Mar-22	397	425
APAC	Apr-22	404	425
APAC	May-22	458	425
APAC	Jun-22	447	425
APAC	Jul-22	413	425
APAC	Aug-22	458	425
APAC	Sep-22	385	425
APAC	Oct-22	421	425
APAC	Nov-22	448	425
APAC	Dec-22	397	425
EMEA	Jan-22	363.5	362.5
EMEA	Feb-22	359.5	362.5

The equivalent pivot table looks similar to the following, with the column for each individual month of the year being contained within the larger row for `Month Year`:

Equivalent pivot table (cropped) with the `Month Year` field pivoted horizontally

Area (Values)	**Jan-22**	**Feb-22**	**Mar-22**	**Apr-22**	**May-22**	**Jun-22**	**Jul-22**	**Aug-22**	**Sep-22**	**Oct-22**	**Nov-22**	**Dec-22**
APAC - Actual	435	434	397	404	458	447	413	458	385	421	448	397
APAC - Target	425	425	425	425	425	425	425	425	425	425	425	425
EMEA - Actual	363.5	359.5	337.5	361.5	341.5	337.5	379.5	352.5	327.5	337.5	360.5	334.5
EMEA - Target	362.5	362.5	362.5	362.5	362.5	362.5	362.5	362.5	362.5	362.5	362.5	362.5
NA - Actual	378	415	363	356	403	343	401	365	393	340	360	405
NA - Target	375	375	375	375	375	375	375	375	375	375	375	375

Example of data displayed as a table and an equivalent pivot table with the `Month Year` field pivoted horizontally



First

The `First` prefix to a `LOAD` or `SELECT` (SQL) statement is used for loading a set maximum number of records from a data source table. A typical use case for using the `First` prefix is when you want to retrieve a small subset of records from a large and/or slow data load step. As soon as the defined "`n`" number of records has been loaded, the load step terminates prematurely, and the rest of the script execution continues as normal.

Syntax:

```
First n ( loadstatement | selectstatement )
```

Arguments

Argument	Description
n	An arbitrary expression that evaluates to an integer indicating the maximum number of records to be read. n can also be enclosed in parentheses: (n).
loadstatement | selectstatement	The load statement/select statement that follows the n argument will define the specified table that must be loaded with the set maximum number of records.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the SET DateFormat statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Function examples

Example	Result
FIRST 10 LOAD * from abc.csv;	This example will retrieve the first ten lines from an excel file.
FIRST (1) SQL SELECT * from Orders;	This example will retrieve the first selected line from the Orders dataset.

Example – Load the first five rows

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates from the first two weeks of 2020.
- The First variable that instructs the application to only load the first five records.

Load script

```
Sales:
FIRST 5
LOAD
*
Inline [
date,sales
```

```
01/01/2020,6000
01/02/2020,3000
01/03/2020,6000
01/04/2020,8000
01/05/2020,5000
01/06/2020,7000
01/07/2020,3000
01/08/2020,5000
01/09/2020,9000
01/10/2020,5000
01/11/2020,7000
01/12/2020,7000
01/13/2020,7000
01/14/2020,7000
];
```

Results

Load the data and open a sheet. Create a new table and add `Date` as a field and `sum(sales)` as a measure.

Results table

Date	sum(sales)
01/01/2020	6000
01/02/2020	3000
01/03/2020	6000
01/04/2020	8000
01/05/2020	5000

The script only loads the first five records of the `Sales` table.

Generic

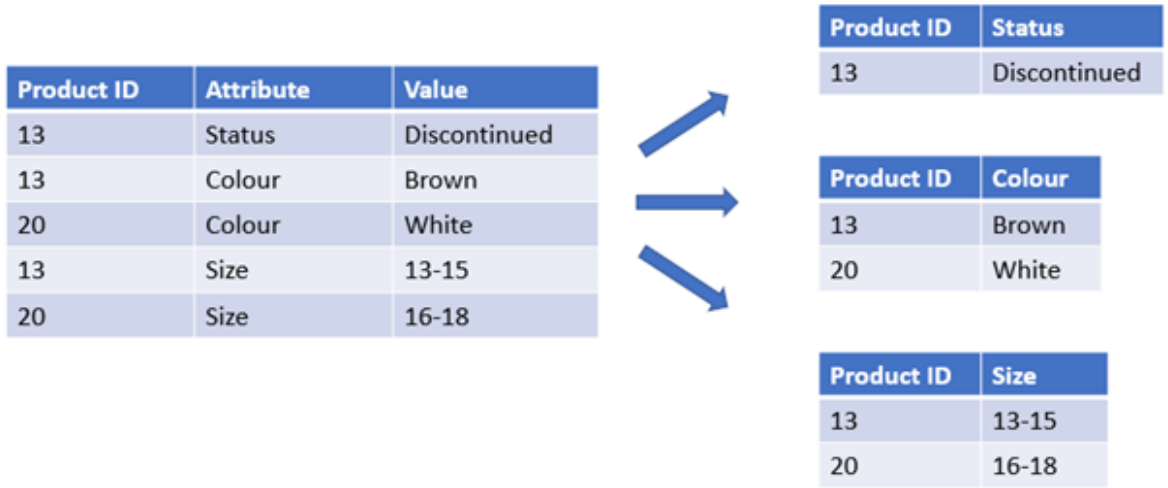
The **Generic** load prefix allows for conversion of entity–attribute–value modeled data (EAV) into a traditional, normalized relational table structure. EAV modeling is alternatively referred to as "generic data modeling" or "open schema".

Example of EAV modeled data and an equivalent denormalized relational table

Product ID	Attribute	Value
13	Status	Discontinued
13	Colour	Brown
20	Colour	White
13	Size	13-15
20	Size	16-18

Product ID	Status	Colour	Size
13	Discontinued	Brown	13-15
20		White	16-18

Example of EAV modeled data and an equivalent set of normalized relational tables



While it is technically possible to load and analyze EAV modeled data in Qlik, it is often easier to work with an equivalent traditional relational data structure.

Syntax:

```
Generic( loadstatement | selectstatement )
```

These topics may help you work with this function:

Related topics

Topic	Description
Crosstable (page 39)	The `Crosstable` load prefix transforms data that is horizontally-oriented into vertically-oriented data. From a purely functional perspective, it performs the opposite transformation to the `Generic` load prefix, although the prefixes typically serve entirely different use cases.
Generic databases in *Manage data*	EAV structured data models are further described here.

Example 1 – Transforming EAV structured data with the Generic load prefix

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains a dataset which is loaded into a table named `Transactions`. The dataset includes a date field. The default `MonthNames` definition is used.

Load script

```
Products:
Generic
Load * inline [
Product ID, Attribute, Value
13, Status, Discontinued
13, Color, Brown
20, Color, White
13, Size, 13-15
20, Size, 16-18
2, Status, Discontinued
5, Color, Brown
2, Color, White
44, Color, Brown
45, Size, 16-18
45, Color, Brown
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension: `Color`.

Add this measure:

```
=Count([Product ID])
```

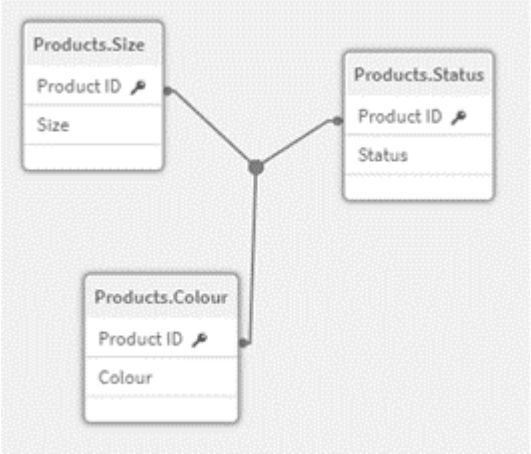
Now you can inspect the number of products by color.

Results table

Color	=Count([Product ID])
Brown	4
White	2

Note the shape of the data model, where each attribute has been broken out into a separate table named according to the original target table tag `Product`.Each table has the attribute as a suffix. One example of this is `Product.Color`. The resulting Product Attribute output records are associated by the `Product ID`.

Data model viewer representation of the results



Resulting table of records: Products.Status

Product ID	Status
13	Discontinued
2	Discontinued

Resulting table of records: Products.Size

Product ID	Size
13	13-15
20	16-18
45	16-18

Resulting table of records: Products.Color

Product ID	Color
13	Brown
5	Brown
44	Brown
45	Brown
20	White
2	White

Example 2 – Analyzing EAV structured data without the Generic load prefix

Load script and chart expression

Overview

This example shows how to analyze EAV structured data in its original form.

Open the Data load editor and add the load script below to a new tab.

The load script contains a dataset which is loaded into a table named `Products` in an EAV structure.

In this example, we are still counting products by color attribute. In order to analyze data structured in this way, you will need to apply expression-level filtering of products carrying the Attribute value `Color`.

Furthermore, individual attributes are not available to select as dimensions or fields, making it harder to determine how to build effective visualizations.

Load script

```
Products:
Load * Inline
[
Product ID, Attribute, Value
13, Status, Discontinued
13, Color, Brown
20, Color, White
13, Size, 13-15
20, Size, 16-18
2, Status, Discontinued
5, Color, Brown
2, Color, White
44, Color, Brown
45, Size, 16-18
45, Color, Brown
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension: `Value`.

Create the following measure:

```
=Count({<Attribute={'Color'}>} [Product ID])
```

Now you can inspect the number of products by color.

Resulting table of records: Products.Status

Value	=Count({<Attribute={'Color'}>} [Product ID])
Brown	4
White	2

Example 3 – Denormalizing the resulting output tables from a Generic load (advanced)

Load script and chart expression

Overview

In this example, we show how the normalised data structure produced by the `Generic` load prefix can be denormalised back into a consolidated `Product` dimension table. This is an advanced modeling technique which can be employed as part of data model performance tuning.

Open the Data load editor and add the load script below to a new tab.

Load script

```
Products:

Generic
Load * inline [
Product ID, Attribute, Value
13, Status, Discontinued
13, Color, Brown
20, Color, White
13, Size, 13-15
20, Size, 16-18
2, Status, Discontinued
5, Color, Brown
2, Color, White
44, Color, Brown
45, Size, 16-18
45, Color, Brown
];

RENAME TABLE Products.Color TO Products;

OUTER JOIN (Products)
LOAD * RESIDENT Products.Size;

OUTER JOIN (Products)
LOAD * RESIDENT Products.Status;
DROP TABLES Products.Size,Products.Status;
```

Results

Open the Data model viewer and note the shape of the resulting data model. Only one denormalized table is present. It is a combination of the three intermediary output tables: `Products.Size`, `Products.Status`, and `Products.Color`.

Resulting internal data model

Products
Product ID
Status
Color
Size

Resulting table of records: Products

Product ID	Status	Color	Size
13	Discontinued	Brown	13-15
20	-	White	16-18
2	Discontinued	White	-
5	-	Brown	-
44	-	Brown	-
45	-	Brown	16-18

Load the data and open a sheet. Create a new table and add this field as a dimension: `Color`.

Add this measure:

```
=Count([Product ID])
```

Results table

Color	=Count([Product ID])
Brown	4
White	2

Hierarchy

The **hierarchy** prefix is used to transform a parent-child hierarchy table to a table that is useful in a Qlik Sense data model. It can be put in front of a **LOAD** or a **SELECT** statement and will use the result of the loading statement as input for a table transformation.

The prefix creates an expanded nodes table, which normally has the same number of records as the input table, but in addition each level in the hierarchy is stored in a separate field. The path field can be used in a tree structure.

Syntax:

```
Hierarchy (NodeID, ParentID, NodeName, [ParentName, [PathSource, [PathName,
[PathDelimiter, Depth]]]])(loadstatement | selectstatement)
```

The input table must be an adjacent nodes table. Adjacent nodes tables are tables where each record corresponds to a node and has a field that contains a reference to the parent node. In such a table the node is stored on one record only but the node can still have any number of children. The table may of course contain additional fields describing attributes for the nodes.

The prefix creates an expanded nodes table, which normally has the same number of records as the input table, but in addition each level in the hierarchy is stored in a separate field. The path field can be used in a tree structure.

Usually the input table has exactly one record per node and in such a case the output table will contain the same number of records. However, sometimes there are nodes with multiple parents, i.e. one node is represented by several records in the input table. If so, the output table may have more records than the input table.

All nodes with a parent id not found in the node id column (including nodes with missing parent id) will be considered as roots. Also, only nodes with a connection to a root node - direct or indirect - will be loaded, thus avoiding circular references.

Additional fields containing the name of the parent node, the path of the node and the depth of the node can be created.

Arguments:

Arguments

Argument	Description
NodeID	The name of the field that contains the node id. This field must exist in the input table.
ParentID	The name of the field that contains the node id of the parent node. This field must exist in the input table.
NodeName	The name of the field that contains the name of the node. This field must exist in the input table.
ParentName	A string used to name the new **ParentName** field. If omitted, this field will not be created.
ParentSource	The name of the field that contains the name of the node used to build the node path. Optional parameter. If omitted, **NodeName** will be used.
PathName	A string used to name the new **Path** field, which contains the path from the root to the node. Optional parameter. If omitted, this field will not be created.
PathDelimiter	A string used as delimiter in the new **Path** field. Optional parameter. If omitted, '/' will be used.
Depth	A string used to name the new **Depth** field, which contains the depth of the node in the hierarchy. Optional parameter. If omitted, this field will not be created.

Example:

```
Hierarchy(NodeID, ParentID, NodeName, ParentName, NodeName, PathName, '\', Depth) LOAD *
inline [
NodeID, ParentID, NodeName
1, 4, London
2, 3, Munich
3, 5, Germany
4, 5, UK
5, , Europe
];
```

NodeID	ParentID	NodeName	NodeName1	NodeName2	NodeName3	ParentName	PathName	Depth
1	4	London	Europe	UK	London	UK	Europe\UK\London	3
2	3	Munich	Europe	Germany	Munich	Germany	Europe\Germany\Munich	3
3	5	Germany	Europe	Germany	-	Europe	Europe\Germany	2
4	5	UK	Europe	UK	-	Europe	Europe\UK	2
5		Europe	Europe	-	-	-	Europe	1

HierarchyBelongsTo

This prefix is used to transform a parent-child hierarchy table to a table that is useful in a Qlik Sense data model. It can be put in front of a **LOAD** or a **SELECT** statement and will use the result of the loading statement as input for a table transformation.

The prefix creates a table containing all ancestor-child relations of the hierarchy. The ancestor fields can then be used to select entire trees in the hierarchy. The output table in most cases contains several records per node.

Syntax:

```
HierarchyBelongsTo (NodeID, ParentID, NodeName, AncestorID, AncestorName,
[DepthDiff])(loadstatement | selectstatement)
```

The input table must be an adjacent nodes table. Adjacent nodes tables are tables where each record corresponds to a node and has a field that contains a reference to the parent node. In such a table the node is stored on one record only but the node can still have any number of children. The table may of course contain additional fields describing attributes for the nodes.

The prefix creates a table containing all ancestor-child relations of the hierarchy. The ancestor fields can then be used to select entire trees in the hierarchy. The output table in most cases contains several records per node.

An additional field containing the depth difference of the nodes can be created.

Arguments:

Arguments

Argument	Description
NodeID	The name of the field that contains the node id. This field must exist in the input table.
ParentID	The name of the field that contains the node id of the parent node. This field must exist in the input table.
NodeName	The name of the field that contains the name of the node. This field must exist in the input table.
AncestorID	A string used to name the new ancestor id field, which contains the id of the ancestor node.
AncestorName	A string used to name the new ancestor field, which contains the name of the ancestor node.
DepthDiff	A string used to name the new **DepthDiff** field, which contains the depth of the node in the hierarchy relative the ancestor node. Optional parameter. If omitted, this field will not be created.

Example:

```
HierarchyBelongsTo (NodeID, AncestorID, NodeName, AncestorID, AncestorName, DepthDiff) LOAD *
inline [
NodeID, AncestorID, NodeName
1, 4, London
2, 3, Munich
3, 5, Germany
4, 5, UK
5, , Europe
];
```

Results

NodeID	AncestorID	NodeName	AncestorName	DepthDiff
1	1	London	London	0
1	4	London	UK	1
1	5	London	Europe	2
2	2	Munich	Munich	0
2	3	Munich	Germany	1
2	5	Munich	Europe	2
3	3	Germany	Germany	0
3	5	Germany	Europe	1

NodeID	AncestorID	NodeName	AncestorName	DepthDiff
4	4	UK	UK	0
4	5	UK	Europe	1
5	5	Europe	Europe	0

Inner

The **join** and **keep** prefixes can be preceded by the prefix **inner**. If used before **join** it specifies that an inner join should be used. The resulting table will thus only contain combinations of field values from the raw data tables where the linking field values are represented in both tables. If used before **keep**, it specifies that both raw data tables should be reduced to their common intersection before being stored in Qlik Sense.

Syntax:

```
Inner ( Join | Keep) [ (tablename) ](loadstatement |selectstatement )
```

Arguments:

Arguments

Argument	Description
tablename	The named table to be compared to the loaded table.
loadstatement or selectstatement	The **LOAD** or **SELECT** statement for the loaded table.

Example

Load script

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
Table1:
Load * inline [
Column1, Column2
A, B
1, aa
2, cc
3, ee ];

Table2:
Inner Join Load * inline [
Column1, Column3
A, C
1, xx
4, yy ];
```

Result

Resulting table

Column1	Column2	Column3
A	B	C
1	aa	xx

Explanation

This example demonstrates the Inner Join output where only values present in both the first (left) and the second (right) tables are joined.

IntervalMatch

The **IntervalMatch** prefix is used to create a table matching discrete numeric values to one or more numeric intervals, and optionally matching the values of one or several additional keys.

Syntax:

```
IntervalMatch (matchfield)(loadstatement | selectstatement )
IntervalMatch (matchfield,keyfield1 [ , keyfield2, ... keyfield5 ] )
(loadstatement | selectstatement )
```

The **IntervalMatch** prefix must be placed before a **LOAD** or a **SELECT** statement that loads the intervals. The field containing the discrete data points (Time in the example below) and additional keys must already have been loaded into Qlik Sense before the statement with the **IntervalMatch** prefix. The prefix does not by itself read this field from the database table. The prefix transforms the loaded table of intervals and keys to a table that contains an additional column: the discrete numeric data points. It also expands the number of records so that the new table has one record per possible combination of discrete data point, interval and value of the key field(s).

The intervals may be overlapping and the discrete values will be linked to all matching intervals.

When the IntervalMatch prefix is extended with key fields, it is used to create a table matching discrete numeric values to one or more numeric intervals, while at the same time matching the values of one or several additional keys.

In order to avoid undefined interval limits being disregarded, it may be necessary to allow NULL values to map to other fields that constitute the lower or upper limits to the interval. This can be handled by the **NullAsValue** statement or by an explicit test that replaces NULL values with a numeric value well before or after any of the discrete numeric data points.

Arguments:

Arguments

Argument	Description
matchfield	The field containing the discrete numeric values to be linked to intervals.
keyfield	Fields that contain the additional attributes that are to be matched in the transformation.
loadstatement orselectstatement	Must result in a table, where the first field contains the lower limit of each interval, the second field contains the upper limit of each interval, and in the case of using key matching, the third and any subsequent fields contain the keyfield(s) present in the **IntervalMatch** statement. The intervals are always closed, i.e. the end points are included in the interval. Non-numeric limits render the interval to be disregarded (undefined).

Example 1:

In the two tables below, the first one lists a number of discrete events and the second one defines the start and end times for the production of different orders. By means of the **IntervalMatch** prefix it is possible to logically connect the two tables in order to find out e.g. which orders were affected by disturbances and which orders were processed by which shifts.

```
EventLog:
LOAD * Inline [
Time, Event, Comment
00:00, 0, Start of shift 1
01:18, 1, Line stop
02:23, 2, Line restart 50%
04:15, 3, Line speed 100%
08:00, 4, Start of shift 2
11:43, 5, End of production
];

OrderLog:
LOAD * INLINE [
Start, End, Order
01:00, 03:35, A
02:30, 07:58, B
03:04, 10:27, C
07:23, 11:43, D
];

//Link the field Time to the time intervals defined by the fields Start and End.
Inner Join IntervalMatch ( Time )
LOAD Start, End
Resident OrderLog;
```

The table **OrderLog** contains now an additional column: *Time*. The number of records is also expanded.

Table with additional column

Time	Start	End	Order
00:00	-	-	-
01:18	01:00	03:35	A
02:23	01:00	03:35	A
04:15	02:30	07:58	B
04:15	03:04	10:27	C
08:00	03:04	10:27	C
08:00	07:23	11:43	D
11:43	07:23	11:43	D

Example 2: (using keyfield)

Same example than above, adding *ProductionLine* as a key field.

```
EventLog:
LOAD * Inline [
Time, Event, Comment, ProductionLine
00:00, 0, Start of shift 1, P1
01:00, 0, Start of shift 1, P2
01:18, 1, Line stop, P1
02:23, 2, Line restart 50%, P1
04:15, 3, Line speed 100%, P1
08:00, 4, Start of shift 2, P1
09:00, 4, Start of shift 2, P2
11:43, 5, End of production, P1
11:43, 5, End of production, P2
];

OrderLog:
LOAD * INLINE [
Start, End, Order, ProductionLine
01:00, 03:35, A, P1
02:30, 07:58, B, P1
03:04, 10:27, C, P1
07:23, 11:43, D, P2
];

//Link the field Time to the time intervals defined by the fields Start and End and match the
values
// to the key ProductionLine.
Inner Join
IntervalMatch ( Time, ProductionLine )
LOAD Start, End, ProductionLine
Resident OrderLog;
```

A table box could now be created as below:

Tablebox example

ProductionLine	Time	Event	Comment	Order	Start	End
P1	00:00	0	Start of shift 1	-	-	-
P2	01:00	0	Start of shift 1	-	-	-
P1	01:18	1	Line stop	A	01:00	03:35
P1	02:23	2	Line restart 50%	A	01:00	03:35
P1	04:15	3	Line speed 100%	B	02:30	07:58
P1	04:15	3	Line speed 100%	C	03:04	10:27
P1	08:00	4	Start of shift 2	C	03:04	10:27
P2	09:00	4	Start of shift 2	D	07:23	11:43
P1	11:43	5	End of production	-	-	-
P2	11:43	5	End of production	D	07:23	11:43

Join

The **join** prefix joins the loaded table with an existing named table or the last previously created data table.

The effect of joining data is to extend the target table by an additional set of fields or attributes, namely ones not already present in the target table. Any common field names between the source data set and the target table are used to work out how to associate the new incoming records. This is commonly referred to as a “natural join”. A Qlik join operation can lead to the resulting target table having more or fewer records than it started with, depending on the uniqueness of the join association and the type of join employed.

There are four types of joins:

Left join

Left joins are the most common join type. For example, if you have a transaction data set and would like to combine it with a reference data set, you would typically use a `Left Join`. You would load the transaction table first, then load the reference data set while joining it via a `Left Join` prefix onto the already loaded transaction table. A `Left Join` would keep all transactions as-is and add on the supplementary reference data fields where a match is found.

Inner join

When you have two data sets where you only care about any results where there is a matching association, consider using an `Inner Join`. This will eliminate all records from both the source data loaded and the target table if no match is found. As a result, this may leave your target table with fewer records than before the join operation took place.

Outer join

When you need to keep both the target records and all of the incoming records, use an `Outer Join`. Where no match is found, each set of records is still kept while the fields from the opposite side of the join will remain unpopulated (null).

If the type keyword is omitted, the default join type is an outer join.

Right join

This join type keeps all the records about to be loaded, while reducing the records in the table targeted by the join to only those records where there is an association match in the incoming records. This is a niche join type that is sometimes used as a means of trimming down an already pre-loaded table of records to a required subset.

Example results sets from different types of join operations

DATASETS

Target Table

Trade ID	Asset Class
101533	Fixed Income
606601	Commodities

Incoming Dataset

Trade ID	Exchange
101533	LSE
79052	Hong Kong

OPERATION / **OUTPUT**

LEFT JOIN →

Trade ID	Asset Class	
101533	Fixed Income	LSE
606601	Commodities	

INNER JOIN →

Trade ID	Asset Class	
101533	Fixed Income	LSE

OUTER JOIN →

Trade ID	Asset Class	
101533	Fixed Income	LSE
606601	Commodities	
79052		Hong Kong

RIGHT JOIN →

Trade ID	Asset Class	
101533	Fixed Income	LSE
79052		Hong Kong

If there are no field names in common between the source and target of a join operation, the join will result in a cartesian product of all rows – this is called a "cross join".

Example result set from a "cross join" operation

Syntax:

```
[inner | outer | left | right ]Join [ (tablename ) ]( loadstatement |
selectstatement )
```

Arguments

Argument	Description
tablename	The named table to be compared to the loaded table.
loadstatement or selectstatement	The **LOAD** or **SELECT** statement for the loaded table.

These topics may help you work with this function:

Related topics

Topic	Description
Combining tables with Join and Jeep in *Manage data*	This topic provides further explanation of the concepts of “joining” and “keeping” data sets.
Keep (page 73)	The `Keep` load prefix is similar to the `Join` prefix, but it does not combine the source and target datasets. Instead, it trims each dataset according to the type of operation adopted (inner, outer, left, or right).

Example 1 - Left join: Enriching a target table with a reference data set

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset representing change records, which is loaded into a table named `Changes`. It includes a Status ID key field.
- A second dataset representing change statuses, which is loaded and combined with the original change records by joining it with a left `Join` load prefix.

This left join ensures that the change records remain intact while adding on status attributes where a match in the incoming status records is found based on a common Status ID.

Load script

```
Changes:
Load * inline [
Change ID      Status ID       Scheduled Start Date    Scheduled End Date      Business Impact
10030  4       19/01/2022      23/02/2022      None
10015  3       04/01/2022      15/02/2022      Low
10103  1       02/04/2022      29/05/2022      Medium
10185  2       23/06/2022      08/09/2022      None
10323  1       08/11/2022      26/11/2022      High
10326  2       11/11/2022      05/12/2022      None
10138  2       07/05/2022      03/08/2022      None
10031  3       20/01/2022      25/03/2022      Low
10040  1       29/01/2022      22/04/2022      None
10134  1       03/05/2022      08/07/2022      Low
10334  2       19/11/2022      06/02/2023      Low
10220  2       28/07/2022      06/09/2022      None
10264  1       10/09/2022      17/10/2022      Medium
10116  1       15/04/2022      24/04/2022      None
10187  2       25/06/2022      24/08/2022      Low
] (delimiter is '\t');

Status:
Left Join (Changes)
Load * inline [
Status ID      Status  Sub Status
1      Open    Not Started
2      Open    Started
3      Closed  Complete
4      Closed  Cancelled
5      Closed  Obsolete
] (delimiter is '\t');
```

Results

Open the Data model viewer and note the shape of the data model. Only one denormalized table is present. It is a combination of all the original change records, with the matching status attributes joined onto each change record.

Resulting internal data model

Changes
Change ID
Status ID
Scheduled Start Date
Scheduled End Date
Business Impact
Status
Sub Status

If you expand the preview window in the Data model viewer, you will see a portion of this full result set organized into a table:

Preview of Changes table in the Data model viewer

Change ID	**Status ID**	**Scheduled Start Date**	**Scheduled End Date**	**Business Impact**	**Status**	**Sub Status**
10015	3	04/01/2022	15/02/2022	Low	Closed	Complete
10030	4	19/01/2022	23/02/2022	None	Closed	Cancelled
10031	3	20/01/2022	25/03/2022	Low	Closed	Complete
10040	1	29/01/2022	22/04/2022	None	Open	Not Started
10103	1	02/04/2022	29/05/2022	Medium	Open	Not Started
10116	1	15/04/2022	24/04/2022	None	Open	Not Started
10134	1	03/05/2022	08/07/2022	Low	Open	Not Started
10138	2	07/05/2022	03/08/2022	None	Open	Started
10185	2	23/06/2022	08/09/2022	None	Open	Started
10187	2	25/06/2022	24/08/2022	Low	Open	Started
10220	2	28/07/2022	06/09/2022	None	Open	Started
10264	1	10/09/2022	17/10/2022	Medium	Open	Not Started
10323	1	08/11/2022	26/11/2022	High	Open	Not Started
10326	2	11/11/2022	05/12/2022	None	Open	Started
10334	2	19/11/2022	06/02/2023	Low	Open	Started

Since the fifth row in the Status table (Status ID: '5', Status: 'Closed', Sub Status: 'Obsolete') does not correspond to any of the records in the Changes table, the information in this row does not appear in the result set above.

Return to the Data load editor. Load the data and open a sheet. Create a new table and add this field as a dimension: `Status`.

Add this measure:

```
=Count([Change ID])
```

Now you can inspect the number of Changes by Status.

Results table

Status	=Count([Change ID])
Open	12
Closed	3

Example 2 – Inner join: Combining matching records only

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset representing change records, which is loaded into a table named `Changes`.
- A second dataset representing change records originating from the source system `JIRA`.This is loaded and combined with the original records by joining it with an `Inner Join` load prefix.

This `Inner Join` ensures that only the five change records which are found in both datasets are kept.

Load script

```
Changes:
Load * inline [
Change ID      Status ID       Scheduled Start Date    Scheduled End Date      Business Impact
10030  4       19/01/2022      23/02/2022      None
10015  3       04/01/2022      15/02/2022      Low
10103  1       02/04/2022      29/05/2022      Medium
10185  2       23/06/2022      08/09/2022      None
10323  1       08/11/2022      26/11/2022      High
10326  2       11/11/2022      05/12/2022      None
10138  2       07/05/2022      03/08/2022      None
10031  3       20/01/2022      25/03/2022      Low
10040  1       29/01/2022      22/04/2022      None
10134  1       03/05/2022      08/07/2022      Low
10334  2       19/11/2022      06/02/2023      Low
10220  2       28/07/2022      06/09/2022      None
10264  1       10/09/2022      17/10/2022      Medium
10116  1       15/04/2022      24/04/2022      None
10187  2       25/06/2022      24/08/2022      Low
] (delimiter is '\t');
```

```
JIRA_changes:
Inner Join (Changes)
Load
[Ticket ID] AS [Change ID],
[Source System]
inline
[
Ticket ID      Source System
10000  JIRA
10030  JIRA
10323  JIRA
10134  JIRA
10334  JIRA
10220  JIRA
20000  TFS
] (delimiter is '\t');
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `Source System`
- `Change ID`
- `Business Impact`

Now you can inspect the five resulting records. The resultant table from an `Inner Join` will only include records with matching information in both datasets.

Results table

Source System	Change ID	Business Impact
JIRA	10030	None
JIRA	10134	Low
JIRA	10220	None
JIRA	10323	High
JIRA	10334	Low

Example 3 – Outer join: Combining overlapping record sets

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset representing change records, which is loaded into a table named `Changes`.
- A second dataset representing change records originating from the source system `JIRA`, which is loaded and combined with the original records by joining it with an `Outer Join` load prefix.

This ensures that all the overlapping change records from both datasets are kept.

Load script

```
// 8 Change records

Changes:
Load * inline [
Change ID      Status ID       Scheduled Start Date    Scheduled End Date      Business Impact
10030  4       19/01/2022      23/02/2022      None
10015  3       04/01/2022      15/02/2022      Low
10138  2       07/05/2022      03/08/2022      None
10031  3       20/01/2022      25/03/2022      Low
10040  1       29/01/2022      22/04/2022      None
10134  1       03/05/2022      08/07/2022      Low
10334  2       19/11/2022      06/02/2023      Low
10220  2       28/07/2022      06/09/2022      None
] (delimiter is '\t');

// 6 Change records

JIRA_changes:
Outer Join (Changes)
Load
  [Ticket ID] AS [Change ID],
  [Source System]
inline
[
Ticket ID      Source System
10030  JIRA
10323  JIRA
10134  JIRA
10334  JIRA
10220  JIRA
10597  JIRA
] (delimiter is '\t');
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `Source System`
- `Change ID`
- `Business Impact`

Now you can inspect the 10 resulting records.

Results table

Source System	Change ID	Business Impact
JIRA	10030	None
JIRA	10134	Low
JIRA	10220	None
JIRA	10323	-
JIRA	10334	Low
JIRA	10597	-
-	10015	Low
-	10031	Low
-	10040	None
-	10138	None

Example 4 – Right join: Trimming down a target table by a secondary master dataset

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset representing change records, which is loaded into a table named `Changes`.
- A second dataset representing change records originating from the source system `Teamwork`.This is loaded and combined with the original records by joining it with a `Right Join` load prefix.

This ensures that only `Teamwork` change records are kept, while not losing any `Teamwork` records if the target table does not have a matching `Change ID`.

Load script

```
Changes:

Load * inline [
Change ID      Status ID       Scheduled Start Date    Scheduled End Date      Business Impact
10030  4       19/01/2022      23/02/2022      None
10015  3       04/01/2022      15/02/2022      Low
10103  1       02/04/2022      29/05/2022      Medium
10185  2       23/06/2022      08/09/2022      None
10323  1       08/11/2022      26/11/2022      High
10326  2       11/11/2022      05/12/2022      None
10138  2       07/05/2022      03/08/2022      None
10031  3       20/01/2022      25/03/2022      Low
10040  1       29/01/2022      22/04/2022      None
```

```
10134  1       03/05/2022      08/07/2022      Low
10334  2       19/11/2022      06/02/2023      Low
10220  2       28/07/2022      06/09/2022      None
10264  1       10/09/2022      17/10/2022      Medium
10116  1       15/04/2022      24/04/2022      None
10187  2       25/06/2022      24/08/2022      Low
] (delimiter is '\t');

Teamwork_changes:
Right Join (Changes)
Load
  [Ticket ID] AS [Change ID],
  [Source System]
inline
[
Ticket ID      Source System
10040  Teamwork
10015  Teamwork
10103  Teamwork
10031  Teamwork
50231  Teamwork
] (delimiter is '\t');
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `Source System`
- `Change ID`
- `Business Impact`

Now you can inspect the five resulting records.

Results table

Source System	Change ID	Business Impact
Teamwork	10015	Low
Teamwork	10031	Low
Teamwork	10040	None
Teamwork	10103	Medium
Teamwork	50231	-

Keep

The **keep** prefix is similar to the **join** prefix. Just as the **join** prefix, it compares the loaded table with an existing named table or the last previously created data table, but instead of joining the loaded table with an existing table, it has the effect of reducing one or both of the two tables before they are stored in Qlik Sense,

based on the intersection of table data. The comparison made is equivalent to a natural join made over all the common fields, i.e. the same way as in a corresponding join. However, the two tables are not joined and will be kept in Qlik Sense as two separately named tables.

Syntax:

```
(inner | left | right) keep [(tablename ) ]( loadstatement | selectstatement
)
```

The **keep** prefix must be preceded by one of the prefixes **inner**, **left** or **right**.

The explicit **join** prefix in Qlik Sense script language performs a full join of the two tables. The result is one table. In many cases such joins will result in very large tables. One of the main features of Qlik Sense is its ability to make associations between multiple tables instead of joining them, which greatly reduces memory usage, increases processing speed and offers enormous flexibility. Explicit joins should therefore generally be avoided in Qlik Sense scripts. The keep functionality was designed to reduce the number of cases where explicit joins needs to be used.

Arguments:

Arguments

Argument	Description
tablename	The named table to be compared to the loaded table.
loadstatement or selectstatement	The **LOAD** or **SELECT** statement for the loaded table.

Example:

```
Inner Keep LOAD * from abc.csv;
Left Keep SELECT * from table1;
tab1:
LOAD * from file1.csv;
tab2:
LOAD * from file2.csv;
.. .. ..
Left Keep (tab1) LOAD * from file3.csv;
```

Left

The **Join** and **Keep** prefixes can be preceded by the prefix **left**.

If used before **join** it specifies that a left join should be used. The resulting table will only contain combinations of field values from the raw data tables where the linking field values are represented in the first table. If used before **keep**, it specifies that the second raw data table should be reduced to its common intersection with the first table, before being stored in Qlik Sense.

Were you looking for the string function by the same name? See: Left (page 1007)

Syntax:

```
Left ( Join | Keep) [ (tablename) ](loadstatement | selectstatement)
```

Arguments:

Arguments

Argument	Description
tablename	The named table to be compared to the loaded table.
loadstatement or selectstatement	The **LOAD** or **SELECT** statement for the loaded table.

Example

Load script

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
Table1:
Load * inline [
Column1, Column2
A, B
1, aa
2, cc
3, ee ];

Table2:
Left Join Load * inline [
Column1, Column3
A, C
1, xx
4, yy ];
```

Result

Resulting table

Column1	Column2	Column3
A	B	C
1	aa	xx
2	cc	-
3	ee	-

Explanation

This example demonstrates the Left Join output where only values present in the first (left) table are joined.

Mapping

The **mapping** prefix is used to create a mapping table that can be used to, for example, replacing field values and field names during script execution.

Syntax:

```
Mapping( loadstatement | selectstatement )
```

The **mapping** prefix can be put in front of a **LOAD** or a **SELECT** statement and will store the result of the loading statement as a mapping table. Mapping provides an efficient way to substituting field values during script execution, e.g. replacing US, U.S. or America with USA. A mapping table consists of two columns, the first containing comparison values and the second containing the desired mapping values. Mapping tables are stored temporarily in memory and dropped automatically after script execution.

The content of the mapping table can be accessed using e.g. the **Map ... Using** statement, the **Rename Field** statement, the **Applymap()** function or the **Mapsubstring()** function.

Example:

In this example we load a list of salespersons with a country code representing their country of residence. We use a table mapping a country code to a country to replace the country code with the country name. Only three countries are defined in the mapping table, other country codes are mapped to 'Rest of the world'.

```
// Load mapping table of country codes:
map1:
mapping LOAD *
Inline [
CCode, Country
Sw, Sweden
Dk, Denmark
No, Norway
] ;
// Load list of salesmen, mapping country code to country
// If the country code is not in the mapping table, put Rest of the world
Salespersons:
LOAD *,
ApplyMap('map1', CCode,'Rest of the world') As Country
Inline [
CCode, Salesperson
Sw, John
Sw, Mary
Sw, Per
Dk, Preben
Dk, Olle
No, Ole
Sf, Ristu] ;
// We don't need the CCode anymore
Drop Field 'CCode';
```

The resulting table looks like this:

Mapping table

Salesperson	Country
John	Sweden
Mary	Sweden
Per	Sweden
Preben	Denmark
Olle	Denmark
Ole	Norway
Risttu	Rest of the world

Merge

The **Merge** prefix can be added to any **LOAD** or **SELECT** statement in the script to specify that the loaded table should be merged into another table. It also specifies that this statement should be run in a partial reload.

The typical use case is when you load a change log and want to use this to apply inserts, updates, and deletes to an existing table.

> *For partial reload to work properly, the app must be opened with data before a partial reload is triggered.*

Perform a partial reload using the **Reload** button. You can also use the Qlik Engine JSON API.

Syntax:

```
Merge [only] [(SequenceNoField [, SequenceNoVar])] On ListOfKeys [Concatenate
[(TableName)]] (loadstatement | selectstatement)
```

Arguments:

Arguments

Argument	Description
only	An optional qualifier denoting that the statement should be executed only during partial reloads. The statement is disregarded during normal (non-partial) reloads.
SequenceNoField	The name of the field containing a timestamp or a sequence number that defines the order of the operations.
SequenceNoVar	The name of the variable that gets assigned the maximum value for SequenceNoField of the table being merged.

Argument	Description
ListOfKeys	A comma separated list of field names specifying the primary key.
Operation	The first field of the load statement must contain the operation as a text string: 'Insert', 'Update', or 'Delete'. ‘i’, ‘u’ and ‘d’ are also accepted.

General functionality

During a normal (non-partial) reload, the **Merge LOAD** construction works as a normal **Load** statement but with the additional functionality of removing older obsolete records and records marked for deletion. The first field of the **Load** statement must hold information about the operation: Insert, Update, or Delete.

For each loaded record, the record identifier is compared with previously loaded records, and only the latest record (according to the sequence number) will be kept. If the latest record is marked with Delete, none will be kept.

Target table

Which table to modify is determined by the set of fields. If a table with the same set of fields (except the first field; the operation) already exists, this will be the relevant table to modify. Alternatively, a **Concatenate** prefix can be used to specify the table. If the target table is not determined, the result of the **Merge LOAD** construction is stored in a new table.

If the Concatenate prefix is used, the resulting table has a set of fields corresponding to the union of the existing table and the input to the merge. Hence, the target table may get more fields than the change log that is used as input to the merge.

A partial reload does the same as a full reload. One difference is that a partial reload rarely creates a new table. Unless you have used the **Only** clause, a target table with the same set of fields from the previous script execution always exists.

Sequence number

If the loaded change log is an accumulated log, that is, it contains changes that already have been loaded, the parameter SequenceNoVar can be used in a **Where** clause to limit the amount of input data. The **Merge LOAD** could then be made to only load records where the field SequenceNoField is greater than SequenceNoVar. Upon completion, the **Merge LOAD** assigns a new value to the SequenceNoVar with the maximum value seen in the SequenceNoField field.

Operations

The **Merge LOAD** can have fewer fields than the target table. The different operations treat missing fields differently:

Insert: Fields missing in the **Merge LOAD**, but existing in the target table, get a NULL in the target table.

Delete: Missing fields do not affect the result. The relevant records are deleted anyway.

Update: Fields listed in the **Merge LOAD** are updated in the target table. Missing fields are not changed. This means that the two following statements are not identical:

- Merge on Key Concatenate Load 'U' as Operation, Key, F1, Null() as F2 From ...;
- Merge on Key Concatenate Load 'U' as Operation, Key, F1 From ...;

The first statement updates the listed records and changes F2 to NULL. The second does not change F2, but instead, leaves the values in the target table.

Examples

Example 1: Simple merge with specified table

In this example, an inline table named Persons is loaded with three rows. **Merge** then changes the table as follows:

- Adds the row, *Mary, 4* .
- Deletes the row, *Steven, 3*.
- Assigns the number *5* to *Jake* .

The *LastChangeDate* variable is set to the maximum value in the *ChangeDate* column after **Merge** is executed.

Load script

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
Set DateFormat='D/M/YYYY';
Persons:
load * inline [
Name, Number
Jake, 3
Jill, 2
Steven, 3
];

Merge (ChangeDate, LastChangeDate) on Name Concatenate(Persons)
LOAD * inline [
Operation, ChangeDate,   Name,     Number
Insert,    1/1/2021,     Mary,     4
Delete,    1/1/2021,     Steven,
Update,    2/1/2021,     Jake,     5
];
```

Result

Prior to the **Merge Load**, the resulting table appears as follows:

Resulting table

Name	Number
Jake	3
Jill	2
Steven	3

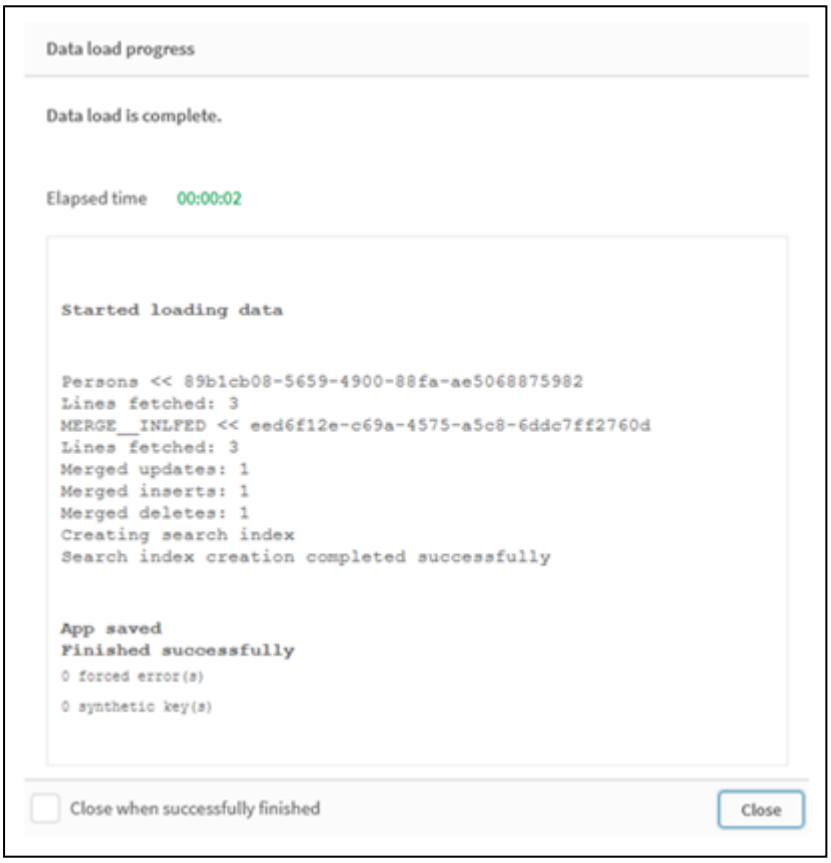
Following the **Merge Load**, the table appears as follows:

Resulting table

ChangeDate	Name	Number
2/1/2021	Jake	5
-	Jill	2
1/1/2021	Mary	4

When the data is loaded, the **Data load progress** dialog box shows the operations that are performed:

Data load progress dialog box



Example 2: Data load script with missing fields

In this example, the same data as above is loaded, but now with an ID for each person.

Merge changes the table as follows:

- Adds the row, *Mary, 4*.
- Deletes the row, *Steven, 3*.
- Assigns the number *5* to *Jake*.
- Assigns the number *6* to *Jill*.

Load script

Here we use two **Merge Load** statements, one for 'Insert' and 'Delete', and a second one for the 'Update'.

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
Set DateFormat='D/M/YYYY';
Persons:
Load * Inline [
PersonID, Name, Number
1, Jake, 3
2, Jill, 2
3, Steven, 3
];

Merge (ChangeDate, LastChangeDate) on PersonID Concatenate(Persons)
Load * Inline [
Operation, ChangeDate,   PersonID, Name,     Number
Insert,    1/1/2021,     4,        Mary,     4
Delete,    1/1/2021,     3,        Steven,
];

Merge (ChangeDate, LastChangeDate) on PersonID Concatenate(Persons)
Load * Inline [
Operation, ChangeDate,   PersonID, Number
Update,    2/1/2021,     1,        5
Update,    3/1/2021,     2,        6
];
```

Result

Following the **Merge Load** statements, the table appears as follows:

Resulting table

PersonID	ChangeDate	Name	Number
1	2/1/2021	Jake	5
2	3/1/2021	Jill	6
4	1/1/2021	Mary	4

Note that the second **Merge** statement does not include the field **Name**, and as a consequence, the names have not been changed.

Example 3: Data load script - Partial reload using a Where-clause with ChangeDate

In the following example, the **Only** argument specifies that the **Merge** command is only executed during a partial reload. Updates are filtered based on the previously captured LastChangeDate. After **Merge** is finished, LastChangeDate variable is assigned the maximum value of the ChangeDate column processed during the merge.

Load script

```
Merge Only (ChangeDate, LastChangeDate) on Name Concatenate(Persons)
LOAD Operation, ChangeDate,  Name, Number
from [lib://ChangeFilesFolder/BulkChangesInPersonsTable.csv] (txt)
where ChangeDate >= $(LastChangeDate);
```

NoConcatenate

The **NoConcatenate** prefix forces two loaded tables with identical field sets to be treated as two separate internal tables, when they would otherwise be automatically concatenated.

Syntax:

```
NoConcatenate( loadstatement | selectstatement )
```

Example:

```
LOAD A,B from file1.csv;
NoConcatenate LOAD A,B from file2.csv;
```

Only

The **Only** script keyword is used as an aggregation function, or as part of the syntax in partial reload prefixes **Add**, **Replace**, and **Merge**.

Outer

The explicit **Join** prefix can be preceded by the prefix **Outer** to specify an outer join. In an outer join, all combinations between the two tables are generated. The resulting table will thus contain combinations of field values from the raw data tables where the linking field values are represented in one or both tables. The **Outer** keyword is optional and is the default join type used when a join prefix is not specified.

Syntax:

```
Outer Join [ (tablename) ](loadstatement |selectstatement )
```

Arguments:

Arguments

Argument	Description
tablename	The named table to be compared to the loaded table.
loadstatement or selectstatement	The **LOAD** or **SELECT** statement for the loaded table.

Example

Load script

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
Table1:
Load * inline [
Column1, Column2
A, B
1, aa
2, cc
3, ee ];

Table2:
Outer Join Load * inline [
Column1, Column3
A, C
1, xx
4, yy ];
```

Resulting table

Column1	Column2	Column3
A	B	C
1	aa	xx
2	cc	-
3	ee	-
4	-	yy

Explanation

In this example, the two tables, Table1 and Table2, are merged into a single table labeled Table1. In cases like this, the **outer** prefix is often used to join several tables into a single table to perform aggregations over the values of a single table.

Partial reload

A full reload always starts by deleting all tables in the existing data model, and then runs the load script.

A partial reload will not do this. Instead it keeps all tables in the data model and then executes only **Load** and **Select** statements preceded by an **Add**, **Merge**, or **Replace** prefix. Other data tables are not affected by the command. The **only** argument denotes that the statement should be executed only during partial reloads, and should be disregarded during full reloads. The following table summarizes statement execution for partial and full reloads.

Statement	Full reload	Partial reload
Load ...	Statement will run	Statement will not run
Add/Replace/Merge Load ...	Statement will run	Statement will run
Add/Replace/Merge Only Load ...	Statement will not run	Statement will run

Partial reloads have several benefits compared to full reloads:

- Faster, because only data recently changed needs to be loaded. With large data sets the difference is significant.
- Less memory is consumed, because less data is loaded.
- More reliable, because queries to source data run faster, reducing the risk of network problems.

For partial reload to work properly, the app must be opened with data before a partial reload is triggered.

Perform a partial reload using the **Reload** button. You can also use the Qlik Engine JSON API.

Limitation

A partial reload can remove values from the data. However, this will not be reflected in the list of distinct values, which is a table maintained internally. So, after a partial reload, the list will contain all distinct values that have existed in the field since the last full reload, which may be more than what currently exists after the partial reload. This affects the output of the FieldValueCount() and the FieldValue() functions. The FieldValueCount() could potentially return a number greater than the current number of field values.

Example

Example 1

Load script

Add the example script to your app and do a partial reload. To see the result, add the fields listed in the results column to a sheet in your app.

```
T1:
Add only Load distinct recno()+10 as Num autogenerate 10;
```

Result

Resulting table

Num	Count(Num)
11	1
12	1
13	1
14	1
15	1
16	1
17	1
18	1

Num	Count(Num)
19	1
20	1

Explanation

The statement is only executed during a partial reload. If the "distinct" prefix is omitted, the count of the **Num** field will increase with each subsequent partial reload.

Example 2

Load script

Add the example script to your app. Do a full reload and view the result. Next, do a partial reload and view the result. To see the results, add the fields listed in the results column to a sheet in your app.

```
T1:
Load recno() as ID, recno() as Value autogenerate 10;

T1:
Replace only Load recno() as ID, repeat(recno(),3) as Value autogenerate 10;
```

Result

Output table after full reload

ID	Value
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

Output table after partial reload

ID	Value
1	111

ID	Value
2	222
3	333
4	444
5	555
6	666
7	777
8	888
9	999
10	101010

Explanation

The first table is loaded during a full reload and the second table simply replaces the first table during a partial reload.

Replace

The **Replace** script keyword is used as a string function, or as a prefix in partial reload.

Replace

The **Replace** prefix can be added to any **LOAD** or **SELECT** statement in the script to specify that the loaded table should replace another table. It also specifies that this statement should be run in a partial reload. The **Replace** prefix can also be used in a **Map** statement.

> *For partial reload to work properly, the app must be opened with data before a partial reload is triggered.*

Perform a partial reload using the **Reload** button. You can also use the Qlik Engine JSON API.

Syntax:

```
Replace [only] [Concatenate[(tablename)]] (loadstatement | selectstatement)
```

```
Replace [only] mapstatement
```

During a normal (non-partial) reload, the **Replace LOAD** construction will work as a normal **LOAD** statement but be preceded by a **Drop Table**. First the old table will be dropped, then records will be generated and stored as a new table.

If the **Concatenate** prefix is used, or if there exists a table with the same set of fields, this will be the relevant table to drop. Otherwise, there is no table to drop and the **Replace LOAD** construction will be identical to a normal **LOAD**.

A partial reload will do the same. The only difference is that there is always a table from the previous script execution to drop. The **Replace LOAD** construction will always first drop the old table, then create a new one.

The **Replace Map...Using** statement causes mapping to take place also during partial script execution.

Arguments:

Arguments

Argument	Description
only	An optional qualifier denoting that the statement should be executed only during partial reloads. It should be disregarded during normal (non-partial) reloads.

Examples and results:

Example	Result

`Tab1:`
`Replace LOAD *`
`from File1.csv;`
	During both normal and partial reload, the Qlik Sense table Tab1 is initially dropped. Thereafter new data is loaded from File1.csv and stored in Tab1.

`Tab1:`
`Replace only`
`LOAD * from`
`File1.csv;`
	During normal reload, this statement is disregarded.
	
	During partial reload, any Qlik Sense table previously named Tab1 is initially dropped. Thereafter new data is loaded from File1.csv and stored in Tab1.

`Tab1:`
`LOAD a,b,c from`
`File1.csv;`
`Replace LOAD`
`a,b,c from`
`File2.csv;`
	During normal reload, the file File1.csv is first read into the Qlik Sense table Tab1, but then immediately dropped and replaced by new data loaded from File2.csv. All data from File1.csv is lost.
	
	During partial reload, the entire Qlik Sense table Tab1 is initially dropped. Thereafter it is replaced by new data loaded from File2.csv.

`Tab1:`
`LOAD a,b,c from`
`File1.csv;`
`Replace only`
`LOAD a,b,c from`
`File2.csv;`
	During normal reload, data is loaded from File1.csv and stored in the Qlik Sense table Tab1. File2.csv is disregarded.
	
	During partial reload, the entire Qlik Sense table Tab1 is initially dropped. Thereafter it is replaced by new data loaded from File2.csv. All data from File1.csv is lost.

Right

The **Join** and **Keep** prefixes can be preceded by the prefix **right**.

If used before **join** it specifies that a right join should be used. The resulting table will only contain combinations of field values from the raw data tables where the linking field values are represented in the second table. If used before **keep**, it specifies that the first raw data table should be reduced to its common intersection with the second table, before being stored in Qlik Sense.

Were you looking for the string function by the same name? See: Right (page 1017)

Syntax:

```
Right (Join | Keep) [(tablename)](loadstatement |selectstatement )
```

Arguments:

Arguments

Argument	Description
tablename	The named table to be compared to the loaded table.
loadstatement or selectstatement	The **LOAD** or **SELECT** statement for the loaded table.

Example

Load script

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
Table1:
Load * inline [
Column1, Column2
A, B
1, aa
2, cc
3, ee ];

Table2:
Right Join Load * inline [
Column1, Column3
A, C
1, xx
4, yy ];
```

Result

Resulting table

Column1	Column2	Column3
A	B	C
1	aa	xx
4	-	yy

Explanation

This example demonstrates the Right Join output where only values present in the second (right) table are joined.

Sample

The **sample** prefix to a **LOAD** or **SELECT** statement is used for loading a random sample of records from the data source.

Syntax:

```
Sample  p ( loadstatement | selectstatement )
```

Arguments:

Arguments

Argument	Description
p	An arbitrary expression which valuates to a number larger than 0 and lower or equal to 1. The number indicates the probability for a given record to be read. All records will be read but only some of them will be loaded into Qlik Sense.

Example:

```
Sample 0.15 SQL SELECT * from Longtable;
Sample(0.15) LOAD * from Longtab.csv;
```

The parentheses are allowed but not required.

Semantic

The `semantic` load prefix creates a special type of field that can be used in Qlik Sense to connect and manage relational data, such as tree structures, self-referencing parent-child structured data and/or data that can be described as a graph.

Note that the `semantic` load can function similarly to the *Hierarchy (page 56)* and *HierarchyBelongsTo (page 58)* prefixes. All three prefixes can be used as building blocks in effective front-end solutions for traversing relational data.

Syntax:

```
Semantic( loadstatement | selectstatement)
```

A semantic load expects an input that is exactly three or four fields wide with a strict definition of what each ordered field represents, as shown in the table below:

Semantic load fields

Field name	Field description
1st Field:	This tag is a representation of the first of two objects between which there is a relationship.

Field name	Field description
2nd Field:	This tag will be used to describe the “forward” relationship between the first and second object. If the first object is a child and the second object is a parent, you can create a relationship tab that states “parent” or “parent of” as if you are following the relationship from child to parent.
3rd Field:	This tag is a representation of the second of two objects between which there is a relationship.
4th Field:	This field is optional. This tag describes the “backward” or “inverse” relationship between the first and second object. If the first object is a child and the second object is a parent, a relationship tab could state “child” or “child of” as if you are following the relationship from parent to child. If you do not add a fourth field, then the second field tag will be used to describe the relationship in either direction. In that case, an arrow symbol is automatically added as part of the tag.

The following code is an example of the `semantic` prefix.

```
Semantic
Load
Object,
‘Parent’ AS Relationship,
NeighbouringObject AS Object,
‘Child’ AS Relationship
from graphdata.csv;
```

It is allowed and typical practice to label the third field the same as the first field. This creates a self-referencing lookup, so that you can follow object(s) to the related object(s) one relationship step away at a time. If the 3rd field does not carry the same name, then the end result will be a simple lookup from an object(s) to its direct relational neighbor(s) one step away only, which is an output of little practical use.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Related functions

Functions	Interaction
Hierarchy (page 56)	The Hierarchy load prefix is used to divide and organize nodes in parent-child and other graph-like data structures and transform them into tables.
HierarchyBelongsTo (page 58)	The HierarchyBelongsTo load prefix is used to locate and organize the ancestors of parent-child and other graph-like data structures and transform them into tables.

Example - Creating a special field for connecting relationships using the semantic prefix

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset representing geography relation records which is loaded into a table named `GeographyTree`.
    - Each entry has an ID at the beginning of the line and a ParentID at the end of the line.
- The `semantic` prefix which will add one special behavior field labeled, `Relation`.

Load script

```
GeographyTree:
LOAD
    ID,
    Geography,
    if(ParentID='',null(),ParentID) AS ParentID

INLINE [
ID,Geography,ParentID
1,World
2,Europe,1
3,Asia,1
4,North America,1
5,South America,1
6,UK,2
7,Germany,2
8,Sweden,2
9,South Korea,3
10,North Korea,3
11,China,3
12,London,6
13,Birmingham,6
];

SemanticTable:
Semantic Load
```

```
        ID as ID,
        'Parent' as Relation,
        ParentID as ID,
        'Child' as Relation
resident GeographyTree;
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `Id`
- `Geography`

Then, create a filter pane with `Relation` as a dimension. Click **Done editing**.

Results table

Id	Geography
1	World
2	Europe
3	Asia
4	North America
5	South America
6	UK
7	Germany
8	Sweden
9	South Korea
10	North Korea
11	China
12	London
13	Birmingham

Filter pane

Relation
Child
Parent

Click **Europe** from the `Geography` dimension in the table and click **Child** from the `Relation` dimension in the filter pane. Note the expected result in the table:

Results table showing "children" of Europe

Id	Geography
6	UK
7	Germany
8	Sweden

Clicking **Child** again will show places that are "children" of the UK, one step further down.

Results table showing "children" of UK

Id	Geography
12	London
13	Birmingham

Unless

The **unless** prefix and suffix is used for creating a conditional clause which determines whether a statement or exit clause should be evaluated or not. It may be seen as a compact alternative to the full **if..end if** statement.

Syntax:

```
(Unless condition statement | exitstatement Unless condition )
```

The **statement** or the **exitstatement** will only be executed if **condition** is evaluated to False.

The **unless** prefix may be used on statements which already have one or several other statements, including additional **when** or **unless** prefixes.

Arguments:

Arguments

Argument	Description
condition	A logical expression evaluating to True or False.
statement	Any Qlik Sense script statement except control statements.
exitstatement	An **exit for**, **exit do** or **exit sub** clause or an **exit script** statement.

Examples:

```
exit script unless A=1;
unless A=1 LOAD * from myfile.csv;
unless A=1 when B=2 drop table Tab1;
```

When

The **when** prefix and suffix is used for creating a conditional clause which determines whether a statement or exit clause should be executed or not. It may be seen as a compact alternative to the full **if..end if** statement.

Syntax:

```
(when condition statement | exitstatement when condition )
```

The **statement** or the **exitstatement** will only be executed if condition is evaluated to True.

The **when** prefix may be used on statements which already have one or several other statements, including additional **when** or **unless** prefixes.

Arguments

Argument	Description
condition	A logical expression evaluating to True or False.
statement	Any Qlik Sense script statement except control statements.
exitstatement	An **exit for**, **exit do** or **exit sub** clause or an **exit script** statement.

Example 1:

```
exit script when A=1;
```

Example 2:

```
when A=1 LOAD * from myfile.csv;
```

Example 3:

```
when A=1 unless B=2 drop table Tab1;
```

2.5 Script regular statements

Regular statements are typically used for manipulating data in one way or another. These statements may be written over any number of lines in the script and must always be terminated by a semicolon, ";".

All script keywords can be typed with any combination of lower case and upper case characters. Field and variable names used in the statements are however case sensitive.

Script regular statements overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Alias

The **alias** statement is used for setting an alias according to which a field will be renamed whenever it occurs in the script that follows.

```
Alias fieldname as aliasname {,fieldname as aliasname}
```

Autonumber

This statement creates a unique integer value for each distinct evaluated value in a field encountered during the script execution.

```
AutoNumber fields [Using namespace] ]
```

Binary

The **binary** statement is used for loading the data from another QlikView document, including section access data.

```
Binary  [path] filename
```

comment

Provides a way of displaying the field comments (metadata) from databases and spreadsheets. Field names not present in the app will be ignored. If multiple occurrences of a field name are found, the last value is used.

```
Comment field *fieldlist using mapname
Comment field fieldname with comment
```

comment table

Provides a way of displaying the table comments (metadata) from databases or spreadsheets.

```
Comment table tablelist using mapname
Comment table tablename with comment
```

Connect

This functionality is not available in Qlik Sense SaaS.

The **CONNECT** statement is used to define Qlik Sense access to a general database through the OLE DB/ODBC interface. For ODBC, the data source first needs to be specified using the ODBC administrator.

```
ODBC Connect TO connect-string [ ( access_info ) ]
OLEDB CONNECT TO connect-string [ ( access_info ) ]
CUSTOM CONNECT TO connect-string [ ( access_info ) ]
LIB CONNECT TO connection
```

Declare

The **Declare** statement is used to create field definitions, where you can define relations between fields or functions. A set of field definitions can be used to automatically generate derived fields, which can be used as dimensions. For example, you can create a calendar definition, and use that to generate related dimensions, such as year, month, week and day, from a date field.

```
definition_name:
```

```
Declare [Field[s]] Definition [Tagged tag_list ]
[Parameters parameter_list ]
Fields field_list
[Groups group_list ]

<definition name>:
Declare [Field][s] Definition
Using <existing_definition>
[With <parameter_assignment> ]
```

Derive

The **Derive** statement is used to generate derived fields based on a field definition created with a **Declare** statement. You can either specify which data fields to derive fields for, or derive them explicitly or implicitly based on field tags.

```
Derive [Field[s]] From [Field[s]] field_list Using definition
Derive [Field[s]] From Explicit [Tag[s]] (tag_list) Using definition
Derive [Field[s]] From Implicit [Tag[s]] Using definition
```

Directory

The **Directory** statement defines which directory to look in for data files in subsequent **LOAD** statements, until a new **Directory** statement is made.

```
Directory [path]
```

Disconnect

The **Disconnect** statement terminates the current ODBC/OLE DB/Custom connection. This statement is optional.

```
Disconnect
```

drop field

One or several Qlik Sense fields can be dropped from the data model, and thus from memory, at any time during script execution, by means of a **drop field** statement.

> *Both **drop field** and **drop fields** are allowed forms with no difference in effect. If no table is specified, the field will be dropped from all tables where it occurs.*

```
Drop field  fieldname [ , fieldname2 ...] [from tablename1 [ , tablename2
...]]
drop fields fieldname [ , fieldname2 ...] [from tablename1 [ , tablename2
...]]
```

drop table

One or several Qlik Sense internal tables can be dropped from the data model, and thus from memory, at any time during script execution, by means of a **drop table** statement.

*The forms **drop table** and **drop tables** are both accepted.*

```
Drop table  tablename [, tablename2 ...]
drop tables[ tablename [, tablename2 ...]
```

Execute

The **Execute** statement is used to run other programs while Qlik Sense is loading data. For example, to make conversions that are necessary.

```
Execute commandline
```

FlushLog

The **FlushLog** statement forces Qlik Sense to write the content of the script buffer to the script log file.

```
FlushLog
```

Force

The **force** statement forces Qlik Sense to interpret field names and field values of subsequent **LOAD** and **SELECT** statements as written with only upper case letters, with only lower case letters, as always capitalized or as they appear (mixed). This statement makes it possible to associate field values from tables made according to different conventions.

```
Force ( capitalization | case upper | case lower | case mixed )
```

LOAD

The **LOAD** statement loads fields from a file, from data defined in the script, from a previously loaded table, from a web page, from the result of a subsequent **SELECT** statement or by generating data automatically. It is also possible to load data from analytic connections.

```
 Load [ distinct ] *fieldlist
[( from file [ format-spec ] |
from_field fieldassource [format-spec]
inline data [ format-spec ] |
resident table-label |
autogenerate size )]
[ where criterion | while criterion ]
[ group_by groupbyfieldlist ]
[order_by orderbyfieldlist ]
[extension pluginname.functionname(tabledescription)]
```

Let

The **let** statement is a complement to the **set** statement, used for defining script variables. The **let** statement, in opposition to the **set** statement, evaluates the expression on the right side of the ' =' at script run time before it is assigned to the variable.

```
Let variablename=expression
```

Loosen Table

One or more Qlik Sense internal data tables can be explicitly declared loosely coupled during script execution by using a **Loosen Table** statement. When a table is loosely coupled, all associations between field values in the table are removed. A similar effect could be achieved by loading each field of the loosely coupled table as independent, unconnected tables. Loosely coupled can be useful during testing to temporarily isolate different parts of the data structure. A loosely coupled table can be identified in the table viewer by the dotted lines. The use of one or more **Loosen Table** statements in the script will make Qlik Sense disregard any setting of tables as loosely coupled made before the script execution.

```
tablename [ , tablename2 ...]
Loosen Tables tablename [ , tablename2 ...]
```

Map ... using

The **map ... using** statement is used for mapping a certain field value or expression to the values of a specific mapping table. The mapping table is created through the **Mapping** statement.

```
Map  *fieldlist Using  mapname
```

NullAsNull

The **NullAsNull** statement turns off the conversion of NULL values to string values previously set by a **NullAsValue** statement.

```
NullAsNull *fieldlist
```

NullAsValue

The **NullAsValue** statement specifies for which fields that NULL should be converted to a value.

```
NullAsValue *fieldlist
```

Qualify

The **Qualify** statement is used for switching on the qualification of field names, i.e. field names will get the table name as a prefix.

```
Qualify *fieldlist
```

Rem

The **rem** statement is used for inserting remarks, or comments, into the script, or to temporarily deactivate script statements without removing them.

```
Rem string
```

Rename Field

This script function renames one or more existing Qlik Sense field(s) after they have been loaded.

```
Rename field (using mapname | oldname to newname{ , oldname to newname })
```

```
Rename Fields (using mapname | oldname to newname{ , oldname to newname })
```

Rename Table

This script function renames one or more existing Qlik Sense internal table(s) after they have been loaded.

```
Rename table  (using mapname | oldname to newname{ , oldname to newname })
Rename Tables (using mapname | oldname to newname{ , oldname to newname })
```

Section

With the **section** statement, it is possible to define whether the subsequent **LOAD** and **SELECT** statements should be considered as data or as a definition of the access rights.

```
Section (access | application)
```

Select

The selection of fields from an ODBC data source or OLE DB provider is made through standard SQL **SELECT** statements. However, whether the **SELECT** statements are accepted depends on the ODBC driver or OLE DB provider used.

```
Select [all | distinct | distinctrow | top n [percent] ] *fieldlist

From tablelist

[Where criterion ]

[Group by fieldlist [having criterion ] ]

[Order by fieldlist [asc | desc] ]

[ (Inner | Left | Right | Full)Join tablename on fieldref = fieldref ]
```

Set

The **set** statement is used for defining script variables. These can be used for substituting strings, paths, drives, and so on.

```
Set variablename=string
```

Sleep

The **sleep** statement pauses script execution for a specified time.

```
Sleep n
```

SQL

The **SQL** statement allows you to send an arbitrary SQL command through an ODBC or OLE DB connection.

```
SQL sql_command
```

SQLColumns

The **sqlcolumns** statement returns a set of fields describing the columns of an ODBC or OLE DB data source, to which a **connect** has been made.

```
SQLColumns
```

SQLTables

The **sqltables** statement returns a set of fields describing the tables of an ODBC or OLE DB data source, to which a **connect** has been made.

```
SQLTables
```

SQLTypes

The **sqltypes** statement returns a set of fields describing the types of an ODBC or OLE DB data source, to which a **connect** has been made.

```
SQLTypes
```

Star

The string used for representing the set of all the values of a field in the database can be set through the **star** statement. It affects the subsequent **LOAD** and **SELECT** statements.

```
Star is [ string ]
```

Store

The **Store** statement creates a QVD, CSV, or text file.

```
Store [ *fieldlist from] table into filename [ format-spec ];
```

Tag

This script statement provides a way to assign tags to one or more fields or tables. If an attempt to tag a field or table not present in the app is made, the tagging will be ignored. If conflicting occurrences of a field or tag name are found, the last value is used.

```
Tag[field|fields] fieldlist with tagname
Tag [field|fields] fieldlist using mapname
Tag table tablelist with tagname
```

Trace

The **trace** statement writes a string to the **Script Execution Progress** window and to the script log file, when used. It is very useful for debugging purposes. Using $-expansions of variables that are calculated prior to the **trace** statement, you can customize the message.

```
Trace string
```

Unmap

The **Unmap** statement disables field value mapping specified by a previous **Map ... Using** statement for subsequently loaded fields.

```
Unmap *fieldlist
```

Unqualify

The **Unqualify** statement is used for switching off the qualification of field names that has been previously switched on by the **Qualify** statement.

```
Unqualify *fieldlist
```

Untag

This script statement provides a way to remove tags from fields or tables. If an attempt to untag a field or table not present in the app is made, the untagging will be ignored.

```
Untag[field|fields] fieldlist with tagname
Tag [field|fields] fieldlist using mapname
```

```
Tag table tablelist with tagname
```

Alias

The **alias** statement is used for setting an alias according to which a field will be renamed whenever it occurs in the script that follows.

Syntax:

```
alias fieldname as aliasname {,fieldname as aliasname}
```

Arguments:

Arguments

Argument	Description
fieldname	The name of the field in your source data
aliasname	An alias name you want to use instead

Examples and results:

Example	Result
`Alias ID_N as NameID;`	
`Alias A as Name, B as Number, C as Date;`	The name changes defined through this statement are used on all subsequent **SELECT** and **LOAD** statements. A new alias can be defined for a field name by a new **alias** statement at any subsequent position in the script.

AutoNumber

This statement creates a unique integer value for each distinct evaluated value in a field encountered during the script execution.

You can also use the *autonumber (page 490)* function inside a **LOAD** statement, but this has some limitations when you want to use an optimized load. You can create an optimized load by loading the data from a **QVD** file first, and then using the **AutoNumber** statement to convert values to symbol keys.

Syntax:

```
AutoNumber *fieldlist [Using namespace] ]
```

Arguments:

Arguments

Argument	Description
*fieldlist	A comma-separated list of the fields where the values should be replaced by a unique integer value. You can use wildcard characters ? and * in the field names to include all fields with matching names. You can also use * to include all fields. You need to quote field names when wildcards are used.
namespace	**`Using`** `namespace` is optional. You can use this option if you want to create a namespace, where identical values in different fields share the same key. If you do not use this option, all fields will have a separate key index.

Limitations:

When you have several **LOAD** statements in the script, you need to place the **AutoNumber** statement after the final **LOAD** statement.

Example - script with AutoNumber

Script example

In this example, the data is first loaded without the **AutoNumber** statement. The **AutoNumber** statement is then added to show the effect.

Data used in the example

Load the following data as an inline load in the data load editor to create the script example below. Leave the **AutoNumber** statement commented out for now.

```
RegionSales:
LOAD *,
Region &'|'& Year &'|'& Month as KeyToOtherTable
INLINE
[ Region, Year, Month, Sales
North, 2014,   May,    245
North, 2014,   May,    347
North, 2014,   June,   127
South, 2014,   June,   645
South, 2013,   May,    367
South, 2013,   May,    221
];

Budget:
LOAD Budget,
Region &'|'& Year &'|'& Month as KeyToOtherTable
INLINE
[Region, Year, Month, Budget
```

```
North,  2014,   May,    200
North,  2014,   May,    350
North,  2014,   June,   150
South,  2014,   June,   500
South,  2013,   May,    300
South,  2013,   May,    200
];

//AutoNumber KeyToOtherTable;
```

Create visualizations

Create two table visualizations in a Qlik Sense sheet. Add **KeyToOtherTable**, **Region**, **Year**, **Month**, and **Sales** as dimensions to the first table. Add **KeyToOtherTable**, **Region**, **Year**, **Month**, and **Budget** as dimensions to the second table.

Result

RegionSales table

KeyToOtherTable	Region	Year	Month	Sales
North|2014|June	North	2014	June	127
North|2014|May	North	2014	May	245
North|2014|May	North	2014	May	347
South|2013|May	South	2013	May	221
South|2013|May	South	2013	May	367
South|2014|June	South	2014	June	645

Budget table

KeyToOtherTable	Region	Year	Month	Budget
North|2014|June	North	2014	June	150
North|2014|May	North	2014	May	200
North|2014|May	North	2014	May	350
South|2013|May	South	2013	May	200
South|2013|May	South	2013	May	300
South|2014|June	South	2014	June	500

Explanation

The example shows a composite field **KeyToOtherTable** that links the two tables. **AutoNumber** is not used. Note the length of the **KeyToOtherTable** values.

Add AutoNumber statement

Uncomment the **AutoNumber** statement in the load script.

```
AutoNumber KeyToOtherTable;
```

Result

RegionSales table

KeyToOtherTable	Region	Year	Month	Sales
1	North	2014	June	127
1	North	2014	May	245
2	North	2014	May	347
3	South	2013	May	221
4	South	2013	May	367
4	South	2014	June	645

Budget table

KeyToOtherTable	Region	Year	Month	Budget
1	North	2014	June	150
1	North	2014	May	200
2	North	2014	May	350
3	South	2013	May	200
4	South	2013	May	300
4	South	2014	June	500

Explanation

The **KeyToOtherTable** field values have been replaced with unique integer values and, as a result, the length of the field values has been reduced, thus conserving memory. The key fields in both tables are affected by **AutoNumber** and the tables remain linked. The example is brief for demonstration purposes, but would be meaningful with a table containing a large number of rows.

Binary

The **binary** statement is used for loading the data from another Qlik Sense app or QlikView document, including section access data. Other elements of the app are not included, for example, sheets, stories, visualizations, master items or variables.

Only one **binary** statement is allowed in the script. The **binary** statement must be the first statement of the script, even before the SET statements usually located at the beginning of the script.

Syntax:

```
binary [path] filename
```

Arguments:

Arguments

Argument	Description
path	The path to the file which should be a reference to a folder data connection. This is required if the file is not located in the Qlik Sense working directory. **Example: *'lib://Table Files/'*** In legacy scripting mode, the following path formats are also supported: • absolute **Example: *c:\data*** • relative to the app containing this script line. **Example: *data***
filename	The name of the file, including the file extension .qvw or .qvf.

Limitations:

You cannot use **binary** to load data from an app on the same Qlik Sense Enterprise deployment by referring to the app ID. You can only load from a *.qvf* file.

Examples

String	Description
`Binary lib://DataFolder/customer.qvw;`	In this example, the file must be in located in the **Folder** data connection. This may be, for example, a folder that your administrator creates on the Qlik Sense server. Click **Create new connection** in the data load editor and then select **Folder** under **File locations**.
`Binary customer.qvf;`	In this example, the file must be in located in the Qlik Sense working directory.
`Binary c:\qv\customer.qvw;`	This example using an absolute file path will only work in legacy scripting mode.

Comment field

Provides a way of displaying the field comments (metadata) from databases and spreadsheets. Field names not present in the app will be ignored. If multiple occurrences of a field name are found, the last value is used.

Syntax:

```
comment [fields] *fieldlist using mapname
comment [field] fieldname with comment
```

The map table used should have two columns, the first containing field names and the second the comments.

Arguments:

Arguments

Argument	Description
**fieldlist*	A comma separated list of the fields to be commented. Using * as field list indicates all fields. The wildcard characters * and ? are allowed in field names. Quoting of field names may be necessary when wildcards are used.
mapname	The name of a mapping table previously read in a mapping **LOAD** or mapping **SELECT** statement.
fieldname	The name of the field that should be commented.
comment	The comment that should be added to the field.

Example 1:

```
commentmap:
mapping LOAD * inline [
a,b
Alpha,This field contains text values
Num,This field contains numeric values
];
comment fields using commentmap;
```

Example 2:

```
comment field Alpha with AFieldContainingCharacters;
comment field Num with '*A field containing numbers';
comment Gamma with 'Mickey Mouse field';
```

Comment table

Provides a way of displaying the table comments (metadata) from databases or spreadsheets.

Table names not present in the app are ignored. If multiple occurrences of a table name are found, the last value is used. The keyword can be used to read comments from a data source.

Syntax:

```
comment [tables] tablelist using mapname
comment [table] tablename with comment
```

Arguments:

Arguments

Argument	Description
tablelist	(table{,table})
mapname	The name of a mapping table previously read in a mapping **LOAD** or mapping **SELECT** statement.
tablename	The name of the table that should be commented.
comment	The comment that should be added to the table.

Example 1:

```
Commentmap:
mapping LOAD * inline [
a,b
Main,This is the fact table
Currencies, Currency helper table
];
comment tables using Commentmap;
```

Example 2:

```
comment table Main with 'Main fact table';
```

Connect

The **CONNECT** statement is used to define Qlik Sense access to a general database through the OLE DB/ODBC interface. For ODBC, the data source first needs to be specified using the ODBC administrator.

This functionality is not available in Qlik Sense SaaS.

This statement supports only folder data connections in standard mode.

Syntax:

```
ODBC CONNECT TO connect-string
OLEDB CONNECT TO connect-string
CUSTOM CONNECT TO connect-string
LIB CONNECT TO connection
```

Arguments:

Arguments

Argument	Description

connect-string	`connect-string ::= datasourcename { ; conn-spec-item }`
	The connection string is the data source name and an optional list of one or more connection specification items. If the data source name contains blanks, or if any connection specification items are listed, the connection string must be enclosed by quotation marks.
	
	datasourcename must be a defined ODBC data source or a string that defines an OLE DB provider.
	
	`conn-spec-item ::=`**`DBQ=`**`database_specifier |`**`DriverID=`**`driver_specifier |`**`UID=`**`userid |`**`PWD=`**`password`
	
	The possible connection specification items may differ between different databases. For some databases, also other items than the above are possible. For OLE DB, some of the connection specific items are mandatory and not optional.

connection	The name of a data connection stored in the data load editor.

If the **ODBC** is placed before **CONNECT**, the ODBC interface will be used; else, OLE DB will be used.

Using **LIB CONNECT TO** connects to a database using a stored data connection that was created in the data load editor.

Example 1:

```
ODBC CONNECT TO 'Sales
DBQ=C:\Program Files\Access\Samples\Sales.mdb';
```

The data source defined through this statement is used by subsequent **Select (SQL)** statements, until a new **CONNECT** statement is made.

Example 2:

```
LIB CONNECT TO 'DataConnection';
```

Connect32

This statement is used the same way as the **CONNECT** statement, but forces a 64-bit system to use a 32-bit ODBC/OLE DB provider. Not applicable for custom connect.

Connect64

This statement is used the same way as the as the **CONNECT** statement, but forces use of a 64-bit provider. Not applicable for custom connect.

Declare

The **Declare** statement is used to create field definitions, where you can define relations between fields or functions. A set of field definitions can be used to automatically generate derived fields, which can be used as dimensions. For example, you can create a calendar definition, and use that to generate related dimensions, such as year, month, week and day, from a date field.

You can use **Declare** to either set up a new field definition, or to create a field definition based on an already existing definition.

Setting up a new field definition

Syntax:

```
definition_name:
Declare [Field[s]] Definition [Tagged tag_list ]
[Parameters parameter_list ]
Fields field_list
```

Arguments:

Argument	Description
definition_name	Name of the field definition, ended with a colon. *Do not use autoCalendar as name for field definitions, as this name is reserved for auto-generated calendar templates.* **Example:** `Calendar:`
tag_list	A comma separated list of tags to apply to fields derived from the field definition. Applying tags is optional, but if you do not apply tags that are used to specify sort order, such as $date, $numeric or $text, the derived field will be sorted by load order as default. **Example:** `'$date'Thank you for bringing this to our attention, and apologies for the inconvenience.`
parameter_list	A comma separated list of parameters. A parameter is defined in the form `name=value` and is assigned a start value, which can be overridden when a field definition is re-used. Optional. **Example:** `first_month_of_year = 1`

Argument	Description
field_list	A comma separated list of fields to generate when the field definition is used. A field is defined in the form `<expression> As field_name tagged tag`. Use $1 to reference the data field from which the derived fields should be generated. **Example:** `Year($1) As Year tagged ('$numeric')`

Example:

```
Calendar:
DECLARE FIELD DEFINITION TAGGED '$date'
   Parameters
      first_month_of_year = 1
   Fields
      Year($1) As Year Tagged ('$numeric'),
      Month($1) as Month Tagged ('$numeric'),
      Date($1) as Date Tagged ('$date'),
      Week($1) as Week Tagged ('$numeric'),
      Weekday($1) as Weekday Tagged ('$numeric'),
      DayNumberOfYear($1, first_month_of_year) as DayNumberOfYear Tagged ('$numeric')
;
```

The calendar is now defined, and you can apply it to the date fields that have been loaded, in this case OrderDate and ShippingDate, using a **Derive** clause.

Re-using an existing field definition

Syntax:

```
<definition name>:
Declare [Field][s] Definition
Using <existing_definition>
[With <parameter_assignment> ]
```

Arguments:

Argument	Description
definition_name	Name of the field definition, ended with a colon. **Example:** `MyCalendar:`

Argument	Description
existing_ definition	The field definition to re-use when creating the new field definition. The new field definition will function the same way as the definition it is based on, with the exception if you use parameter_assignment to change a value used in the field expressions. **Example:** `Using Calendar`
parameter_ assignment	A comma separated list of parameter assignments. A parameter assignment is defined in the form `name=value` and overrides the parameter value that is set in the base field definition. Optional. **Example:** `first_month_of_year = 4`

Example:

In this example we re-use the calendar definition that was created in the previous example. In this case we want to use a fiscal year that starts in April. This is achieved by assigning the value 4 to the first_month_of_year parameter, which will affect the DayNumberOfYear field that is defined.

The example assumes that you use the sample data and field definition from the previous example.

```
MyCalendar:
DECLARE FIELD DEFINITION USING Calendar WITH first_month_of_year=4;

DERIVE FIELDS FROM FIELDS OrderDate,ShippingDate USING MyCalendar;
```

When you have reloaded the data script, the generated fields are available in the sheet editor, with names OrderDate.MyCalendar.* and ShippingDate.MyCalendar.*.

Derive

The **Derive** statement is used to generate derived fields based on a field definition created with a **Declare** statement. You can either specify which data fields to derive fields for, or derive them explicitly or implicitly based on field tags.

Syntax:

```
Derive [Field[s]] From [Field[s]] field_list Using definition
Derive [Field[s]] From Explicit [Tag[s]] tag_list Using definition
Derive [Field[s]] From Implicit [Tag[s]] Using definition
```

Arguments:

Arguments

Argument	Description
definition	Name of the field definition to use when deriving fields. **Example:** `Calendar`
field_list	A comma separated list of data fields from which the derived fields should be generated, based on the field definition. The data fields should be fields you have already loaded in the script. **Example:** `OrderDate, ShippingDate`
tag_list	A comma separated list of tags. Derived fields will be generated for all data fields with any of the listed tags. The list of tags should be enclosed by round brackets. **Example:** `('$date','$timestamp')`

Examples:

- Derive fields for specific data fields.
  In this case we specify the OrderDate and ShippingDate fields.
  `DERIVE FIELDS FROM FIELDS OrderDate,ShippingDate USING Calendar;`
- Derive fields for all fields with a specific tag.
  In this case we derive fields based on Calendar for all fields with a `$date` tag.
  `DERIVE FIELDS FROM EXPLICIT TAGS ('$date') USING Calendar;`
- Derive fields for all fields with the field definition tag.
  In this case we derive fields for all data fields with the same tag as the Calendar field definition, which in this case is `$date`.
  `DERIVE FIELDS FROM IMPLICIT TAG USING Calendar;`

Directory

The **Directory** statement defines which directory to look in for data files in subsequent **LOAD** statements, until a new **Directory** statement is made.

Syntax:

```
Directory[path]
```

If the **Directory** statement is issued without a **path** or left out, Qlik Sense will look in the Qlik Sense working directory.

Arguments:

Arguments

Argument	Description
path	A text that can be interpreted as the path to the data file. The path is the path to the file, either: • absolute **Example:** ***c:\data*** • relative to the Qlik Sense app working directory. **Example:** ***data*** • URL address (HTTP or FTP), pointing to a location on the Internet or an intranet. **Example:** ***http://www.qlik.com***

Examples:

```
DIRECTORY C:\userfiles\data; // OR -> DIRECTORY data\

LOAD * FROM
[data1.csv] // ONLY THE FILE NAME CAN BE SPECIFIED HERE (WITHOUT THE FULL PATH)
(ansi, txt, delimiter is ',', embedded labels);

LOAD * FROM
[data2.txt] // ONLY THE FILE NAME CAN BE SPECIFIED HERE UNTIL A NEW DIRECTORY STATEMENT IS
MADE
(ansi, txt, delimiter is '\t', embedded labels);
```

Disconnect

The **Disconnect** statement terminates the current ODBC/OLE DB/Custom connection. This statement is optional.

Syntax:

```
Disconnect
```

The connection will be automatically terminated when a new **connect** statement is executed or when the script execution is finished.

Example:

```
Disconnect;
```

Drop

The **Drop** script keyword can be used to drop tables or fields from the database.

Drop field

One or several Qlik Sense fields can be dropped from the data model, and thus from memory, at any time during script execution, by means of a **drop field** statement.

> *Both **drop field** and **drop fields** are allowed forms with no difference in effect. If no table is specified, the field will be dropped from all tables where it occurs.*

Syntax:

```
Drop field fieldname { , fieldname2 ...} [from tablename1 { , tablename2
...}]
Drop fields fieldname { , fieldname2 ...} [from tablename1 { , tablename2
...}]
```

Examples:

```
Drop field A;
Drop fields A,B;
Drop field A from X;
Drop fields A,B from X,Y;
```

Drop table

One or several Qlik Sense internal tables can be dropped from the data model, and thus from memory, at any time during script execution, by means of a **drop table** statement.

Syntax:

```
drop table  tablename {, tablename2 ...}
drop tables tablename {, tablename2 ...}
```

> *The forms **drop table** and **drop tables** are both accepted.*

The following items will be lost as a result of this:

- The actual table(s).
- All fields which are not part of remaining tables.
- Field values in remaining fields, which came exclusively from the dropped table(s).

Examples and results:

Example	Result
`drop table Orders, Salesmen, T456a;`	This line results in three tables being dropped from memory.

`Tab1:`
`Load * Inline [`
`Customer, Items, UnitPrice`
`Bob, 5, 1.50`
`];`
 	
`Tab2:`
`LOAD Customer, Sum( Items * UnitPrice ) as Sales`
`resident Tab1`
`group by Customer;`
 	
`drop table Tab1;`
	Once the table *Tab2* is created, the table *Tab1* is dropped.

Drop table

One or several Qlik Sense internal tables can be dropped from the data model, and thus from memory, at any time during script execution, by means of a **drop table** statement.

Syntax:

```
drop table  tablename {, tablename2 ...}
drop tables tablename {, tablename2 ...}
```

*The forms **drop table** and **drop tables** are both accepted.*

The following items will be lost as a result of this:

- The actual table(s).
- All fields which are not part of remaining tables.
- Field values in remaining fields, which came exclusively from the dropped table(s).

Examples and results:

Example	Result
`drop table Orders, Salesmen, T456a;`	This line results in three tables being dropped from memory.

Example	Result

`Tab1:`
`Load * Inline [`
`Customer, Items, UnitPrice`
`Bob, 5, 1.50`
`];`
 	
`Tab2:`
`LOAD Customer, Sum( Items * UnitPrice ) as Sales`
`resident Tab1`
`group by Customer;`
 	
`drop table Tab1;`
	Once the table *Tab2* is created, the table *Tab1* is dropped.

Execute

The **Execute** statement is used to run other programs while Qlik Sense is loading data. For example, to make conversions that are necessary.

This functionality is not available in Qlik Sense SaaS.

This statement is not supported in standard mode.

Syntax:

```
execute commandline
```

Arguments:

Arguments

Argument	Description
commandline	A text that can be interpreted by the operating system as a command line. You can refer to an absolute file path or a lib:// folder path.

If you want to use **Execute** the following conditions need to be met:

- You must run in legacy mode (applicable for Qlik Sense and Qlik Sense Desktop).
- You need to set OverrideScriptSecurity to 1 in *Settings.ini* (applicable for Qlik Sense). *Settings.ini* is located in *C:\ProgramData\Qlik\Sense\Engine* and is generally an empty file.

*If you set OverrideScriptSecurity to enable **Execute**, any user can execute files on the server. For example, a user can attach an executable file to an app, and then execute the file in the data load script.*

Do the following:

1. Make a copy of *Settings.ini* and open it in a text editor.
2. Check that the file includes *[Settings 7]* in the first line.
3. Insert a new line and type *OverrideScriptSecurity=1*.
4. Insert an empty line at the end of the file.
5. Save the file.
6. Substitute *Settings.ini* with your edited file.
7. Restart Qlik Sense Engine Service (QES).

If Qlik Sense is running as a service, some commands may not behave as expected.

Example:

```
Execute C:\Program Files\Office12\Excel.exe;
Execute lib://win\notepad.exe // win is a folder connection referring to c:\windows
```

Field/Fields

The **Field** and **Fields** script keywords are used in **Declare**, **Derive**, **Drop**, **Comment**, **Rename** and **Tag/Untag** statements.

FlushLog

The **FlushLog** statement forces Qlik Sense to write the content of the script buffer to the script log file.

Syntax:

```
FlushLog
```

The content of the buffer is written to the log file. This command can be useful for debugging purposes, as you will receive data that otherwise may have been lost in a failed script execution.

Example:

```
FlushLog;
```

Force

The **force** statement forces Qlik Sense to interpret field names and field values of subsequent **LOAD** and **SELECT** statements as written with only upper case letters, with only lower case letters, as always capitalized or as they appear (mixed). This statement makes it possible to associate field values from tables made according to different conventions.

Syntax:

```
Force ( capitalization | case upper | case lower | case mixed )
```

If nothing is specified, force case mixed is assumed. The force statement is valid until a new force statement is made.

The **force** statement has no effect in the access section: all field values loaded are case insensitive.

Examples and results

Example	Result

This example shows how to force capitalization.
 	
`FORCE Capitalization;`
`Capitalization:`
`LOAD * Inline [`
`ab`
`Cd`
`eF`
`GH`
`];`
	The **Capitalization** table contains the following values:
	
	`Ab`
	`Cd`
	`Ef`
	`Gh`
	All values are capitalized.

This example shows how to force case upper.
 	
`FORCE Case Upper;`
`CaseUpper:`
`LOAD * Inline [`
`ab`
`Cd`
`eF`
`GH`
`];`
	The **CaseUpper** table contains the following values:
	
	`AB`
	`CD`
	`EF`
	`GH`
	All values are upper case.

This example shows how to force case lower.
 	
`FORCE Case Lower;`
`CaseLower:`
`LOAD * Inline [`
`ab`
`Cd`
`eF`
`GH`
`];`
	The **CaseLower** table contains the following values:
	
	`ab`
	`cd`
	`ef`
	`gh`
	All values are lower case.

This example shows how to force case mixed.
 	
`FORCE Case Mixed;`
`CaseMixed:`
`LOAD * Inline [`
`ab`
`Cd`
`eF`
`GH`
`];`
	The **CaseMixed** table contains the following values:
	
	`ab`
	`Cd`
	`eF`
	`GH`
	All values are as they appear in the script.

See also:

From

The **From** script keyword is used in **Load** statements to refer to a file, and in **Select** statements to refer to a database table or view.

Load

The **LOAD** statement loads fields from a file, from data defined in the script, from a previously loaded table, from a web page, from the result of a subsequent **SELECT** statement or by generating data automatically. It is also possible to load data from analytic connections.

Syntax:

```
LOAD [ distinct ] fieldlist
[( from file [ format-spec ] |
from_field fieldassource [format-spec]|
inline data [ format-spec ] |
resident table-label |
autogenerate size ) |extension pluginname.functionname([script]
tabledescription)]
[ where criterion | while criterion ]
[ group by groupbyfieldlist ]
[order by orderbyfieldlist ]
```

Arguments:

Arguments

Argument	Description
distinct	You can use **distinct** as a predicate if you only want to load unique records. If there are duplicate records, the first instance will be loaded. If you are using preceding loads, you need to place **distinct** in the first load statement, as **distinct** only affects the destination table.

Argument	Description
fieldlist	*fieldlist ::= (* | field {**,** * | field })* A list of the fields to be loaded. Using * as a field list indicates all fields in the table. *field ::= (fieldref | expression) [**as** aliasname]* The field definition must always contain a literal, a reference to an existing field, or an expression. *fieldref ::= (fieldname |@fieldnumber |@startpos:endpos [**I** | **U** | **R** | **B** | **T**])* *fieldname* is a text that is identical to a field name in the table. Note that the field name must be enclosed by straight double quotation marks or square brackets if it contains e.g. spaces. Sometimes field names are not explicitly available. Then a different notation is used: *@fieldnumber* represents the field number in a delimited table file. It must be a positive integer preceded by "@". The numbering is always made from 1 and up to the number of fields. *@startpos*:*endpos* represents the start and end positions of a field in a file with fixed length records. The positions must both be positive integers. The two numbers must be preceded by "@" and separated by a colon. The numbering is always made from 1 and up to the number of positions. In the last field, **n** is used as end position. • If *@startpos*:*endpos* is immediately followed by the characters **I** or **U**, the bytes read will be interpreted as a binary signed (**I**) or unsigned (**U**) integer (Intel byte order). The number of positions read must be 1, 2 or 4. • If *@startpos*:*endpos* is immediately followed by the character **R**, the bytes read will be interpreted as a binary real number (IEEE 32-bit or 64 bit floating point). The number of positions read must be 4 or 8. • If *@startpos*:*endpos* is immediately followed by the character **B**, the bytes read will be interpreted as a BCD (Binary Coded Decimal) numbers according to the COMP-3 standard. Any number of bytes may be specified. *expression* can be a numeric function or a string function based on one or several other fields in the same table. For further information, see the syntax of expressions. **as** is used for assigning a new name to the field.

Argument	Description
from	**from** is used if data should be loaded from a file using a folder or a web file data connection. *file ::= [path] filename* **Example: *'lib://Table Files/'*** If the path is omitted, Qlik Sense searches for the file in the directory specified by the **Directory** statement. If there is no **Directory** statement, Qlik Sense searches in the working directory, *C:\Users\{user}\Documents\Qlik\Sense\Apps*. *In a Qlik Sense server installation, the working directory is specified in Qlik Sense Repository Service, by default it is C:\ProgramData\Qlik\Sense\Apps.* The *filename* may contain the standard DOS wildcard characters (* and ?). This will cause all the matching files in the specified directory to be loaded. *format-spec ::= (fspec-item { , fspec-item })* The format specification consists of a list of several format specification items, within brackets. **Legacy scripting mode** In legacy scripting mode, the following path formats are also supported: • absolute **Example: *c:\data*** • relative to the Qlik Sense app working directory. **Example: *data*** • URL address (HTTP or FTP), pointing to a location on the Internet or an intranet. **Example: *http://www.qlik.com***
from_field	**from_field** is used if data should be loaded from a previously loaded field. *fieldassource::=***(***tablename, fieldname****)*** The field is the name of the previously loaded *tablename* and *fieldname*. *format-spec ::= (fspec-item {, fspec-item })* The format specification consists of a list of several format specification items, within brackets.

Argument	Description
inline	**inline** is used if data should be typed within the script, and not loaded from a file. *data ::= [text]* Data entered through an **inline** clause must be enclosed by double quotation marks or by square brackets. The text between these is interpreted in the same way as the content of a file. Hence, where you would insert a new line in a text file, you should also do it in the text of an **inline** clause, i.e. by pressing the Enter key when typing the script. The number of columns are defined by the first line. *format-spec ::= (fspec-item {, fspec-item })* The format specification consists of a list of several format specification items, within brackets.
resident	**resident** is used if data should be loaded from a previously loaded table. *table label* is a label preceding the **LOAD** or **SELECT** statement(s) that created the original table. The label should be given with a colon at the end.
autogenerate	**autogenerate** is used if data should be automatically generated by Qlik Sense. *size ::= number* *Number* is an integer indicating the number of records to be generated. The field list must not contain expressions which require data from an external data source or a previously loaded table, unless you refer to a single field value in a previously loaded table with the **Peek** function.

Argument	Description
extension	You can load data from analytic connections. You need to use the **extension** clause to call a function defined in the server-side extension (SSE) plugin, or evaluate a script. You can send a single table to the SSE plugin, and a single data table is returned. If the plugin does not specify the names of the fields that are returned, the fields will be named Field1, Field2, and so on. `Extension pluginname.functionname( tabledescription );` • Loading data using a function in an SSE plugin *tabledescription ::= (table { ,tablefield})* If you do not state table fields, the fields will be used in load order. • Loading data by evaluating a script in an SSE plugin *tabledescription ::= (script, table { ,tablefield})* **Data type handling in the table field definition** Data types are automatically detected in analytic connections. If the data has no numeric values and at least one non-NULL text string, the field is considered as text. In any other case it is considered as numeric. You can force the data type by wrapping a field name with **String()** or **Mixed()**. • **String()** forces the field to be text. If the field is numeric, the text part of the dual value is extracted, there is no conversion performed. • **Mixed()** forces the field to be dual. **String()** or **Mixed()** cannot be used outside **extension** table field definitions, and you cannot use other Qlik Sense functions in a table field definition. **More about analytic connections** You need to configure analytic connections before you can use them.
where	**where** is a clause used for stating whether a record should be included in the selection or not. The selection is included if *criterion* is True. *criterion* is a logical expression.
while	**while** is a clause used for stating whether a record should be repeatedly read. The same record is read as long as *criterion* is True. In order to be useful, a **while** clause must typically include the **IterNo( )** function. *criterion* is a logical expression.

Argument	Description
group by	**group by** is a clause used for defining over which fields the data should be aggregated (grouped). The aggregation fields should be included in some way in the expressions loaded. No other fields than the aggregation fields may be used outside aggregation functions in the loaded expressions. *groupbyfieldlist ::= (fieldname { ,fieldname })*
order by	**order by** is a clause used for sorting the records of a resident table before they are processed by the **load** statement. The resident table can be sorted by one or more fields in ascending or descending order. The sorting is made primarily by numeric value and secondarily by national collation order. This clause may only be used when the data source is a resident table. The ordering fields specify which field the resident table is sorted by. The field can be specified by its name or by its number in the resident table (the first field is number 1). orderbyfieldlist ::= fieldname [sortorder] { , fieldname [sortorder] } *sortorder* is either *asc* for ascending or *desc* for descending. If no *sortorder* is specified, *asc* is assumed. *fieldname*, *path*, *filename* and *aliasname* are text strings representing what the respective names imply. Any field in the source table can be used as *fieldname*. However, fields created through the as clause (*aliasname*) are out of scope and cannot be used inside the same **load** statement.

If no source of data is given by means of a **from**, **inline**, **resident**, **from_field, extension** or **autogenerate** clause, data will be loaded from the result of the immediately succeeding **SELECT** or **LOAD** statement. The succeeding statement should not have a prefix.

Examples:

Loading different file formats
Load a delimited data file with default options:

```
LOAD * from data1.csv;
```

Load a delimited data file from a library connection (DataFiles):

```
LOAD * from 'lib://DataFiles/data1.csv';
```

Load all delimited data files from a library connection (DataFiles):

```
LOAD * from 'lib://DataFiles/*.csv';
```

Load a delimited file, specifying comma as delimiter and with embedded labels:

```
LOAD * from 'c:\userfiles\data1.csv' (ansi, txt, delimiter is ',', embedded labels);
```

Load a delimited file specifying tab as delimiter and with embedded labels:

```
LOAD * from 'c:\userfiles\data2.txt' (ansi, txt, delimiter is '\t', embedded labels);
```

Load a dif file with embedded headers:

```
LOAD * from file2.dif (ansi, dif, embedded labels);
```

Load three fields from a fixed record file without headers:

```
LOAD @1:2 as ID, @3:25 as Name, @57:80 as City from data4.fix (ansi, fix, no labels, header is
0, record is 80);
```

Load a QVX file, specifying an absolute path:

```
LOAD * from C:\qdssamples\xyz.qvx (qvx);
```

Loading web files

Load from the default URL set in the web file data connection:

```
LOAD * from [lib://MyWebFile];
```

Load from a specific URL, and override the URL set in the web file data connection:

```
LOAD * from [lib://MyWebFile] (URL is 'http://localhost:8000/foo.bar');
```

Load from a specific URL set in a variable using dollar-sign expansion:

```
SET dynamicURL = 'http://localhost/foo.bar';
LOAD * from [lib://MyWebFile] (URL is '$(dynamicURL)');
```

Selecting certain fields, renaming and calculating fields

Load only three specific fields from a delimited file:

```
LOAD FirstName, LastName, Number from data1.csv;
```

Rename first field as A and second field as B when loading a file without labels:

```
LOAD @1 as A, @2 as B from data3.txt (ansi, txt, delimiter is '\t', no labels);
```

Load Name as a concatenation of FirstName, a space character, and LastName:

```
LOAD FirstName&' '&LastName as Name from data1.csv;
```

Load Quantity, Price and Value (the product of Quantity and Price):

```
LOAD Quantity, Price, Quantity*Price as Value from data1.csv;
```

Selecting certain records

Load only unique records, duplicate records will be discarded:

```
LOAD distinct FirstName, LastName, Number from data1.csv;
```

Load only records where the field Litres has a value above zero:

```
LOAD * from Consumption.csv where Litres>0;
```

Loading data not on file and auto-generated data

Load a table with inline data, two fields named CatID and Category:

```
LOAD * Inline
[CatID, Category
0,Regular
1,Occasional
2,Permanent];
```

Load a table with inline data, three fields named UserID, Password and Access:

```
LOAD * Inline [UserID, Password, Access
A, ABC456, User
B, VIP789, Admin];
```

Load a table with 10 000 rows. Field A will contain the number of the read record (1,2,3,4,5...) and field B will contain a random number between 0 and 1:

```
LOAD RecNo( ) as A, rand( ) as B autogenerate(10000);
```

The parenthesis after autogenerate is allowed but not required.

Loading data from a previously loaded table

First we load a delimited table file and name it tab1:

```
tab1:
SELECT A,B,C,D from 'lib://DataFiles/data1.csv';
```

Load fields from the already loaded tab1 table as tab2:

```
tab2:
LOAD A,B,month(C),A*B+D as E resident tab1;
```

Load fields from already loaded table tab1 but only records where A is larger than B:

```
tab3:
LOAD A,A+B+C resident tab1 where A>B;
```

Load fields from already loaded table tab1 ordered by A:

```
LOAD A,B*C as E resident tab1 order by A;
```

Load fields from already loaded table tab1, ordered by the first field, then the second field:

```
LOAD A,B*C as E resident tab1 order by 1,2;
```

Load fields from already loaded table tab1 ordered by C descending, then B in ascending order, and then the first field in descending order:

```
LOAD A,B*C as E resident tab1 order by C desc, B asc, 1 desc;
```

Loading data from previously loaded fields

Load field Types from previously loaded table Characters as A:

```
LOAD A from_field (Characters, Types);
```

Loading data from a succeeding table (preceding load)

Load A, B and calculated fields X and Y from Table1 that is loaded in succeeding **SELECT** statement:

```
LOAD A, B, if(C>0,'positive','negative') as X, weekday(D) as Y;
SELECT A,B,C,D from Table1;
```

Grouping data

Load fields grouped (aggregated) by ArtNo:

```
LOAD ArtNo, round(Sum(TransAmount),0.05) as ArtNoTotal from table.csv group by ArtNo;
```

Load fields grouped (aggregated) by Week and ArtNo:

```
LOAD Week, ArtNo, round(Avg(TransAmount),0.05) as WeekArtNoAverages from table.csv group by
Week, ArtNo;
```

Reading one record repeatedly

In this example we have a input file Grades.csv containing the grades for each student condensed in one field:

```
Student,Grades
Mike,5234
John,3345
Pete,1234
Paul,3352
```

The grades, in a 1-5 scale, represent subjects Math, English, Science and History. We can separate the grades into separate values by reading each record several times with a **while** clause, using the **IterNo( )** function as a counter. In each read, the grade is extracted with the **Mid** function and stored in Grade, and the subject is selected using the **pick** function and stored in Subject. The final **while** clause contains the test to check if all grades have been read (four per student in this case), which means next student record should be read.

```
MyTab:
LOAD Student,
mid(Grades,IterNo( ),1) as Grade,
pick(IterNo( ), 'Math', 'English', 'Science', 'History') as Subject from Grades.csv
while IsNum(mid(Grades,IterNo(),1));
```

The result is a table containing this data:

Student	Subject	Grade
John	English	3
John	History	5
John	Math	3
John	Science	4
Mike	English	2
Mike	History	4
Mike	Math	5
Mike	Science	3
Paul	English	3
Paul	History	2
Paul	Math	3
Paul	Science	5
Pete	English	2
Pete	History	4
Pete	Math	1
Pete	Science	3

Loading from analytic connections
The following sample data is used.

```
Values:
Load
 Rand() as A,
 Rand() as B,
 Rand() as C
AutoGenerate(50);
```

Loading data using a function

In these examples, we assume that we have an analytic connection plugin named *P* that contains a custom function *Calculate(Parameter1, Parameter2)*. The function returns the table *Results* that contains the fields *Field1* and *Field2*.

`Load * Extension P.Calculate( Values{A, C} );`
Load all fields that are returned when sending the fields A and C to the function.

`Load Field1 Extension P.Calculate( Values{A, C} );`
Load only the Field1 field when sending the fields A and C to the function.

`Load * Extension P.Calculate( Values );`
Load all fields that are returned when sending the fields A and B to the function. As fields are not specified, A and B are used as they are the first in order in the table.

`Load * Extension P.Calculate( Values {C, C});`
Load all fields that are returned when sending the field C to both parameters of the function.

`Load * Extension P.Calculate( Values {String(A), Mixed(B)});`
Load all fields that are returned when sending the field A forced as a string and B forced as a numeric to the function.

Loading data by evaluating a script

`Load A as A_echo, B as B_echo Extension R.ScriptEval( 'q;', Values{A, B} );`
Load the table returned by the script q when sending the values of A and B.

`Load * Extension R.ScriptEval( '$(My_R_Script)', Values{A, B} );`
Load the table returned by the script stored in the My_R_Script variable when sending the values of A and B.

`Load * Extension R.ScriptEval( '$(My_R_Script)', Values{B as D, *} );`
Load the table returned by the script stored in the My_R_Script variable when sending the values of B renamed to D, A and C. Using * sends the remaining unreferenced fields.

The file extension of DataFiles connections is case sensitive. For example: .qvd.

Format specification items

Each format specification item defines a certain property of the table file:

`fspec-item ::=` [**ansi | oem | mac | UTF-8 | Unicode | txt | fix | dif | biff | ooxml | html | xml | kml | qvd | qvx | delimiter is** `char` | **no eof | embedded labels | explicit labels | no labels | table is [**`tablename`**]** | **header is** `n` | **header is** `line` | **header is** `n` **lines** | **comment is** `string` | **record is** `n` | **record is** `line` | **record is** `n` **lines** | **no quotes |msq** | **URL is** `string` | **userAgent is** `string`]

Character set

Character set is a file specifier for the **LOAD** statement that defines the character set used in the file.

The **ansi**, **oem** and **mac** specifiers were used in QlikView and will still work. However, they will not be generated when creating the **LOAD** statement with Qlik Sense.

Syntax:

```
utf8 | unicode | ansi | oem | mac | codepage is
```

Arguments:

Arguments

Argument	Description
utf8	UTF-8 character set
unicode	Unicode character set
ansi	Windows, codepage 1252
oem	DOS, OS/2, AS400 and others
mac	Codepage 10000
codepage is	With the **codepage** specifier, it is possible to use any Windows codepage as *N* .

Limitations:

Conversion from the **oem** character set is not implemented for MacOS. If nothing is specified, codepage 1252 is assumed under Windows.

Example:

```
LOAD * from a.txt (utf8, txt, delimiter is ',' , embedded labels)
LOAD * from a.txt (unicode, txt, delimiter is ',' , embedded labels)
LOAD * from a.txt (codepage is 10000, txt, delimiter is ',' , no labels)
```

See also:

Load (page 119)

Table format

The table format is a file specifier for the **LOAD** statement that defines the file type. If nothing is specified, a *.txt* file is assumed.

Table format types

Type	Description
txt	In a delimited text file the columns in the table are separated by a delimiter character.
fix	In a fixed record file, each field is exactly a certain number of characters. Typically, many fixed record length files contains records separated by a linefeed, but there are more advanced options to specify record size in bytes or to span over more than one line with **Record is**. *If the data contains multi-byte characters, field breaks can become misaligned as the format is based on a fixed length in bytes.*
dif	In a *.dif* file, (Data Interchange Format) a special format for defining the table is used.
biff	Qlik Sense can also interpret data in standard Excel files by means of the *biff* format (Binary Interchange File Format).
ooxml	Excel 2007 and later versions use the ooxml *.xslx* format.
html	If the table is part of an html page or file, html should be used.
xml	xml (Extensible Markup Language) is a common markup language that is used to represent data structures in a textual format.
qvd	The format *qvd* is the proprietary QVD files format, exported from a Qlik Sense app.
qvx	*qvx* is a file/stream format for high performance output to Qlik Sense.

Delimiter is

For delimited table files, an arbitrary delimiter can be specified through the **delimiter is** specifier. This specifier is relevant only for delimited .txt files.

Syntax:

```
delimiter is char
```

Arguments:

Arguments

Argument	Description
char	Specifies a single character from the 127 ASCII characters.

Additionally, the following values can be used:

Optional values

Value	Description
'\t'	representing a tab sign, with or without quotation marks.
'\\'	representing a backslash (\ ) character.
'spaces'	representing all combinations of one or more spaces. Non-printable characters with an ASCII-value below 32, with the exception of CR and LF, will be interpreted as spaces.

If nothing is specified, **delimiter is ','** is assumed.

Example:

```
LOAD * from a.txt (utf8, txt, delimiter is ',' , embedded labels);
```

See also:

Load (page 119)

No eof

The **no eof** specifier is used to disregard end-of-file character when loading delimited **.txt** files.

Syntax:

```
no eof
```

If the **no eof** specifier is used, characters with code point 26, which otherwise denotes end-of-file, are disregarded and can be part of a field value.

It is relevant only for delimited text files.

Example:

```
LOAD * from a.txt (txt, utf8, embedded labels, delimiter is ' ', no eof);
```

See also:

Load (page 119)

Labels

Labels is a file specifier for the **LOAD** statement that defines where in a file the field names can be found.

Syntax:

```
embedded labels|explicit labels|no labels
```

The field names can be found in different places of the file. If the first record contains the field names, **embedded labels** should be used. If there are no field names to be found, **no labels** should be used. In *dif* files, a separate header section with explicit field names is sometimes used. In such a case, **explicit labels** should be used. If nothing is specified, **embedded labels** is assumed, also for *dif* files.

Example 1:

```
LOAD * from a.txt (unicode, txt, delimiter is ',' , embedded labels
```

Example 2:

```
LOAD * from a.txt (codePage is 1252, txt, delimiter is ',' , no labels)
```

See also:

Load (page 119)

Header is

Specifies the header size in table files. An arbitrary header length can be specified through the **header is** specifier. A header is a text section not used by Qlik Sense.

Syntax:

```
header is n
header is line
header is n lines
```

The header length can be given in bytes (**header is n**), or in lines (**header is line** or **header is n lines**). **n** must be a positive integer, representing the header length. If not specified, **header is 0** is assumed. The **header is** specifier is only relevant for table files.

Example:

This is an example of a data source table containing a header text line that should not be interpreted as data by Qlik Sense.

```
*Header line
Col1,Col2
a,B
c,D
```

Using the **header is 1 lines** specifier, the first line will not be loaded as data. In the example, the **embedded labels** specifier tells Qlik Sense to interpret the first non-excluded line as containing field labels.

```
LOAD Col1, Col2
FROM 'lib://files/header.txt'
(txt, embedded labels, delimiter is ',', msq, header is 1 lines);
```

The result is a table with two fields, Col1 and Col2.

See also:

Load (page 119)

Record is

For fixed record length files, the record length must be specified through the **record is** specifier.

Syntax:

```
Record is n
Record is line
Record is n lines
```

Arguments:

Arguments

Argument	Description
n	Specifies the record length in bytes.
line	Specifies the record length as one line.
n lines	Specifies the record length in lines where n is a positive integer representing the record length.

Limitations:

The **record is** specifier is only relevant for **fix** files.

See also:

Load (page 119)

Quotes

Quotes is a file specifier for the **LOAD** statement that defines whether quotes can be used and the precedence between quotes and separators. For text files only.

Syntax:

```
no quotes
msq
```

If the specifier is omitted, standard quoting is used, that is, the quotes " " or ' ' can be used, but only if they are the first and last non blank character of a field value.

Arguments:

Arguments

Argument	Description
no quotes	Used if quotation marks are not to be accepted in a text file.
msq	Used to specify modern style quoting, allowing multi-line content in fields. Fields containing end-of-line characters must be enclosed within double quotes. One limitation of the msq option is that single double-quote (") characters appearing as first or last character in field content will be interpreted as start or end of multi-line content, which may lead to unpredicted results in the data set loaded. In this case you should use standard quoting instead, omitting the specifier.

XML

This script specifier is used when loading xml files. Valid options for the **XML** specifier are listed in syntax.

You cannot load DTD files in Qlik Sense.

Syntax:

```
xmlsimple
```

See also:

Load (page 119)

KML

This script specifier is used when loading KML files to use in a map visualization.

Syntax:

```
kml
```

The KML file can represent either area data (for example, countries or regions) represented by polygons, line data (for example tracks or roads), or point data (for example, cities or places) represented by points in the form [long, lat].

URL is

This script specifier is used to set the URL of a web file data connection when loading a web file.

Syntax:

```
URL is string
```

Arguments:

Arguments

Argument	Description
string	Specifies the URL of the file to load. This will override the URL set in the web file connection that is used.

Limitations:

The **URL is** specifier is only relevant for web files. You need to use an existing web file data connection.

See also:

Load (page 119)

userAgent is

This script specifier is used to set the browser user agent when loading a web file.

Syntax:

```
userAgent is string
```

Arguments:

Arguments

Argument	Description
string	Specifies the browser user agent string. This will override the default browser user agent "Mozilla/5.0".

Limitations:

The **userAgent is** specifier is only relevant for web files.

See also:

Load (page 119)

Let

The **let** statement is a complement to the **set** statement, used for defining script variables. The **let** statement, in opposition to the **set** statement, evaluates the expression on the right side of the ' =' at script run time before it is assigned to the variable.

Syntax:

```
Let variablename=expression
```

Examples and results:

Example	Result

`Set x=3+4;`
`Let y=3+4;`
`z=$(y)+1;`
	`$(x)` will be evaluated as ' 3+4 '
	
	`$(y)` will be evaluated as ' 7 '
	
	`$(z)` will be evaluated as ' 8 '
	
	Note the difference between the **Set** and **Let** statements. The **Set** statement assigns the string '3+4' to the variable, whereas the **Let** statement evaluates the string and assigns 7 to the variable.

Let T=now( );	`$(T)` will be given the value of the current time.

Loosen Table

One or more Qlik Sense internal data tables can be explicitly declared loosely coupled during script execution by using a **Loosen Table** statement. When a table is loosely coupled, all associations between field values in the table are removed. A similar effect could be achieved by loading each field of the loosely coupled table as independent, unconnected tables. Loosely coupled can be useful during testing to temporarily isolate different parts of the data structure. A loosely coupled table can be identified in the table viewer by the dotted lines. The use of one or more **Loosen Table** statements in the script will make Qlik Sense disregard any setting of tables as loosely coupled made before the script execution.

Syntax:

```
Loosen Tabletablename [ , tablename2 ...]
Loosen Tablestablename [ , tablename2 ...]
```

Either syntax: **Loosen Table** or **Loosen Tables** can be used.

*Should Qlik Sense find circular references in the data structure which cannot be broken by tables declared loosely coupled interactively or explicitly in the script, one or more additional tables will be forced loosely coupled until no circular references remain. When this happens, the **Loop Warning** dialog, gives a warning.*

Example:

```
Tab1:
SELECT * from Trans;
Loosen Table Tab1;
```

Map

The **map ... using** statement is used for mapping a certain field value or expression to the values of a specific mapping table. The mapping table is created through the **Mapping** statement.

Syntax:

```
Map fieldlist Using  mapname
```

The automatic mapping is done for fields loaded after the **Map ... Using** statement until the end of the script or until an **Unmap** statement is encountered.

The mapping is done last in the chain of events leading up to the field being stored in the internal table in Qlik Sense. This means that mapping is not done every time a field name is encountered as part of an expression, but rather when the value is stored under the field name in the internal table. If mapping on the expression level is required, the **Applymap()** function has to be used instead.

Arguments:

Arguments

Argument	Description
fieldlist	A comma separated list of the fields that should be mapped from this point in the script. Using * as field list indicates all fields. The wildcard characters * and ? are allowed in field names. Quoting of field names may be necessary when wildcards are used.
mapname	The name of a mapping table previously read in a **mapping load** or **mapping select** statement.

Examples and results:

Example	Result
Map Country Using Cmap;	Enables mapping of the field Country using the map Cmap.
Map A, B, C Using X;	Enables mapping of the fields A, B and C using the map X.
Map * Using GenMap;	Enables mapping of all fields using GenMap.

NullAsNull

The **NullAsNull** statement turns off the conversion of NULL values to string values previously set by a **NullAsValue** statement.

Syntax:

```
NullAsNull *fieldlist
```

The **NullAsValue** statement operates as a switch and can be turned on or off several times in the script, using either a **NullAsValue** or a **NullAsNull** statement.

Arguments:

Arguments

Argument	Description
*fieldlist	A comma separated list of the fields for which **NullAsNull** should be turned on. Using * as field list indicates all fields. The wildcard characters * and ? are allowed in field names. Quoting of field names may be necessary when wildcards are used.

Example:

```
NullAsNull A,B;
LOAD A,B from x.csv;
```

NullAsValue

The **NullAsValue** statement specifies for which fields that NULL should be converted to a value.

Syntax:

```
NullAsValue *fieldlist
```

By default, Qlik Sense considers NULL values to be missing or undefined entities. However, certain database contexts imply that NULL values are to be considered as special values rather than simply missing values. The fact that NULL values are normally not allowed to link to other NULL values can be suspended by means of the **NullAsValue** statement.

The **NullAsValue** statement operates as a switch and will operate on subsequent loading statements. It can be switched off again by means of the **NullAsNull** statement.

Arguments:

Arguments

Argument	Description
*fieldlist	A comma separated list of the fields for which **NullAsValue** should be turned on. Using * as field list indicates all fields. The wildcard characters * and ? are allowed in field names. Quoting of field names may be necessary when wildcards are used.

Example:

```
NullAsValue A,B;
Set NullValue = 'NULL';
LOAD A,B from x.csv;
```

Qualify

The **Qualify** statement is used for switching on the qualification of field names, i.e. field names will get the table name as a prefix.

Syntax:

```
Qualify *fieldlist
```

The automatic join between fields with the same name in different tables can be suspended by means of the **qualify** statement, which qualifies the field name with its table name. If qualified, the field name(s) will be renamed when found in a table. The new name will be in the form of *tablename.fieldname*. *Tablename* is equivalent to the label of the current table, or, if no label exists, to the name appearing after **from** in **LOAD** and **SELECT** statements.

The qualification will be made for all fields loaded after the **qualify** statement.

Qualification is always turned off by default at the beginning of script execution. Qualification of a field name can be activated at any time using a **qualify** statement. Qualification can be turned off at any time using an **Unqualify** statement.

*The **qualify** statement should not be used in conjunction with partial reload.*

Arguments:

Arguments

Argument	Description
*fieldlist	A comma separated list of the fields for which qualification should be turned on. Using * as field list indicates all fields. The wildcard characters * and ? are allowed in field names. Quoting of field names may be necessary when wildcards are used.

Example 1:

```
Qualify B;
LOAD A,B from x.csv;
LOAD A,B from y.csv;
```

The two tables **x.csv** and **y.csv** are associated only through **A**. Three fields will result: A, x.B, y.B.

Example 2:

In an unfamiliar database, it is often useful to start out by making sure that only one or a few fields are associated, as illustrated in this example:

```
qualify *;
unqualify TransID;
SQL SELECT * from tab1;
SQL SELECT * from tab2;
SQL SELECT * from tab3;
```

Only **TransID** will be used for associations between the tables *tab1*, *tab2* and *tab3*.

Rem

The **rem** statement is used for inserting remarks, or comments, into the script, or to temporarily deactivate script statements without removing them.

Syntax:

```
Rem string
```

Everything between the **rem** and the next semicolon **;** is considered to be a comment.

There are two alternative methods available for making comments in the script:

1. It is possible to create a comment anywhere in the script - except between two quotes - by placing the section in question between **/*** and ***/**.
2. When typing **//** in the script, all text that follows to the right on the same row becomes a comment. (Note the exception //: that may be used as part of an Internet address.)

Arguments:

Arguments

Argument	Description
string	An arbitrary text.

Example:

```
Rem ** This is a comment **;
/* This is also a comment */
// This is a comment as well
```

Rename

The **Rename** script keyword can be used to rename tables or fields that are already loaded.

Rename field

This script function renames one or more existing Qlik Sense field(s) after they have been loaded.

> *It is not recommended to name a variable identically to a field or a function in Qlik Sense.*

Either syntax: **rename field** or **rename fields** can be used.

Syntax:

```
Rename Field (using mapname | oldname to newname{ , oldname to newname })
Rename Fields (using mapname | oldname to newname{ , oldname to newname })
```

Arguments:

Argument	Description
mapname	The name of a previously loaded mapping table containing one or more pairs of old and new field names.
oldname	The old field name.
newname	The new field name.

Limitations:

You cannot rename two fields to having the same name.

Example 1:

```
Rename Field XAZ0007 to Sales;
```

Example 2:

```
FieldMap:
Mapping SQL SELECT oldnames, newnames from datadictionary;
Rename Fields using FieldMap;
```

Rename table

This script function renames one or more existing Qlik Sense internal table(s) after they have been loaded.

Either syntax: **rename table** or **rename tables** can be used.

Syntax:

```
Rename Table (using mapname | oldname to newname{ , oldname to newname })
Rename Tables (using mapname | oldname to newname{ , oldname to newname })
```

Arguments:

Arguments

Argument	Description
mapname	The name of a previously loaded mapping table containing one or more pairs of old and new table names.
oldname	The old table name.
newname	The new table name.

Limitations:

Two differently named tables cannot be renamed to having the same name. The script will generate an error if you try to rename a table to the same name as an existing table.

Example 1:

```
Tab1:
SELECT * from Trans;
Rename Table Tab1 to Xyz;
```

Example 2:

```
TabMap:
Mapping LOAD oldnames, newnames from tabnames.csv;
Rename Tables using TabMap;
```

Search

The **Search** statement is used for including or excluding fields in smart search.

Syntax:

```
Search Include *fieldlist
Search Exclude *fieldlist
```

You can use several Search statements to refine your selection of fields to include. The statements are evaluated from top to bottom.

Arguments:

Arguments

Argument	Description
*fieldlist	A comma separated list of the fields to include or exclude from searches in smart search. Using * as field list indicates all fields. The wildcard characters * and ? are allowed in field names. Quoting of field names may be necessary when wildcards are used.

Example:

Search examples

Statement	Description
`Search Include *;`	Include all fields in searches in smart search.
`Search Exclude [*ID];`	Exclude all fields ending with ID from searches in smart search.
`Search Exclude '*ID';`	Exclude all fields ending with ID from searches in smart search.
`Search Include ProductID;`	Include the field ProductID in searches in smart search.

The combined result of these three statements, in this sequence, is that all fields ending with ID except ProductID are excluded from searches in smart search.

Section

With the **section** statement, it is possible to define whether the subsequent **LOAD** and **SELECT** statements should be considered as data or as a definition of the access rights.

Syntax:

```
Section (access | application)
```

If nothing is specified, **section application** is assumed. The **section** definition is valid until a new **section** statement is made.

Example:

```
Section access;
Section application;
```

Select

The selection of fields from an ODBC data source or OLE DB provider is made through standard SQL **SELECT** statements. However, whether the **SELECT** statements are accepted depends on the ODBC driver or OLE DB provider used. Use of the **SELECT** statement requires an open data connection to the source.

Syntax:

```
Select [all | distinct | distinctrow | top n [percent] ] fieldlist

From tablelist

[where criterion ]

[group by fieldlist [having criterion ] ]

[order by fieldlist [asc | desc] ]

[ (Inner | Left | Right | Full) join tablename on fieldref = fieldref ]
```

Furthermore, several **SELECT** statements can sometimes be concatenated into one through the use of a **union** operator:

```
selectstatement Union selectstatement
```

The **SELECT** statement is interpreted by the ODBC driver or OLE DB provider, so deviations from the general SQL syntax might occur depending on the capabilities of the ODBC drivers or OLE DB provider, for example:.

- **as** is sometimes not allowed, i.e. *aliasname* must follow immediately after *fieldname*.
- **as** is sometimes compulsory if an *aliasname* is used.
- **distinct**, **as**, **where**, **group by**, **order by**, or **union** is sometimes not supported.
- The ODBC driver sometimes does not accept all the different quotation marks listed above.

*This is not a complete description of the SQL **SELECT** statement! E.g. **SELECT** statements can be nested, several joins can be made in one **SELECT** statement, the number of functions allowed in expressions is sometimes very large, etc.*

Arguments:

Arguments

Argument	Description
distinct	**distinct** is a predicate used if duplicate combinations of values in the selected fields only should be loaded once.
distinctrow	**distinctrow** is a predicate used if duplicate records in the source table only should be loaded once.

Argument	Description
fieldlist	**`fieldlist ::= (*| field ) {, field }`** A list of the fields to be selected. Using * as field list indicates all fields in the table. **`fieldlist ::= field {, field }`** A list of one or more fields, separated by commas. **`field ::= ( fieldref| expression ) [as aliasname ]`** The expression can e.g. be a numeric or string function based on one or several other fields. Some of the operators and functions usually accepted are: +, -, *, /, & (string concatenation), sum(fieldname), count(fieldname), avg(fieldname)(average), month (fieldname), etc. See the documentation of the ODBC driver for more information. **`fieldref ::= [ tablename. ] fieldname`** The **tablename** and the **fieldname** are text strings identical to what they imply. They must be enclosed by straight double quotation marks if they contain e.g. spaces. The **as** clause is used for assigning a new name to the field.
from	**`tablelist ::= table {, table }`** The list of tables that the fields are to be selected from. **`table ::= tablename [ [as ] aliasname ]`** The **tablename** may or may not be put within quotes.
where	**where** is a clause used for stating whether a record should be included in the selection or not. **criterion** is a logical expression that can sometimes be very complex. Some of the operators accepted are: numeric operators and functions, **=**, **<>** or **#(**not equal), **>**, **>=**, **<**, **<=**, **and**, **or**, **not**, **exists**, **some**, **all**, **in** and also new **SELECT** statements. See the documentation of the ODBC driver or OLE DB providerfor more information.
group by	**group by** is a clause used for aggregating (group) several records into one. Within one group, for a certain field, all the records must either have the same value, or the field can only be used from within an expression, e.g. as a sum or an average. The expression based on one or several fields is defined in the expression of the field symbol.
having	**having** is a clause used for qualifying groups in a similar manner to how the **where** clause is used for qualifying records.
order by	**order by** is a clause used for stating the sort order of the resulting table of the **SELECT** statement.
join	**join** is a qualifier stating if several tables are to be joined together into one. Field names and table names must be put within quotes if they contain blank spaces or letters from the national character sets. When the script is automatically generated by Qlik Sense, the quotation mark used is the one preferred by the ODBC driver or OLE DB provider specified in the data source definition of the data source in the **Connect** statement.

Example 1:

```
SELECT * FROM `Categories`;
```

Example 2:

```
SELECT `Category ID`, `Category Name` FROM `Categories`;
```

Example 3:

```
SELECT `Order ID`, `Product ID`,
`Unit Price` * Quantity * (1-Discount) as NetSales
FROM `Order Details`;
```

Example 4:

```
SELECT `Order Details`.`Order ID`,
Sum(`Order Details`.`Unit Price` * `Order Details`.Quantity) as `Result`
FROM `Order Details`, Orders
where Orders.`Order ID` = `Order Details`.`Order ID`
group by `Order Details`.`Order ID`;
```

Set

The **set** statement is used for defining script variables. These can be used for substituting strings, paths, drives, and so on.

Syntax:

```
Set variablename=string
```

Example 1:

```
Set FileToUse=Data1.csv;
```

Example 2:

```
Set Constant="My string";
```

Example 3:

```
Set BudgetYear=2012;
```

Sleep

The **sleep** statement pauses script execution for a specified time.

Syntax:

```
Sleep n
```

Arguments:

Argument	Description
n	Stated in milliseconds, where *n* is a positive integer no larger than 3600000 (i.e. 1 hour). The value may be an expression.

Example 1:

```
Sleep 10000;
```

Example 2:

```
Sleep t*1000;
```

SQL

The **SQL** statement allows you to send an arbitrary SQL command through an ODBC or OLE DB connection.

Syntax:

```
SQL sql_command
```

Sending SQL statements which update the database will return an error if Qlik Sense has opened the ODBC connection in read-only mode.

The syntax:

`SQL SELECT * from tab1;`
is allowed, and is the preferred syntax for **SELECT**, for reasons of consistency. The SQL prefix will, however, remain optional for **SELECT** statements.

Arguments:

Argument	Description
sql_command	A valid SQL command.

Example 1:

```
SQL leave;
```

Example 2:

```
SQL Execute <storedProc>;
```

SQLColumns

The **sqlcolumns** statement returns a set of fields describing the columns of an ODBC or OLE DB data source, to which a **connect** has been made.

Syntax:

```
SQLcolumns
```

The fields can be combined with the fields generated by the **sqltables** and **sqltypes** commands in order to give a good overview of a given database. The twelve standard fields are:

TABLE_QUALIFIER

TABLE_OWNER

TABLE_NAME

COLUMN_NAME

DATA_TYPE

TYPE_NAME

PRECISION

LENGTH

SCALE

RADIX

NULLABLE

REMARKS

For a detailed description of these fields, see an ODBC reference handbook.

Example:

```
Connect to 'MS Access 7.0 Database; DBQ=C:\Course3\DataSrc\QWT.mbd';
SQLcolumns;
```

Some ODBC drivers may not support this command. Some ODBC drivers may produce additional fields.

SQLTables

The **sqltables** statement returns a set of fields describing the tables of an ODBC or OLE DB data source, to which a **connect** has been made.

Syntax:

```
SQLTables
```

The fields can be combined with the fields generated by the **sqlcolumns** and **sqltypes** commands in order to give a good overview of a given database. The five standard fields are:

TABLE_QUALIFIER

TABLE_OWNER

TABLE_NAME

TABLE_TYPE

REMARKS

For a detailed description of these fields, see an ODBC reference handbook.

Example:

```
Connect to 'MS Access 7.0 Database; DBQ=C:\Course3\DataSrc\QWT.mbd';
SQLTables;
```

Some ODBC drivers may not support this command. Some ODBC drivers may produce additional fields.

SQLTypes

The **sqltypes** statement returns a set of fields describing the types of an ODBC or OLE DB data source, to which a **connect** has been made.

Syntax:

```
SQLTypes
```

The fields can be combined with the fields generated by the **sqlcolumns** and **sqltables** commands in order to give a good overview of a given database. The fifteen standard fields are:

TYPE_NAME

DATA_TYPE

PRECISION

LITERAL_PREFIX

LITERAL_SUFFIX

CREATE_PARAMS

NULLABLE

CASE_SENSITIVE

SEARCHABLE

UNSIGNED_ATTRIBUTE

MONEY

AUTO_INCREMENT

LOCAL_TYPE_NAME

MINIMUM_SCALE

MAXIMUM_SCALE

For a detailed description of these fields, see an ODBC reference handbook.

Example:

```
Connect to 'MS Access 7.0 Database; DBQ=C:\Course3\DataSrc\QWT.mbd';
SQLTypes;
```

Some ODBC drivers may not support this command. Some ODBC drivers may produce additional fields.

Star

The string used for representing the set of all the values of a field in the database can be set through the **star** statement. It affects the subsequent **LOAD** and **SELECT** statements.

Syntax:

```
Star is[ string ]
```

Arguments:

Arguments

Argument	Description
string	An arbitrary text. Note that the string must be enclosed by quotation marks if it contains blanks. If nothing is specified, **star is;** is assumed, i.e. there is no star symbol available unless explicitly specified. This definition is valid until a new **star** statement is made.

The **Star is** statement is not recommended for use in the data part of the script (under **Section Application**) if section access is used. The star character is however fully supported for the protected fields in the **Section Access** part of the script. In this case you do not need to use the explicit **Star is** statement since this is always implicit in section access.

Limitations

- You cannot use the star character with key fields; that is, fields that link tables.
- You cannot use the star character with any fields affected by the **Unqualify** statement as this can affect fields that link tables.
- You cannot use the star character with non-logical tables, for example, info-load tables or mapping-load tables.
- When the star character is used in a reducing field (a field that links to the data) in section access , it represents the values listed in this field in section access. It does not represent other values that may exist in the data but are not listed in section access.
- You cannot use the star character with fields affected by any form of data reduction outside the **Section Access** area.

Example

The example below is an extract of a data load script featuring section access.

```
Star is *;

Section Access;
LOAD * INLINE [
ACCESS, USERID, OMIT
ADMIN, ADMIN,
USER, USER1, SALES
USER, USER2, WAREHOUSE
USER, USER3, EMPLOYEES
USER, USER4, SALES
USER, USER4, WAREHOUSE
USER, USER5, *
];

Section Application;
LOAD * INLINE [
SALES, WAREHOUSE, EMPLOYEES, ORDERS
1, 2, 3, 4
];
```

The following applies:

- The *Star* sign is *.
- The user *ADMIN* sees all fields. Nothing is omitted.
- The user *USER1* is not able to see the field *SALES*.
- The user *USER2* is not able to see the field *WAREHOUSE* .
- The user *USER3* cannot see the field *EMPLOYEES*.
- The user *USER4* is added twice to the solution to OMIT two fields for this user, *SALES* and *WAREHOUSE*.
- The *USER5* has a "*" added which means that all listed fields in OMIT are unavailable, that is, user *USER5* cannot see the fields *SALES*, *WAREHOUSE* and *EMPLOYEES* but this user can see the field *ORDERS*.

Store

The **Store** statement creates a QVD, CSV, or text file.

Syntax:

```
Store [ fieldlist from] table into filename [ format-spec ];
```

The statement will create an explicitly named QVD, CSV, or TXT file.

The statement can only export fields from one data table. If fields from several tables are to be exported, an explicit join must be made previously in the script to create the data table that should be exported.

The text values are exported to the CSV file in UTF-8 format. A delimiter can be specified, see **LOAD**. The **store** statement to a CSV file does not support BIFF export.

Arguments:

Store command arguments

Argument	Description
*fieldlist::= (* | field) { , field })*	A list of the fields to be selected. Using * as field list indicates all fields. *field::= fieldname [***as** *aliasname]* *fieldname* is a text that is identical to a field name in *table*. (Note that the field name must be enclosed b straight double quotation marks or square brackets if it contains spaces or other non-standard characters.) *aliasname* is an alternate name for the field to be used in the resulting QVD or CSV file.
table	A script label representing an already loaded table to be used as source for data.
filename	The name of the target file including a valid path to an existing folder data connection. **Example: *'lib://Table Files/target.qvd'*** In legacy scripting mode, the following path formats are also supported: • absolute **Example: *c:\data\sales.qvd*** • relative to the Qlik Sense app working directory. **Example: *data\sales.qvd*** If the path is omitted, Qlik Sense stores the file in the directory specified by the **Directory** statement. If there is no **Directory** statement, Qlik Sense stores the file in the working directory, *C:\Users\{user}\Documents\Qlik\Sense\Apps*.
*format-spec ::=***(** (**txt** | **qvd**) **)**	The format specification consists of the text **txt** for text files, or the text **qvd** for qvd files. If the format specification is omitted, **qvd** is assumed.

Examples:

```
Store mytable into xyz.qvd (qvd);
Store * from mytable into 'lib://FolderConnection/myfile.qvd';
```

```
Store Name, RegNo from mytable into xyz.qvd;
Store Name as a, RegNo as b from mytable into 'lib://FolderConnection/myfile.qvd';
Store mytable into myfile.txt (txt);
Store * from mytable into 'lib://FolderConnection/myfile.qvd';
```

The file extension of DataFiles connections is case sensitive. For example: .qvd.

Table/Tables

The **Table** and **Tables** script keywords are used in **Drop**, **Comment** and **Rename** statements, as well as a format specifier in **Load** statements.

Tag

This script statement provides a way to assign tags to one or more fields or tables. If an attempt to tag a field or table not present in the app is made, the tagging will be ignored. If conflicting occurrences of a field or tag name are found, the last value is used.

Syntax:

```
Tag [field|fields] fieldlist with tagname
```

```
Tag [field|fields] fieldlist using mapname
```

```
Tag table tablelist with tagname
```

Arguments

Argument	Description
fieldlist	One or several fields that should be tagged, in a comma separated list.
mapname	The name of a mapping table previously loaded in a **mapping Load** or **mapping Select** statement.
tablelist	A comma separated list of the tables that should be tagged.
tagname	The name of the tag that should be applied to the field.

Example 1:

```
tagmap:
mapping LOAD * inline [
a,b
Alpha,MyTag
Num,MyTag
];
tag fields using tagmap;
```

Example 2:

```
tag field Alpha with 'MyTag2';
```

Trace

The **trace** statement writes a string to the **Script Execution Progress** window and to the script log file, when used. It is very useful for debugging purposes. Using $-expansions of variables that are calculated prior to the **trace** statement, you can customize the message.

Syntax:

```
Trace string
```

Example 1:

The following statement can be used right after the Load statement that loads the 'Main' table.

```
Trace Main table loaded;
```

This will display the text ‘Main table loaded’ in the script execution dialog and in the log file.

Example 2:

The following statements can be used right after the Load statement that loads the 'Main' table.

```
Let MyMessage = NoOfRows('Main') & ' rows in Main table';
Trace $(MyMessage);
```

This will display a text showing the number of rows in the script execution dialog and in the log file, for example, ‘265,391 rows in Main table’ .

Unmap

The **Unmap** statement disables field value mapping specified by a previous **Map ... Using** statement for subsequently loaded fields.

Syntax:

```
Unmap *fieldlist
```

Arguments:

Arguments

Argument	Description
*fieldlist	a comma separated list of the fields that should no longer be mapped from this point in the script. Using * as field list indicates all fields. The wildcard characters * and ? are allowed in field names. Quoting of field names may be necessary when wildcards are used.

Examples and results:

Example	Result
Unmap Country;	Disables mapping of field Country.
Unmap A, B, C;	Disables mapping of fields A, B and C.
Unmap * ;	Disables mapping of all fields.

Unqualify

The **Unqualify** statement is used for switching off the qualification of field names that has been previously switched on by the **Qualify** statement.

Syntax:

```
Unqualify *fieldlist
```

Arguments:

Arguments

Argument	Description
*fieldlist	A comma separated list of the fields for which qualification should be turned on. Using * as field list indicates all fields. The wildcard characters * and ? are allowed in field names. Quoting of field names may be necessary when wildcards are used. Refer to the documentation for the **Qualify** statement for further information.

Example 1:

In an unfamiliar database, it is often useful to start out by making sure that only one or a few fields are associated, as illustrated in this example:

```
qualify *;
unqualify TransID;
SQL SELECT * from tab1;
SQL SELECT * from tab2;
SQL SELECT * from tab3;
```

First, qualification is turned on for all fields.
Then qualification is turned off for **TransID**.
Only **TransID** will be used for associations between the tables *tab1*, *tab2* and *tab3*. All other fields will be qualified with the table name.

Untag

This script statement provides a way to remove tags from fields or tables. If an attempt to untag a field or table not present in the app is made, the untagging will be ignored.

Syntax:

```
Untag [field|fields] fieldlist with tagname
```

```
Untag [field|fields] fieldlist using mapname
```

```
Untag table tablelist with tagname
```

Arguments:

Arguments

Argument	Description
fieldlist	One or several fields which tags should be removed, in a comma separated list.
mapname	The name of a mapping table previously loaded in a mapping **LOAD** or mapping **SELECT** statement.
tablelist	A comma separated list of the tables that should be untagged.
tagname	The name of the tag that should be removed from the field.

Example 1:

```
tagmap:
mapping LOAD * inline [
a,b
Alpha,MyTag
Num,MyTag
];
Untag fields using tagmap;
```

Example 2:

```
Untag field Alpha with MyTag2;
```

2.6 Working directory

If you are referencing a file in a script statement and the path is omitted, Qlik Sense searches for the file in the following order:

1. The directory specified by a **Directory** statement (only supported in legacy scripting mode).
2. If there is no **Directory** statement, Qlik Sense searches in the working directory.

Qlik Sense Desktop working directory

In Qlik Sense Desktop, the working directory is *C:\Users\{user}\Documents\Qlik\Sense\Apps*.

Qlik Sense working directory

In a Qlik Sense server installation, the working directory is specified in Qlik Sense Repository Service, by default it is *C:\ProgramData\Qlik\Sense\Apps*. See the Qlik Management Console help for more information.

2 Working with variables in the data load editor

A variable in Qlik Sense is a container storing a static value or a calculation, for example a numeric or alphanumeric value. When you use the variable in the app, any change made to the variable is applied everywhere the variable is used. You can define variables in the variables overview, or in the script using the data load editor. You set the value of a variable using **Let** or **Set** statements in the data load script.

You can also work with the Qlik Sense variables from the variables overview when editing a sheet.

2.7 Overview

If the first character of a variable value is an equals sign ' = ' Qlik Sense will try to evaluate the value as a formula (Qlik Sense expression) and then display or return the result rather than the actual formula text.

When used, the variable is substituted by its value. Variables can be used in the script for dollar sign expansion and in various control statements. This is very useful if the same string is repeated many times in the script, for example, a path.

Some special system variables will be set by Qlik Sense at the start of the script execution regardless of their previous values.

2.8 Defining a variable

Variables provide the ability to store static values or the result of a calculation. When defining a variable, use the following syntax:

```
set variablename = string
```

or

```
let variable = expression
```

The **Set** statement is used for string assignment. It assigns the text to the right of the equal sign to the variable. The **Let** statement evaluates an expression to the right of the equal sign at script run time and assigns the result of the expression to the variable.

Variables are case sensitive.

It is not recommended to name a variable identically to a field or a function in Qlik Sense.

Examples:

`set x = 3 + 4;` // the variable will get the string '3 + 4' as the value.

`let x = 3 + 4;` // returns 7 as the value.

`set x = Today(); //` returns 'Today()' as the value.

`let x = Today(); //` returns today's date as the value, for example, ‘9/27/2021’.

2.9 Deleting a variable

If you remove a variable from the script and reload the data, the variable stays in the app. If you want to fully remove the variable from the app, you must also delete the variable from the variables dialog.

2.10 Loading a variable value as a field value

If you want to load a variable value as a field value in a **LOAD** statement and the result of the dollar expansion is text rather than numeric or an expression then you need to enclose the expanded variable in single quotes.

Example:

This example loads the system variable containing the list of script errors to a table. You can note that the expansion of ScriptErrorCount in the **If** clause does not require quotes, while the expansion of ScriptErrorList requires quotes.

```
IF $(ScriptErrorCount) >= 1 THEN
   LOAD '$(ScriptErrorList)' AS Error AutoGenerate 1;
END IF
```

2.11 Variable calculation

There are several ways to use variables with calculated values in Qlik Sense, and the result depends on how you define it and how you call it in an expression.

In this example, we load some inline data:

```
LOAD * INLINE [
    Dim, Sales
    A, 150
    A, 200
    B, 240
    B, 230
    C, 410
    C, 330
];
```

Let's define two variables:

```
Let vSales  = 'Sum(Sales)' ;
Let vSales2  = '=Sum(Sales)' ;
```

In the second variable, we add an equal sign before the expression. This will cause the variable to be calculated before it is expanded and the expression is evaluated.

If you use the vSales variable as it is, for example in a measure, the result will be the string Sum(Sales), that is, no calculation is performed.

If you add a dollar-sign expansion and call $(vSales) in the expression, the variable is expanded, and the sum of Sales is displayed.

Finally, if you call $(vSales2), the variable will be calculated before it is expanded. This means that the result displayed is the total sum of Sales. The difference between using =$(vSales) and =$(vSales2) as measure expressions is seen in this chart showing the results:

Results

Dim	$(vSales)	$(vSales2)
A	350	1560
B	470	1560
C	740	1560

As you can see, $(vSales) results in the partial sum for a dimension value, while $(vSales2) results in the total sum.

The following script variables are available:

- *Error variables (page 211)*
- *Number interpretation variables (page 167)*
- *System variables (page 159)*
- *Value handling variables (page 165)*

2.12 System variables

System variables, some of which are system-defined, provide information about the system and the Qlik Sense app.

System variables overview

Some of the functions are described further after the overview. For those functions, you can click the function name in the syntax to immediately access the details for that specific function.

Floppy

Returns the drive letter of the first floppy drive found, normally *a:*. This is a system-defined variable.

```
Floppy
```

This variable is not supported in standard mode.

CD

Returns the drive letter of the first CD-ROM drive found. If no CD-ROM is found, then *c:* is returned. This is a system-defined variable.

```
CD
```

This variable is not supported in standard mode.

Include

The **Include/Must_Include** variable specifies a file that contains text that should be included in the script and evaluated as script code. It is not used to add data. You can store parts of your script code in a separate text file and reuse it in several apps. This is a user-defined variable.

```
$(Include=filename)
$(Must_Include=filename)
```

HidePrefix

All field names beginning with this text string will be hidden in the same manner as the system fields. This is a user-defined variable.

```
HidePrefix
```

HideSuffix

All field names ending with this text string will be hidden in the same manner as the system fields. This is a user-defined variable.

```
HideSuffix
```

QvPath

Returns the browse string to the Qlik Sense executable. This is a system-defined variable.

```
QvPath
```

This variable is not supported in standard mode.

QvRoot

Returns the root directory of the Qlik Sense executable. This is a system-defined variable.

```
QvRoot
```

This variable is not supported in standard mode.

QvWorkPath

Returns the browse string to the current Qlik Sense app. This is a system-defined variable.

```
QvWorkPath
```

This variable is not supported in standard mode.

QvWorkRoot

Returns the root directory of the current Qlik Sense app. This is a system-defined variable.

```
QvWorkRoot
```

This variable is not supported in standard mode.

StripComments

If this variable is set to 0, stripping of /*..*/ and // comments in the script will be inhibited. If this variable is not defined, stripping of comments will always be performed.

```
StripComments
```

Verbatim

Normally all field values are automatically stripped of leading and trailing blanks (ASCII 32) before being loaded into the Qlik Sense database. Setting this variable to 1 suspends the stripping of blanks. Tab (ASCII 9) and hard space (ANSI 160) characters are never stripped.

```
Verbatim
```

OpenUrlTimeout

This variable defines the timeout in seconds that Qlik Sense should respect when getting data from URL sources (e.g. HTML pages). If omitted, the timeout is about 20 minutes.

```
OpenUrlTimeout
```

WinPath

Returns the browse string to Windows. This is a system-defined variable.

```
WinPath
```

This variable is not supported in standard mode.

WinRoot

Returns the root directory of Windows. This is a system-defined variable.

```
WinRoot
```

This variable is not supported in standard mode.

CollationLocale

Specifies which locale to use for sort order and search matching. The value is the culture name of a locale, for example 'en-US'.This is a system-defined variable.

```
CollationLocale
```

CreateSearchIndexOnReload

This variable defines if search index files should be created during data reload.

```
CreateSearchIndexOnReload
```

CreateSearchIndexOnReload

This variable defines if search index files should be created during data reload.

Syntax:

```
CreateSearchIndexOnReload
```

You can define if search index files should be created during data reload, or if they should be created after the first search request of the user. The benefit of creating search index files during data reload is that you avoid the waiting time experienced by the first user making a search. This needs to be weighed against the longer data reload time required by search index creation.

If this variable is omitted, search index files will not be created during data reload.

> *For session apps, search index files will not be created during data reload, regardless of the setting of this variable.*

Example 1: Create search index fields during data reload

```
set CreateSearchIndexOnReload=1;
```

Example 2: Create search index fields after first search request

```
set CreateSearchIndexOnReload=0;
```

HidePrefix

All field names beginning with this text string will be hidden in the same manner as the system fields. This is a user-defined variable.

Syntax:

```
HidePrefix
```

Example:

```
set HidePrefix='_' ;
```

If this statement is used, the field names beginning with an underscore will not be shown in the field name lists when the system fields are hidden.

HideSuffix

All field names ending with this text string will be hidden in the same manner as the system fields. This is a user-defined variable.

Syntax:

```
HideSuffix
```

Example:

```
set HideSuffix='%';
```

If this statement is used, the field names ending with a percentage sign will not be shown in the field name lists when the system fields are hidden.

Include

The **Include/Must_Include** variable specifies a file that contains text that should be included in the script and evaluated as script code. It is not used to add data. You can store parts of your script code in a separate text file and reuse it in several apps. This is a user-defined variable.

> *This variable supports only folder data connections in standard mode.*

Syntax:

```
$(Include=filename)
```

```
$(Must_Include=filename)
```

There are two versions of the variable:

- **Include** does not generate an error if the file cannot be found, it will fail silently.
- **Must_Include** generates an error if the file cannot be found.

If you don't specify a path, the filename will be relative to the Qlik Sense app working directory. You can also specify an absolute file path, or a path to a lib:// folder connection. Do not put a space character before or after the equal sign.

> *The construction **set Include =**filename is not applicable.*

Examples:

```
$(Include=abc.txt);
```

```
$(Must_Include=lib://DataFiles/abc.txt);
```

Limitations

Limited cross-compatibility between UTF-8 encoded files under Windows versus Linux.

It is optional to use UTF-8 with BOM (Byte Order Mark). BOM can interfere with the use of UTF-8 in software that does not expect non-ASCII bytes at the start of a file, but that could otherwise handle the text stream.

- Windows systems use BOM in UTF-8 to identify that a file is UTF-8 encoded, despite the fact that there is no ambiguity in the byte storage.

- Unix/Linux use UTF-8 for Unicode, but does not use the BOM as this interferes with the syntax for command files.

This has some implications for Qlik Sense.

- In Windows any file that begins with an UTF-8 BOM is considered a UTF-8 script file. Otherwise ANSI encoding is assumed.
- In Linux, the system default 8 bit code page is UTF-8. This is why the UTF-8 works although it does not contain a BOM.

As a result, portability cannot be guaranteed. It is not always possible to create a file on Windows that can be interpreted by Linux and vice versa. There is no cross compatibility between the two systems regarding UTF-8 encoded files due to different handling of the BOM.

OpenUrlTimeout

This variable defines the timeout in seconds that Qlik Sense should respect when getting data from URL sources (e.g. HTML pages). If omitted, the timeout is about 20 minutes.

Syntax:

OpenUrlTimeout

Example:

```
set OpenUrlTimeout=10;
```

StripComments

If this variable is set to 0, stripping of /*..*/ and // comments in the script will be inhibited. If this variable is not defined, stripping of comments will always be performed.

Syntax:

StripComments

Certain database drivers use /*..*/ as optimization hints in **SELECT** statements. If this is the case, the comments should not be stripped before sending the **SELECT** statement to the database driver.

It is recommended that this variable be reset to 1 immediately after the statement(s) where it is needed.

Example:

```
set StripComments=0;
SQL SELECT * /* <optimization directive> */ FROM Table ;
set StripComments=1;
```

Verbatim

Normally all field values are automatically stripped of leading and trailing blanks (ASCII 32) before being loaded into the Qlik Sense database. Setting this variable to 1 suspends the stripping of blanks. Tab (ASCII 9) and hard space (ANSI 160) characters are never stripped.

Syntax:

```
Verbatim
```

Example:

```
set Verbatim = 1;
```

2.13 Value handling variables

This section describes variables that are used for handling NULL and other values.

Value handling variables overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

NullDisplay

The defined symbol will substitute all NULL values from ODBC, and connectors, on the lowest level of data. This is a user-defined variable.

```
NullDisplay
```

NullInterpret

The defined symbol will be interpreted as NULL when it occurs in a text file, Excel file or an inline statement. This is a user-defined variable.

```
NullInterpret
```

NullValue

If the **NullAsValue** statement is used, the defined symbol will substitute all NULL values in the **NullAsValue** specified fields with the specified string.

```
NullValue
```

OtherSymbol

Defines a symbol to be treated as 'all other values' before a **LOAD/SELECT** statement. This is a user-defined variable.

```
OtherSymbol
```

NullDisplay

The defined symbol will substitute all NULL values from ODBC, and connectors, on the lowest level of data. This is a user-defined variable.

Syntax:

```
NullDisplay
```

Example:

```
set NullDisplay='<NULL>';
```

NullInterpret

The defined symbol will be interpreted as NULL when it occurs in a text file, Excel file or an inline statement. This is a user-defined variable.

Syntax:

```
NullInterpret
```

Examples:

```
set NullInterpret=' ';
set NullInterpret =;
```

will not return NULL values for blank values in Excel, but it will for a CSV text file.

```
set NullInterpret ='';
```

will return NULL values for blank values in Excel.

NullValue

If the **NullAsValue** statement is used, the defined symbol will substitute all NULL values in the **NullAsValue** specified fields with the specified string.

Syntax:

```
NullValue
```

Example:

```
NullAsValue Field1, Field2;
set NullValue='<NULL>';
```

OtherSymbol

Defines a symbol to be treated as 'all other values' before a **LOAD/SELECT** statement. This is a user-defined variable.

Syntax:

```
OtherSymbol
```

Example:

```
set OtherSymbol='+';
LOAD * inline
[X, Y
a, a
b, b];
LOAD * inline
[X, Z
a, a
+, c];
```

The field value Y='b' will now link to Z='c' through the other symbol.

2.14 Number interpretation variables

Number interpretation variables are system defined. The variables are included at the top of the load script and apply number formatting settings at the time of the script execution. They can be deleted, edited, or duplicated.

Number interpretation variables are automatically generated according to the current regional settings of the operating system when a new app is created. In Qlik Sense Desktop, this is according to the settings of the computer operating system. In Qlik Sense, it is according to the operating system of the server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Currency formatting

MoneyDecimalSep

The decimal separator defined replaces the decimal symbol for currency set by your regional settings.

```
MoneyDecimalSep
```

MoneyFormat

The symbol defined replaces the currency symbol set by your regional settings.

```
MoneyFormat
```

MoneyThousandSep

The thousands separator defined replaces the digit grouping symbol for currency set by your regional settings.

```
MoneyThousandSep
```

Number formatting

DecimalSep

The decimal separator defined replaces the decimal symbol set by your regional settings.

```
DecimalSep
```

ThousandSep

The thousands separator defined replaces the digit grouping symbol of the operating system (regional settings).

```
ThousandSep
```

NumericalAbbreviation

The numerical abbreviation sets which abbreviation to use for scale prefixes of numerals, for example M for mega or a million (10^6), and µ for micro (10^{-6}).

```
NumericalAbbreviation
```

Time formatting

DateFormat

This environment variable defines the date format used as the default in the app. The format is used both to interpret and format dates. If the variable is not defined, the date format of the regional settings of the operating system will be fetched when the script runs.

```
DateFormat
```

TimeFormat

The format defined replaces the time format of the operating system (regional settings).

```
TimeFormat
```

TimestampFormat

The format defined replaces the date and time formats of the operating system (regional settings).

```
TimestampFormat
```

MonthNames

The format defined replaces the month names convention of the regional settings.

```
MonthNames
```

LongMonthNames

The format defined replaces the long month names convention in the regional settings.

```
LongMonthNames
```

DayNames

The format defined replaces the weekday names convention set by your regional settings.

```
DayNames
```

LongDayNames

The format defined replaces the long weekday names convention in the regional settings.

```
LongDayNames
```

FirstWeekDay

Integer that defines which day to use as the first day of the week.

```
FirstWeekDay
```

BrokenWeeks

This setting defines if weeks are broken or not.

```
BrokenWeeks
```

ReferenceDay

The setting defines which day in January to set as reference day to define week 1.

```
ReferenceDay
```

FirstMonthOfYear

The setting defines which month to use as first month of the year, which can be used to define financial years that use a monthly offset, for example starting April 1.

> *This setting is currently unused but reserved for future use.*

Valid settings are 1 (January) to 12 (December). Default setting is 1.

Syntax:

```
FirstMonthOfYear
```

Example:

```
Set FirstMonthOfYear=4; //Sets the year to start in April
```

BrokenWeeks

This setting defines if weeks are broken or not.

Syntax:

```
BrokenWeeks
```

By default, Qlik Sense functions use unbroken weeks. This means that:

- In some years, week 1 starts in December, and in other years, week 52 or 53 continues into January.
- Week 1 always has at least 4 days in January.

The alternative is to use broken weeks:

- Week 52 or 53 do not continue into January.
- Week 1 starts on January 1 and is, in most cases, not a full week.

The following values can be used:

- 0 (=use unbroken weeks)
- 1 (= use broken weeks)

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Examples:

```
Set BrokenWeeks=0; //(use unbroken weeks)
Set BrokenWeeks=1; //(use broken weeks)
```

DateFormat

This environment variable defines the date format used as the default in the app and by date returning functions like `date()` and `date#()`. The format is used to interpret and format dates. If the variable is not defined, the date format set by your regional settings is fetched when the script runs.

Syntax:

```
DateFormat
```

DateFormat Function examples

Example	Result
`Set DateFormat='M/D/YY'; //(US format)`	This use of the `DateFormat` function defines the date as the US format, month/day/year.
`Set DateFormat='DD/MM/YY'; //(UK date format)`	This use of the `DateFormat` function defines the date as the UK format, day/month/year.
`Set DateFormat='YYYY/MM/DD'; //(ISO date format)`	This use of the `DateFormat` function defines the date as the ISO format, year/month/day.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 – System variables default

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates.
- The `DateFormat` function, which will use the US date format.

In this example, a dataset is loaded into a table named '`Transactions`'. It includes a `date` field. The US `DateFormat` definition is used. This pattern will be used for implicit text to date conversion when the text dates are loaded.

Load script

```
Set DateFormat='MM/DD/YYYY';

Transactions:
LOAD
date,
month(date) as month,
id,
amount
INLINE
[
date,id,amount
01/01/2022,1,1000
02/01/2022,2,2123
03/01/2022,3,4124
04/01/2022,4,2431
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `month`

Create this measure:

```
=sum(amount)
```

Results table

date	month	=sum(amount)
01/01/2022	Jan	1000
02/01/2022	Feb	2123
03/01/2022	Mar	4124
04/01/2022	Apr	2431

The `DateFormat` definition MM/DD/YYYY is used for implicit conversion of text to dates, which is why the `date` field is properly interpreted as a date. The same format is used to display the date, as shown in the results table.

Example 2 – Change system variable

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset from the previous example.
- The `DateFormat` function, which will use the 'DD/MM/YYYY' format.

Load script

```
SET DateFormat='DD/MM/YYYY';
Transactions:
LOAD
date,
month(date) as month,
id,
amount
INLINE
[
date,id,amount
01/01/2022,1,1000
02/01/2022,2,2123
03/01/2022,3,4124
04/01/2022,4,2431
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `month`

Create this measure:

```
=sum(amount)
```

Results table

date	month	=sum(amount)
01/01/2022	Jan	1000
02/01/2022	Jan	2123
03/01/2022	Jan	4124
04/01/2022	Jan	2431

Because the `DateFormat` definition was set to 'DD/MM/YYYY', you can see that the two digits after the first "/" symbol have been interpreted as the month, resulting in all records being from the month of January.

Example 3 – Date interpretation

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset with dates in numerical format.
- The `DateFormat` variable, which will use the 'DD/MM/YYYY' format.
- The `date()` variable.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
Load
date(numerical_date),
month(date(numerical_date)) as month,
id,
amount
Inline
[
numerical_date,id,amount
43254,1,1000
43255,2,2123
43256,3,4124
43258,4,2431
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- date
- month

Create this measure:

```
=sum(amount)
```

Results table

date	month	=sum(amount)
06/03/2022	Jun	1000
06/04/2022	Jun	2123
06/05/2022	Jun	4124
06/07/2022	Jun	2431

In the load script, you use the `date()` function to convert the numerical date into a date format. Because you do not provide a specified format as a second argument in the function, the `DateFormat` is used. This results in the date field using the format 'MM/DD/YYYY'.

Example 4 – Foreign date formatting

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates.
- The `DateFormat` variable, which uses the 'DD/MM/YYYY' format but is uncommented by forward slashes.

Load script

```
// SET DateFormat='DD/MM/YYYY';


Transactions:
Load
date,
month(date) as month,
id,
amount
Inline
[
date,id,amount
22-05-2022,1,1000
23-05-2022,2,2123
24-05-2022,3,4124
```

```
25-05-2022,4,2431
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `month`

Create this measure:

```
=sum(amount)
```

Results table

date	month	=sum(amount)
22-05-2022	-	1000
23-05-2022	-	2123
24-05-2022	-	4124
25-05-2022	-	2431

In the initial load script, the `DateFormat` being used is the default 'MM/DD/YYYY'. Because the `date` field in the transactions dataset is not in this format, the field is not interpreted as a date. This is shown in the results table where the `month` field values are null.

You can verify the interpreted data types in the Data model viewer by inspecting the `date` field's "Tags" properties:

Preview of the `Transactions` table. Note the "Tags" for the `date` field indicating that the textual input data has not been implicitly converted to a date/timestamp.

date

Density	100%
Subset ratio	100%
Has duplicates	false
Total distinct values	4
Present distinct values	4
Non-null values	4
Tags	$ascii $text

Transactions

date	month	id	amount
22-05-2022	-	1	1000
23-05-2022	-	2	2123
24-05-2022	-	3	4124
25-05-2022	-	4	2431

This can be solved by enabling the `DateFormat` system variable:

```
// SET DateFormat='DD/MM/YYYY';
```

Remove the double forward slashes and reload the data.

Preview of the `Transactions` table. Note the "Tags" for the `date` field indicating that the textual input data has been implicitly converted to a date/timestamp.

date	
Density	100%
Subset ratio	100%
Has duplicates	false
Total distinct values	4
Present distinct values	4
Non-null values	4
Tags	$numeric $integer $timestamp $date

Transactions

date	month	id	amount
22-05-2022	May	1	1000
23-05-2022	May	2	2123
24-05-2022	May	3	4124
25-05-2022	May	4	2431

DayNames

The format defined replaces the weekday names convention set by your regional settings.

Syntax:

`DayNames`

When modifying the variable, a semicolon `;` is required to separate the individual values.

DayName Function examples

Function example	Result definition
`Set DayNames='Mon;Tue;Wed;Thu;Fri;Sat;Sun';`	This use of the DayNames function defines day names in their abbreviated form.
`Set DayNames='M;Tu;W;Th;F;Sa;Su';`	This use of the DayNames function defines day names by their first letters.

The `DayNames` function is often used in combination with the following functions:

Related functions

Function	Interaction
weekday (page 705)	Script function to return `DayNames` as field values .
Date (page 806)	Script function to return `DayNames` as field values.
LongDayNames (page 186)	Long form values of `DayNames`.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 - System variables default

Load script and results

Overview

In this example, the dates in the dataset are set in the MM/DD/YYYY format.

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset with dates, which will be loaded into a table named, `Transactions`.
- A `date` field.
- The default `DayNames` definition.

Load script

```
SET DayNames='Mon;Tue;Wed;Thu;Fri;Sat;Sun';

Transactions:
LOAD
date,
WeekDay(date) as dayname,
id,
amount
INLINE
[
date,id,amount
01/01/2022,1,1000
02/01/2022,2,2123
03/01/2022,3,4124
04/01/2022,4,2431
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `dayname`

Create this measure:

```
sum(amount)
```

Results table

date	dayname	sum(amount)
01/01/2022	Sat	1000
02/01/2022	Tue	2123
03/01/2022	Tue	4124
04/01/2022	Fri	2431

In the load script, the `WeekDay` function is used with the `date` field as the provided argument. In the results table, the output of this `WeekDay` function displays the days of the week in the format of the `DayNames` definition.

Example 2 - Change system variable

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab. The same dataset and scenario from the first example are used.

However, at the start of the script, the `DayNames` definition is modified to use the abbreviated days of the week in Afrikaans.

Load script

```
SET DayNames='Ma;Di;Wo;Do;Vr;Sa;So';

Transactions:
Load
date,
WeekDay(date) as dayname,
id,
amount
Inline
[
date,id,amount
01/01/2022,1,1000
02/01/2022,2,2123
03/01/2022,3,4124
04/01/2022,4,2431
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `dayname`

Create this measure:

```
sum(amount)
```

Results table

date	dayname	sum(amount)
01/01/2022	Sa	1000
02/01/2022	Di	2123
03/01/2022	Di	4124
04/01/2022	Vr	2431

In the results table, the output of this `WeekDay` function displays the days of the week in the format of the `DayNames` definition.

It is important to remember that if the language for the `DayNames` is modified like it has been in this example, the `LongDayNames` would still contain the days of the week in English. This would need to be modified as well if both variables are used in the application.

Example 3 – Date function

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset with dates, which will be loaded into a table named, `Transactions`.
- A `date` field.
- The default `DayNames` definition.

Load script

```
SET DayNames='Mon;Tue;Wed;Thu;Fri;Sat;Sun';

Transactions:
Load
date,
Date(date,'WWW') as dayname,
id,
amount
Inline
[
date,id,amount
01/01/2022,1,1000
02/01/2022,2,2123
03/01/2022,3,4124
04/01/2022,4,2431
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `dayname`

Create this measure:

`sum(amount)`

Results table

date	dayname	sum(amount)
01/01/2022	Sat	1000
02/01/2022	Tue	2123
03/01/2022	Tue	4124
04/01/2022	Fri	2431

The default `DayNames`definition is used. In the load script, the `Date` function is used with the `date` field as the first argument. The second argument is `WWW`. This formatting converts the result into the values stored in the `DayNames` definition. This is displayed in the output of the results table.

DecimalSep

The decimal separator defined replaces the decimal symbol set by your regional settings.

Qlik Sense automatically interprets text as numbers whenever a recognizable number pattern is encountered. The `ThousandSep` and `DecimalSep` system variables determine the makeup of the patterns applied when parsing text as numbers. The `ThousandSep` and `DecimalSep` variables set the default number format pattern when visualizing numeric content in front-end charts and tables. That is, it directly impacts the **Number formatting** options for any front end expression.

Assuming a thousand separator of comma '`,`' and a decimal separator of '`.`', these are examples of patterns that would be implicitly converted to numeric equivalent values:

`0,000.00`

`0000.00`

`0,000`

These are examples of patterns that would remain unchanged as text; that is, not converted to numeric:

`0.000,00`

`0,00`

Syntax:

`DecimalSep`

Function examples

Example	Result
`Set DecimalSep='.';`	Sets '.' as the decimal separator.
`Set DecimalSep=',';`	Sets ',' as the decimal separator.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example – Effect of setting number separator variables on different input data

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of sums and dates with the sums set in different format patterns.
- A table named `Transactions`.
- The `DecimalSep` variable which is set to '`.`'.
- The `ThousandSep` variable which is set to '`,`'.
- The `delimiter` variable that is set as the '|' character to separate the different fields in a line.

Load script

```
Set ThousandSep=',';
Set DecimalSep='.';

Transactions:
Load date,
id,
amount as amount
Inline
[
date|id|amount
```

```
01/01/2022|1|1.000-45
01/02/2022|2|23.344
01/03/2022|3|4124,35
01/04/2022|4|2431.36
01/05/2022|5|4,787
01/06/2022|6|2431.84
01/07/2022|7|4132.5246
01/08/2022|8|3554.284
01/09/2022|9|3.756,178
01/10/2022|10|3,454.356
] (delimiter is '|');
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension: `amount`.

Create this measure:

```
=sum(amount)
```

Results table

Amount	**=Sum(amount)**
Totals	20814.7086
1.000-45	
3.756,178	
4124,35	
23.344	23.344
2431.36	2431.36
2431.84	2431.84
3,454.356	3454.356
3554.284	3554.284
4132.5246	4132.5246
4,787	4787

Any value not interpreted as number remains as text and is aligned to the left by default. Any successfully converted values are aligned to the right, retaining the original input format.

The expression column shows the numeric equivalent, which is by default formatted with only a decimal separator '`.`'. This can be overridden with the **Number formatting** drop down setting in the expression configuration.

FirstWeekDay

Integer that defines which day to use as the first day of the week.

Syntax:

```
FirstWeekDay
```

By default, the Qlik Sense system variables define `FirstWeekDay=6`. This means that Sunday is the first day of the week.

Values that can be set for FirstWeekDay

Value	Day
0	Monday
1	Tuesday
2	Wednesday
3	Thursday
4	Friday
5	Saturday
6	Sunday

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 – Using default value (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

In this example, the load script uses the default Qlik Sense system variable value, `FirstWeekDay=6`. This data contains data for the first 14 days in 2020.

Load script

```
// Example 1: Load Script using the default value of FirstWeekDay=6, i.e. Sunday

SET FirstWeekDay = 6;

Sales:
```

```
LOAD
        date,
        sales,
        week(date) as week,
        weekday(date) as weekday
Inline [
date,sales
01/01/2021,6000
01/02/2021,3000
01/03/2021,6000
01/04/2021,8000
01/05/2021,5000
01/06/2020,7000
01/07/2020,3000
01/08/2020,5000
01/09/2020,9000
01/10/2020,5000
01/11/2020,7000
01/12/2020,7000
01/13/2020,7000
01/14/2020,7000
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `week`
- `weekday`

Results table

Date	week	weekday
01/01/2021	1	Wed
01/02/2021	1	Thu
01/03/2021	1	Fri
01/04/2021	1	Sat
01/05/2021	2	Sun
01/06/2020	2	Mon
01/07/2020	2	Tue
01/08/2020	2	Wed
01/09/2020	2	Thu
01/10/2020	2	Fri
01/11/2020	2	Sat

Date	week	weekday
01/12/2020	3	Sun
01/13/2020	3	Mon
01/14/2020	3	Tue

Because the default settings are being used, the `FirstWeekDay` system variable is set to 6. In the results table, each new week can be seen beginning on Sunday (the 5th and 12th of January).

Example 2 – Changing the FirstWeekDay variable (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

In this example, the data contains the first 14 days in 2020. At the start of the script, we set the `FirstWeekDay` variable to 3.

Load script

```
// Example 2: Load Script setting the value of FirstWeekDay=3, i.e. Thursday

SET FirstWeekDay = 3;

Sales:
LOAD
       date,
       sales,
       week(date) as week,
       weekday(date) as weekday
Inline [
date,sales
01/01/2021,6000
01/02/2021,3000
01/03/2021,6000
01/04/2021,8000
01/05/2021,5000
01/06/2020,7000
01/07/2020,3000
01/08/2020,5000
01/09/2020,9000
01/10/2020,5000
01/11/2020,7000
01/12/2020,7000
01/13/2020,7000
01/14/2020,7000
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `week`
- `weekday`

Results table

Date	week	weekday
01/01/2021	52	Wed
01/02/2021	1	Thu
01/03/2021	1	Fri
01/04/2021	1	Sat
01/05/2021	1	Sun
01/06/2020	1	Mon
01/07/2020	1	Tue
01/08/2020	1	Wed
01/09/2020	2	Thu
01/10/2020	2	Fri
01/11/2020	2	Sat
01/12/2020	2	Sun
01/13/2020	2	Mon
01/14/2020	2	Tue

Because the `FirstWeekDay` system variable is set to 3, the first day of each week will be a Thursday. In the results table, each new week can be seen beginning on Thursday (the 2nd and 9th of January).

LongDayNames

The format defined replaces the long weekday names convention in the regional settings.

Syntax:

`LongDayNames`

The following example of the `LongDayNames` function defines day names in full:

```
Set LongDayNames='Monday;Tuesday;Wednesday;Thursday;Friday;Saturday;Sunday';
```

When modifying the variable, a semicolon `;` is required to separate the individual values.

The `LongDayNames` function can be used in combination with the *Date (page 806)* function which returns `DayNames` as field values.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 - System variable default

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset with dates, which will be loaded into a table named, `Transactions`.
- A `date` field.
- The default `LongDayNames` definition.

Load script

```
SET LongDayNames='Monday;Tuesday;Wednesday;Thursday;Friday;Saturday;Sunday';

Transactions:
LOAD
date,
Date(date,'WWWW') as dayname,
id,
amount
INLINE
[
date,id,amount
01/01/2022,1,1000
02/01/2022,2,2123
03/01/2022,3,4124
04/01/2022,4,2431
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- date
- dayname

Create this measure:

```
=sum(amount)
```

Results table

date	dayname	=sum(amount)
01/01/2022	Saturday	1000
02/01/2022	Tuesday	2123
03/01/2022	Tuesday	4124
04/01/2022	Friday	2431

In the load script, to create a field called, dayname, the Date function is used with the date field as the first argument. The second argument in the function is the formatting WWWW.

Using this formatting converts the values from the first argument into the corresponding full day name that is set in the variable LongDayNames. In the results table, the field values of our created field dayname display this.

Example 2 – Change system variable

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The same dataset and scenario from the first example are used. However, at the start of the script, the LongDayNames definition is modified to use the days of the week in Spanish.

Load Script

```
SET LongDayNames='Lunes;Martes;Miércoles;Jueves;Viernes;Sábado;Domingo';

Transactions:
LOAD
date,
Date(date,'WWWW') as dayname,
id,
amount
INLINE
[
date,id,amount
01/01/2022,1,1000
02/01/2022,2,2123
03/01/2022,3,4124
04/01/2022,4,2431
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `dayname`

Create this measure:

```
=sum(amount)
```

Results table

date	dayname	=sum(amount)
01/01/2022	Sábado	1000
02/01/2022	Martes	2123
03/01/2022	Martes	4124
04/01/2022	Viernes	2431

In the load script, the `LongDayNames` variable is modified to list the days of the week in Spanish.

Then, you create a field called, `dayname`, which is the `Date` function used with the `date` field as the first argument.

The second argument in the function is the formatting `WWWW`. By using this formatting Qlik Sense converts the values from the first argument into the corresponding full day name set in the variable `LongDayNames`.

In the results table, the field values of our created field `dayname` displays the days of the week written in Spanish and in full.

LongMonthNames

The format defined replaces the long month names convention in the regional settings.

Syntax:

`LongMonthNames`

When modifying the variable, the `;` needs to be used to separate the individual values.

The following example of the LongMonthNames function defines month names in full:

```
Set
LongMonthNames='January;February;March;April;May;June;July;August;September;October;November;D
ecember';
```

The `LongMonthNames` function is often used in combination with the following functions:

Related functions

Function	Interaction
Date (page 806)	Script function to return `DayNames`as field values.
LongDayNames (page 186)	Long form values of `DayNames`.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 - System variables default

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates that is loaded into a table named `Transactions`.
- A `date` field.
- The default `LongMonthNames` definition.

Load script

```
SET
LongMonthNames='January;February;March;April;May;June;July;August;September;October;November;D
ecember';

Transactions:
Load
date,
Date(date,'MMMM') as monthname,
id,
amount
Inline
[
date,id,amount
01/01/2022,1,1000.45
01/02/2022,2,2123.34
01/03/2022,3,4124.35
01/04/2022,4,2431.36
01/05/2022,5,4787.78
01/06/2022,6,2431.84
01/07/2022,7,2854.83
01/08/2022,8,3554.28
01/09/2022,9,3756.17
```

```
01/10/2022,10,3454.35
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `monthname`

Create this measure:

```
=sum(amount)
```

Results table

date	monthname	sum(amount)
01/01/2022	January	1000.45
01/02/2022	January	2123.34
01/03/2022	January	4124.35
01/04/2022	January	2431.36
01/05/2022	January	4787.78
01/06/2022	January	2431.84
01/07/2022	January	2854.83
01/08/2022	January	3554.28
01/09/2022	January	3756.17
01/10/2022	January	3454.35

The default `LongMonthNames` definition is used. In the load script, to create a field called, `month`, the `Date` function is used with the `date` field as the first argument. The second argument in the function is the formatting `MMMM`.

Using this formatting Qlik Sense converts the values from the first argument into the corresponding full month name set in the variable `LongMonthNames`. In the results table, the field values of our created field `month` display this.

Example 2 - Change system variable

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates that is loaded into a table named `Transactions`.
- A `date` field.
- The `LongMonthNames` variable that is modified to use the abbreviated days of the week in Spanish.

Load script

```
SET
LongMonthNames='Enero;Febrero;Marzo;Abril;Mayo;Junio;Julio;Agosto;Septiembre;OctubreNoviembre;
Diciembre';

Transactions:
LOAD
date,
Date(date,'MMMM') as monthname,
id,
amount
INLINE
[
date,id,amount
01/01/2022,1,1000
02/01/2022,2,2123
03/01/2022,3,4124
04/01/2022,4,2431
];
```

Results

Load the data and open a sheet. Create a new table and add `sum(amount)` as a measure and these fields as dimensions:

- `date`
- `monthname`

Create this measure:

```
=sum(amount)
```

Results table

date	monthname	sum(amount)
01/01/2022	Enero	1000.45
01/02/2022	Enero	2123.34
01/03/2022	Enero	4124.35
01/04/2022	Enero	2431.36
01/05/2022	Enero	4787.78
01/06/2022	Enero	2431.84
01/07/2022	Enero	2854.83

date	monthname	sum(amount)
01/08/2022	Enero	3554.28
01/09/2022	Enero	3756.17
01/10/2022	Enero	3454.35

In the load script, the `LongMonthNames` variable is modified to list the months of the year in Spanish. Then, to create a field called, `monthname`, the`Date` function is used with the `date` field as the first argument. The second argument in the function is the formatting `MMMM`.

Using this formatting Qlik Sense converts the values from the first argument into the corresponding full month name set in the variable `LongMonthNames`. In the results table, the field values of our created field `monthname` display the month name written in Spanish.

MoneyDecimalSep

The decimal separator defined replaces the decimal symbol for currency set by your regional settings.

Syntax:

MoneyDecimalSep

Example:

```
Set MoneyDecimalSep='.';
```

MoneyFormat

The symbol defined replaces the currency symbol set by your regional settings.

Syntax:

MoneyFormat

Example:

```
Set MoneyFormat='$ #,##0.00; ($ #,##0.00)';
```

MoneyThousandSep

The thousands separator defined replaces the digit grouping symbol for currency set by your regional settings.

Syntax:

MoneyThousandSep

Example:

```
Set MoneyDecimalSep=',';
```

MonthNames

The format defined replaces the month names convention of the regional settings.

Syntax:

MonthNames

When modifying the variable, the ; needs to be used to separate the individual values.

Function examples

Example	Results
`Set MonthNames='Jan;Feb;Mar;Apr;May;Jun;Jul;Aug;Sep;Oct;Nov;Dec';`	This use of the `MonthNames` function defines month names in English and their abbreviated form.
`Set MonthNames='Enero;Feb;Marzo;Abr;Mayo;Jun;Jul;Agosto;Set;Oct;Nov;Dic';`	This use of the `MonthNames` function defines month names in Spanish and their abbreviated form.

The `MonthNames` function can be used in combination with the following functions:

Related functions

Function	Interaction
month (page 655)	Script function to return values defined in `MonthNames` as field values
Date (page 806)	Script function to return values defined in `MonthNames` as field values based on a formatting argument provided
LongMonthNames (page 189)	Long form values of `MonthNames`

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 – System variables default

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates that is loaded into a table named `Transactions`.
- A `date` field.
- The default `MonthNames` definition.

Load script

```
SET MonthNames='Jan;Feb;Mar;Apr;May;Jun;Jul;Aug;Sep;Oct;Nov;Dec';

Transactions:
LOAD
date,
Month(date) as monthname,
id,
amount
INLINE
[
date,id,amount
01/01/2022,1,1000.45
01/02/2022,2,2123.34
01/03/2022,3,4124.35
01/04/2022,4,2431.36
01/05/2022,5,4787.78
01/06/2022,6,2431.84
01/07/2022,7,2854.83
01/08/2022,8,3554.28
01/09/2022,9,3756.17
01/10/2022,10,3454.35
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `monthname`

Create this measure:

```
=sum(amount)
```

Results table

date	monthname	sum(amount)
01/01/2022	Jan	1000.45

date	monthname	sum(amount)
01/02/2022	Jan	2123.34
01/03/2022	Jan	4124.35
01/04/2022	Jan	2431.36
01/05/2022	Jan	4787.78
01/06/2022	Jan	2431.84
01/07/2022	Jan	2854.83
01/08/2022	Jan	3554.28
01/09/2022	Jan	3756.17
01/10/2022	Jan	3454.35

The default `MonthNames` definition is used. In the load script, the `Month` function is used with the `date` field as the provided argument.

In the results table, the output of this `Month` function displays the months of the year in the format of the `MonthNames` definition.

Example 2 - Change system variable

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates that is loaded into a table named `Transactions`.
- A `date` field.
- The `MonthNames` variable that is modified to use the abbreviated months in Spanish.

Load script

```
Set MonthNames='Enero;Feb;Marzo;Abr;Mayo;Jun;Jul;Agosto;Set;Oct;Nov;Dic';

Transactions:
LOAD
date,
month(date) as month,
id,
amount
INLINE
[
date,id,amount
01/01/2022,1,1000
02/01/2022,2,2123
03/01/2022,3,4124
```

```
04/01/2022,4,2431
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `monthname`

Create this measure:

```
=sum(amount)
```

Results table

date	monthname	sum(amount)
01/01/2022	Enero	1000.45
01/02/2022	Enero	2123.34
01/03/2022	Enero	4124.35
01/04/2022	Enero	2431.36
01/05/2022	Enero	4787.78
01/06/2022	Enero	2431.84
01/07/2022	Enero	2854.83
01/08/2022	Enero	3554.28
01/09/2022	Enero	3756.17
01/10/2022	Enero	3454.35

In the load script, first the `MonthNames` variable is modified to list the months of the year abbreviated in Spanish. The `Month` function is used with the `date` field as the provided argument.

In the results table, the output of this `Month` function displays the months of the year in the format of the `MonthNames` definition.

It is important to remember that if the language for the `MonthNames` variable is modified like it has been in this example, the `LongMonthNames` variable would still contain the days of the week in English. The `LongMonthNames` variable would have to be modified if both variables are used in the application.

Example 3 – Date function

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates that is loaded into a table named `Transactions`.
- A `date` field.
- The default `MonthNames` definition.

Load script

```
SET MonthNames='Jan;Feb;Mar;Apr;May;Jun;Jul;Aug;Sep;Oct;Nov;Dec';

Transactions:
LOAD
date,
Month(date, 'MMM') as monthname,
id,
amount
INLINE
[
date,id,amount
01/01/2022,1,1000.45
01/02/2022,2,2123.34
01/03/2022,3,4124.35
01/04/2022,4,2431.36
01/05/2022,5,4787.78
01/06/2022,6,2431.84
01/07/2022,7,2854.83
01/08/2022,8,3554.28
01/09/2022,9,3756.17
01/10/2022,10,3454.35
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `monthname`

Create this measure:

```
=sum(amount)
```

Results table

date	monthname	sum(amount)
01/01/2022	Jan	1000.45
01/02/2022	Jan	2123.34
01/03/2022	Jan	4124.35
01/04/2022	Jan	2431.36
01/05/2022	Jan	4787.78

date	monthname	sum(amount)
01/06/2022	Jan	2431.84
01/07/2022	Jan	2854.83
01/08/2022	Jan	3554.28
01/09/2022	Jan	3756.17
01/10/2022	Jan	3454.35

The default `MonthNames` definition is used. In the load script, the `Date` function is used with the `date` field as the first argument. The second argument is `MMM`.

Using this formatting Qlik Sense converts the values from the first argument into the corresponding month name set in the variable `MonthNames`. In the results table, the field values of our created field `month` display this.

NumericalAbbreviation

The numerical abbreviation sets which abbreviation to use for scale prefixes of numerals, for example M for mega or a million (10^6), and µ for micro (10^{-6}).

Syntax:

`NumericalAbbreviation`

You set the `NumericalAbbreviation` variable to a string containing a list of abbreviation definition pairs, delimited by semi colon. Each abbreviation definition pair should contain the scale (the exponent in decimal base) and the abbreviation separated by a colon, for example, `6:M` for a million.

The default setting is `'3:k;6:M;9:G;12:T;15:P;18:E;21:Z;24:Y;-3:m;-6:µ;-9:n;-12:p;-15:f;-18:a;-21:z;-24:y'`.

Examples:

This setting will change the prefix for a thousand to t and the prefix for a billion to B. This would be useful for financial applications where you would expect abbreviations like t$, M$, and B$.

```
Set NumericalAbbreviation='3:t;6:M;9:B;12:T;15:P;18:E;21:Z;24:Y;-3:m;-6:µ;-9:n;-12:p;-15:f;-18:a;-21:z;-24:y';
```

ReferenceDay

The setting defines which day in January to set as reference day to define week 1. In other words, this setting prescribes how many days in week 1 must be dates within January.

Syntax:

`ReferenceDay`

`ReferenceDay` sets how many days are included in the first week of the year. `ReferenceDay` can be set to any value between `1` and `7`. Any value outside of the `1-7` range is interpreted as the midpoint of the week (`4`), which is equivalent to `ReferenceDay` being set to `4`.

If you do not select a value for the `ReferenceDay` setting, then the default value will show `ReferenceDay=0` which will be interpreted as the midpoint of the week (4), as seen in the `ReferenceDay` values table below.

The `ReferenceDay` function is often used in combination with the following functions:

Related functions

Function	Interaction
BrokenWeeks (page 169)	If the Qlik Sense app is operating with unbroken weeks, the `ReferenceDay` variable setting will be enforced. However, if broken weeks are being used, week 1 will begin on January 1 and terminate in conjunction with the `FirstWeekDay` variable setting and ignore the `ReferenceDay` flag.
FirstWeekDay (page 182)	Integer that defines which day to use as the first day of the week.

Qlik Sense allows the following values to be set for `ReferenceDay`:

ReferenceDay values

Value	Reference day
0 (default)	January 4
1	January 1
2	January 2
3	January 3
4	January 4
5	January 5
6	January 6
7	January 7

In the following example the `ReferenceDay = 3` defines January 3 as the reference day:

```
SET ReferenceDay=3; //(set January 3 as the reference day)
```

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 - Load script using the default value; ReferenceDay=0

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The `ReferenceDay` variable that is set to `0`.
- The `BrokenWeeks` variable that is set to `0` which forces the app to use unbroken weeks.
- A dataset of dates from the end of 2019 to the start of 2020.

Load script

```
SET BrokenWeeks = 0;
SET ReferenceDay = 0;

Sales:
LOAD
date,
sales,
week(date) as week,
weekday(date) as weekday
Inline [
date,sales
12/27/2019,5000
12/28/2019,6000
12/29/2019,7000
12/30/2019,4000
12/31/2019,3000
01/01/2020,6000
01/02/2020,3000
01/03/2020,6000
01/04/2020,8000
01/05/2020,5000
01/06/2020,7000
01/07/2020,3000
01/08/2020,5000
01/09/2020,9000
01/10/2020,5000
01/11/2020,7000
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `week`
- `weekday`

Results table

date	week	weekday
12/27/2019	52	Fri
12/28/2019	52	Sat
12/29/2019	1	Sun
12/30/2019	1	Mon
12/31/2019	1	Tue
01/01/2020	1	Wed
01/02/2020	1	Thu
01/03/2020	1	Fri
01/04/2020	1	Sat
01/05/2020	2	Sun
01/06/2020	2	Mon
01/07/2020	2	Tue
01/08/2020	2	Wed
01/09/2020	2	Thu
01/10/2020	2	Fri
01/11/2020	2	Sat

Week 52 concludes on Saturday, December 28. Because `ReferenceDay` requires January 4 to be included in week 1, week 1 therefore begins on December 29 and concludes on Saturday, January 4.

Example - ReferenceDay variable set to 5

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The `ReferenceDay` variable that is set to `5`.
- The `BrokenWeeks` variable that is set to `0` which forces the app to use unbroken weeks.
- A dataset of dates from the end of 2019 to the start of 2020.

Load script

```
SET BrokenWeeks = 0;
SET ReferenceDay = 5;

Sales:
```

```
LOAD
date,
sales,
week(date) as week,
weekday(date) as weekday
Inline [
date,sales
12/27/2019,5000
12/28/2019,6000
12/29/2019,7000
12/30/2019,4000
12/31/2019,3000
01/01/2020,6000
01/02/2020,3000
01/03/2020,6000
01/04/2020,8000
01/05/2020,5000
01/06/2020,7000
01/07/2020,3000
01/08/2020,5000
01/09/2020,9000
01/10/2020,5000
01/11/2020,7000
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `week`
- `weekday`

Results table

date	week	weekday
12/27/2019	52	Fri
12/28/2019	52	Sat
12/29/2019	53	Sun
12/30/2019	53	Mon
12/31/2019	53	Tue
01/01/2020	53	Wed
01/02/2020	53	Thu
01/03/2020	53	Fri
01/04/2020	53	Sat
01/05/2020	1	Sun

date	week	weekday
01/06/2020	1	Mon
01/07/2020	1	Tue
01/08/2020	1	Wed
01/09/2020	1	Thu
01/10/2020	1	Fri
01/11/2020	1	Sat

Week 52 concludes on Saturday, December 28. The `BrokenWeeks` variable forces the app to use unbroken weeks. The reference day value of `5` requires January 5 to be included in week 1.

However, this is eight days after the conclusion of week 52 of the previous year. Therefore, week 53 begins on December 29 and concludes on January 4. Week 1 begins on Sunday, January 5.

ThousandSep

The thousands separator defined replaces the digit grouping symbol of the operating system (regional settings).

Syntax:

```
ThousandSep
```

Examples:

```
Set ThousandSep=','; //(for example, seven billion must be specified as: 7,000,000,000)
```

TimeFormat

The format defined replaces the time format of the operating system (regional settings).

Syntax:

```
TimeFormat
```

Example:

```
Set TimeFormat='hh:mm:ss';
```

TimestampFormat

The format defined replaces the date and time formats of the operating system (regional settings).

Syntax:

```
TimestampFormat
```

Example:

The following examples use *1983-12-14T13:15:30Z* as timestamp data to show the results of different **SET TimestampFormat** statements. The date format used is **YYYYMMDD** and the time format is **h:mm:ss TT**. The date format is specified in the **SET DateFormat** statement and the time format is specified in the **SET TimeFormat** statement, at the top of the data load script.

Results

Example	Result
`SET TimestampFormat='YYYYMMDD';`	19831214
`SET TimestampFormat='M/D/YY hh:mm:ss[.fff]';`	12/14/83 13:15:30
`SET TimestampFormat='DD/MM/YYYY hh:mm:ss[.fff]';`	14/12/1983 13:15:30
`SET TimestampFormat='DD/MM/YYYY hh:mm:ss[.fff] TT';`	14/12/1983 1:15:30 PM
`SET TimestampFormat='YYYY-MM-DD hh:mm:ss[.fff] TT';`	1983-12-14 01:15:30

Examples: Load script

Example: Load script

In the first load script *SET TimestampFormat='DD/MM/YYYY h:mm:ss[.fff] TT'* is used. In the second load script the timestamp format is changed to *SET TimestampFormat='MM/DD/YYYY hh:mm:ss[.fff]'*. The different results show how the **SET TimeFormat** statement works with different time data formats.

The table below shows the data set that is used in the load scripts that follow. The second column of the table shows the format of each timestamp in the data set. The first five timestamps follow ISO 8601 rules but the sixth does not.

Data set

Table showing the time data used and the format for each timestamp in the data set.

transaction_timestamp	time data format
2018-08-30	YYYY-MM-DD
20180830T193614.857	YYYYMMDDhhmmss.sss
20180830T193614.857+0200	YYYYMMDDhhmmss.sss±hhmm
2018-09-16T12:30-02:00	YYYY-MM-DDhh:mm±hh:mm
2018-09-16T13:15:30Z	YYYY-MM-DDhh:mmZ
9/30/18 19:36:14	M/D/YY hh:mm:ss

In the **Data load editor**, create a new section, and then add the example script and run it. Then add, at least, the fields listed in the results column to a sheet in your app to see the result.

Load script

```
SET FirstWeekDay=0;
SET BrokenWeeks=1;
SET ReferenceDay=0;
SET DayNames='Mon;Tue;Wed;Thu;Fri;Sat;Sun';
SET LongDayNames='Monday;Tuesday;Wednesday;Thursday;Friday;Saturday;Sunday';
SET DateFormat='YYYYMMDD';
SET TimestampFormat='DD/MM/YYYY h:mm:ss[.fff] TT';

Transactions:
Load
*,
Timestamp(transaction_timestamp, 'YYYY-MM-DD hh:mm:ss[.fff]') as LogTimestamp
;

Load * Inline [
transaction_id, transaction_timestamp, transaction_amount, transaction_quantity, discount,
customer_id, size, color_code
3750, 2018-08-30, 12423.56, 23, 0,2038593, L, Red
3751, 20180830T193614.857, 5356.31, 6, 0.1, 203521, m, orange
3752, 20180830T193614.857+0200, 15.75, 1, 0.22, 5646471, S, blue
3753, 2018-09-16T12:30-02:00, 1251, 7, 0, 3036491, l, Black
3754, 2018-09-16T13:15:30Z, 21484.21, 1356, 75, 049681, xs, Red
3755, 9/30/18 19:36:14, -59.18, 2, 0.333333333333333, 2038593, M, Blue
];
```

Results

Qlik Sense table showing results of the TimestampFormat interpretation variable being used in the load script. The last timestamp in the data set does not return a correct date.

transaction_id	transaction_timestamp	LogTimeStamp
3750	2018-08-30	2018-08-30 00:00:00
3751	20180830T193614.857	2018-08-30 19:36:14
3752	20180830T193614.857+0200	2018-08-30 17:36:14
3753	2018-09-16T12:30-02:00	2018-09-16 14:30:00
3754	2018-09-16T13:15:30Z	2018-09-16 13:15:30
3755	9/30/18 19:36:14	-

The next load script uses the same data set. However, it uses *SET TimestampFormat='MM/DD/YYYY hh:mm:ss [.fff]'* to match the non-ISO 8601 format of the sixth timestamp.

In the **Data load editor**, replace the previous example script with the one below and run it. Then add, at least, the fields listed in the results column to a sheet in your app to see the result.

Load script

```
SET FirstWeekDay=0;
SET BrokenWeeks=1;
SET ReferenceDay=0;
SET DayNames='Mon;Tue;Wed;Thu;Fri;Sat;Sun';
SET LongDayNames='Monday;Tuesday;Wednesday;Thursday;Friday;Saturday;Sunday';
SET DateFormat='YYYYMMDD';
SET TimestampFormat='MM/DD/YYYY hh:mm:ss[.fff]';

Transactions:
Load
*,
Timestamp(transaction_timestamp, 'YYYY-MM-DD hh:mm:ss[.fff]') as LogTimestamp
;

Load * Inline [
transaction_id, transaction_timestamp, transaction_amount, transaction_quantity, discount,
customer_id, size, color_code
3750, 2018-08-30, 12423.56, 23, 0,2038593, L, Red
3751, 20180830T193614.857, 5356.31, 6, 0.1, 203521, m, orange
3752, 20180830T193614.857+0200, 15.75, 1, 0.22, 5646471, S, blue
3753, 2018-09-16T12:30-02:00, 1251, 7, 0, 3036491, l, Black
3754, 2018-09-16T13:15:30Z, 21484.21, 1356, 75, 049681, xs, Red
3755, 9/30/18 19:36:14, -59.18, 2, 0.333333333333333, 2038593, M, Blue
];
```

Results

Qlik Sense table showing results of the TimestampFormat interpretation variable being used in the load script.

transaction_id	transaction_timestamp	LogTimeStamp
3750	2018-08-30	2018-08-30 00:00:00
3751	20180830T193614.857	2018-08-30 19:36:14
3752	20180830T193614.857+0200	2018-08-30 17:36:14
3753	2018-09-16T12:30-02:00	2018-09-16 14:30:00
3754	2018-09-16T13:15:30Z	2018-09-16 13:15:30
3755	9/30/18 19:36:14	2018-09-16 19:36:14

2.15 Direct Discovery variables

Direct Discovery system variables

DirectCacheSeconds

You can set a caching limit to the Direct Discovery query results for visualizations. Once this time limit is reached, Qlik Sense clears the cache when new Direct Discovery queries are made. Qlik Sense queries the source data for the selections and creates the cache again for the designated time limit. The result for each combination of selections is cached independently. That is, the cache is refreshed for each selection independently, so one selection refreshes the cache only for the fields selected, and a second selection refreshes cache for its relevant fields. If the second selection includes fields that were refreshed in the first selection, they are not updated in cache again if the caching limit has not been reached.

The Direct Discovery cache does not apply to **Table** visualizations. Table selections query the data source every time.

The limit value must be set in seconds. The default cache limit is 1800 seconds (30 minutes).

The value used for **DirectCacheSeconds** is the value set at the time the **DIRECT QUERY** statement is executed. The value cannot be changed at runtime.

Example:

```
SET DirectCacheSeconds=1800;
```

DirectConnectionMax

You can do asynchronous, parallel calls to the database by using the connection pooling capability. The load script syntax to set up the pooling capability is as follows:

```
SET DirectConnectionMax=10;
```

The numeric setting specifies the maximum number of database connections the Direct Discovery code should use while updating a sheet. The default setting is 1.

> *This variable should be used with caution. Setting it to greater than 1 is known to cause problems when connecting to Microsoft SQL Server.*

DirectUnicodeStrings

Direct Discovery can support the selection of extended Unicode data by using the SQL standard format for extended character string literals (N’<extended string>’) as required by some databases (notably SQL Server). The use of this syntax can be enabled for Direct Discovery with the script variable **DirectUnicodeStrings**.

Setting this variable to 'true' will enable the use of the ANSI standard wide character marker “N” in front of the string literals. Not all databases support this standard. The default setting is 'false'.

DirectDistinctSupport

When a **DIMENSION** field value is selected in a Qlik Sense object, a query is generated for the source database. When the query requires grouping, Direct Discovery uses the **DISTINCT** keyword to select only unique values. Some databases, however, require the **GROUP BY** keyword. Set **DirectDistinctSupport** to 'false' to generate **GROUP BY** instead of **DISTINCT** in queries for unique values.

```
SET DirectDistinctSupport='false';
```

If DirectDistinctSupport is set to true, then **DISTINCT** is used. If it is not set, the default behavior is to use **DISTINCT**.

DirectEnableSubquery

In high cardinality multi-table scenarios, it is possible to generate sub queries in the SQL query instead of generating a large IN clause. This is activated by setting **DirectEnableSubquery** to 'true'. The default value is 'false'.

> *When **DirectEnableSubquery** is enabled, you cannot load tables that are not in Direct Discovery mode.*

```
SET DirectEnableSubquery='true';
```

Teradata query banding variables

Teradata query banding is a function that enables enterprise applications to collaborate with the underlying Teradata database in order to provide for better accounting, prioritization, and workload management. Using query banding you can wrap metadata, such as user credentials, around a query.

Two variables are available, both are strings that are evaluated and sent to the database.

SQLSessionPrefix

This string is sent when a connection to the database is created.

```
SET SQLSessionPrefix = 'SET QUERY_BAND = ' & Chr(39) & 'Who=' & OSuser() & ';' & Chr(39) & '
FOR SESSION;';
```

If **OSuser()** for example returns *WA\sbt*, this will be evaluated to `SET QUERY_BAND = 'Who=WA\sbt;' FOR SESSION;` , which is sent to the database when the connection is created.

SQLQueryPrefix

This string is sent for each single query.

```
SET SQLSessionPrefix = 'SET QUERY_BAND = ' & Chr(39) & 'Who=' & OSuser() & ';' & Chr(39) & '
FOR TRANSACTION;';
```

Direct Discovery character variables

DirectFieldColumnDelimiter

You can set the character used as the field delimiter in **Direct Query** statements for databases that require a character other than comma as the field delimiter. The specified character must be surrounded by single quotation marks in the **SET** statement.

```
SET DirectFieldColumnDelimiter= '|'
```

DirectStringQuoteChar

You can specify a character to use to quote strings in a generated query. The default is a single quotation mark. The specified character must be surrounded by single quotation marks in the **SET** statement.

```
SET DirectStringQuoteChar= '"';
```

DirectIdentifierQuoteStyle

You can specify that non-ANSI quoting of identifiers be used in generated queries. At this time, the only non-ANSI quoting available is GoogleBQ. The default is ANSI. Uppercase, lowercase, and mixed case can be used (ANSI, ansi, Ansi).

```
SET DirectIdentifierQuoteStyle="GoogleBQ";
```

For example, ANSI quoting is used in the following **SELECT** statement:

```
SELECT [Quarter] FROM [qvTest].[sales] GROUP BY [Quarter]
```

When **DirectIdentifierQuoteStyle** is set to "GoogleBQ", the **SELECT** statement would use quoting as follows:

```
SELECT [Quarter] FROM [qvTest.sales] GROUP BY [Quarter]
```

DirectIdentifierQuoteChar

You can specify a character to control the quoting of identifiers in a generated query. This can be set to either one character (such as a double quotation mark) or two (such as a pair of square brackets). The default is a double quotation mark.

```
SET DirectIdentifierQuoteChar='[]';
SET DirectIdentifierQuoteChar='``';
SET DirectIdentifierQuoteChar=' ';
SET DirectIdentifierQuoteChar='"';
```

DirectTableBoxListThreshold

When Direct Discovery fields are used in a **Table** visualization, a threshold is set to limit the number of rows displayed. The default threshold is 1000 records. The default threshold setting can be changed by setting the **DirectTableBoxListThreshold** variable in the load script. For example:

```
SET DirectTableBoxListThreshold=5000;
```

The threshold setting applies only to **Table** visualizations that contain Direct Discovery fields. **Table** visualizations that contain only in-memory fields are not limited by the **DirectTableBoxListThreshold** setting.

No fields are displayed in the **Table** visualization until the selection has fewer records than the threshold limit.

Direct Discovery number interpretation variables

DirectMoneyDecimalSep

The decimal separator defined replaces the decimal symbol for currency in the SQL statement generated to load data using Direct Discovery. This character must match the character used in **DirectMoneyFormat**.

Default value is `'.'`

Example:

```
Set DirectMoneyDecimalSep='.';
```

DirectMoneyFormat
The symbol defined replaces the currency format in the SQL statement generated to load data using Direct Discovery. The currency symbol for the thousands separator should not be included.

Default value is `'#.0000'`

Example:

```
Set DirectMoneyFormat='#.0000';
```

DirectTimeFormat
The time format defined replaces the time format in the SQL statement generated to load data using Direct Discovery.

Example:

```
Set DirectTimeFormat='hh:mm:ss';
```

DirectDateFormat
The date format defined replaces the date format in the SQL statement generated to load data using Direct Discovery.

Example:

```
Set DirectDateFormat='MM/DD/YYYY';
```

DirectTimeStampFormat
The format defined replaces the date and time format in the SQL statement generated in the SQL statement generated to load data using Direct Discovery.

Example:

```
Set DirectTimestampFormat='M/D/YY hh:mm:ss[.fff]';
```

2.16 Error variables

The values of all error variables will exist after the script execution. The first variable, ErrorMode, is input from the user, and the last three are output from Qlik Sense with information on errors in the script.

Error variables overview

Each variable is described further after the overview. You can also click the variable name in the syntax to immediately access the details for that specific variable.

Refer to the Qlik Sense online help for further details about the variables.

ErrorMode
This error variable determines what action is to be taken by Qlik Sense when an error is encountered during script execution.

```
ErrorMode
```

ScriptError

This error variable returns the error code of the last executed script statement.

```
ScriptError
```

ScriptErrorCount

This error variable returns the total number of statements that have caused errors during the current script execution. This variable is always reset to 0 at the start of script execution.

```
ScriptErrorCount
```

ScriptErrorList

This error variable will contain a concatenated list of all script errors that have occurred during the last script execution. Each error is separated by a line feed.

```
ScriptErrorList
```

ErrorMode

This error variable determines what action is to be taken by Qlik Sense when an error is encountered during script execution.

Syntax:

```
ErrorMode
```

Arguments:

Arguments

Argument	Description
ErrorMode=1	The default setting. The script execution will halt and the user will be prompted for action (non-batch mode).
ErrorMode =0	Qlik Sense will simply ignore the failure and continue script execution at the next script statement.
ErrorMode =2	Qlik Sense will trigger an "Execution of script failed..." error message immediately on failure, without prompting the user for action beforehand.

Example:

```
set ErrorMode=0;
```

ScriptError

This error variable returns the error code of the last executed script statement.

Syntax:

```
ScriptError
```

This variable will be reset to 0 after each successfully executed script statement. If an error occurs it will be set to an internal Qlik Sense error code. Error codes are dual values with a numeric and a text component. The following error codes exist:

Script error codes

Error code	Description
0	No error. Dual value text is empty.
1	General error.
2	Syntax error.
3	General ODBC error.
4	General OLE DB error.
5	General custom database error.
6	General XML error.
7	General HTML error.
8	File not found.
9	Database not found.
10	Table not found.
11	Field not found.
12	File has wrong format.
16	Semantic error.

Example:

```
set ErrorMode=0;
LOAD * from abc.qvf;
if ScriptError=8 then
exit script;
//no file;
end if
```

ScriptErrorCount

This error variable returns the total number of statements that have caused errors during the current script execution. This variable is always reset to 0 at the start of script execution.

Syntax:

```
ScriptErrorCount
```

ScriptErrorList

This error variable will contain a concatenated list of all script errors that have occurred during the last script execution. Each error is separated by a line feed.

Syntax:

```
ScriptErrorList
```

2 Script expressions

Expressions can be used in both **LOAD** statements and **SELECT** statements. The syntax and functions described here apply to the **LOAD** statement, and not to the **SELECT** statement, since the latter is interpreted by the ODBC driver and not by Qlik Sense. However, most ODBC drivers are often capable of interpreting a number of the functions described below.

Expressions consist of functions, fields and operators, combined in a syntax.

All expressions in a Qlik Sense script return a number and/or a string, whichever is appropriate. Logical functions and operators return 0 for False and -1 for True. Number to string conversions and vice versa are implicit. Logical operators and functions interpret 0 as False and all else as True.

The general syntax for an expression is:

General syntax

Expression	Fields	Operator
expression ::= (constant	constant	|
expression ::= (constant	fieldref	|
expression ::= (constant	operator1 expression	|
expression ::= (constant	expression operator2 expression	|
expression ::= (constant	function	|
expression ::= (constant	(expression)	)

where:

- **constant** is a string (a text, a date or a time) enclosed by single straight quotation marks, or a number. Constants are written with no thousands separator and with a decimal point as the decimal separator.
- **fieldref** is a field name of the loaded table.
- **operator1** is a unary operator (working on one expression, the one to the right).
- **operator2** is a binary operator (working on two expressions, one on each side).
- **function ::= functionname(parameters)**
- **parameters ::= expression { , expression }**

The number and types of parameters are not arbitrary. They depend on the function used.

Expressions and functions can thus be nested freely, and as long as the expression returns an interpretable value, Qlik Sense will not give any error messages.

3 Chart expressions

A chart (visualization) expression is a combination of functions, fields, and mathematical operators (+ * / =), and other measures. Expressions are used to process data in the app in order to produce a result that can be seen in a visualization. They are not limited to use in measures. You can build visualizations that are more dynamic and powerful, with expressions for titles, subtitles, footnotes, and even dimensions.

This means, for example, that instead of the title of a visualization being static text, it can be made from an expression whose result changes depending on the selections made.

For detailed reference regarding script functions and chart functions, see the Script syntax and chart functions.

3.1 Defining the aggregation scope

There are usually two factors that together determine which records are used to define the value of aggregation in an expression. When working in visualizations, these factors are:

- Dimensional value (of the aggregation in a chart expression)
- Selections

Together, these factors define the scope of the aggregation. You may come across situations where you want your calculation to disregard the selection, the dimension or both. In chart functions, you can achieve this by using the TOTAL qualifier, set analysis, or a combination of the two.

Aggregation: Method and description

Method	Description
TOTAL qualifier	Using the total qualifier inside your aggregation function disregards the dimensional value. The aggregation will be performed on all possible field values. The **TOTAL** qualifier may be followed by a list of one or more field names within angle brackets. These field names should be a subset of the chart dimension variables. In this case, the calculation is made disregarding all chart dimension variables except those listed, that is, one value is returned for each combination of field values in the listed dimension fields. Also, fields that are not currently a dimension in a chart may be included in the list. This may be useful in the case of group dimensions, where the dimension fields are not fixed. Listing all of the variables in the group causes the function to work when the drill-down level changes.
Set analysis	Using set analysis inside your aggregation overrides the selection. The aggregation will be performed on all values split across the dimensions.

Method	Description
TOTAL qualifier and set analysis	Using the **TOTAL** qualifier and set analysis inside your aggregation overrides the selection and disregards the dimensions.

ALL qualifier	Using the **ALL** qualifier inside your aggregation disregards the selection and the dimensions. The equivalent can be achieved with the {1} set analysis statement and the **TOTAL** qualifier:
	`=sum(All Sales)`
	`=sum({1} Total Sales)`

Example: TOTAL qualifier

The following example shows how TOTAL can be used to calculate a relative share. Assuming that Q2 has been selected, using TOTAL calculates the sum of all values disregarding the dimensions.

Example: Total qualifier

Year	Quarter	Sum(Amount)	Sum(TOTAL Amount)	Sum(Amount)/Sum(TOTAL Amount)
		3000	3000	100%
2012	Q2	1700	3000	56,7%
2013	Q2	1300	3000	43,3%

*To show the numbers as a percentage, in the properties panel, for the measure you want to show as a percentage value, under **Number formatting**, select **Number**, and from **Formatting**, choose **Simple** and one of the % formats.*

Example: Set analysis

The following example shows how set analysis can be used to make a comparison between data sets before any selection was made. Assuming that Q2 has been selected, using set analysis with the set definition {1} calculates the sum of all values disregarding any selections but split by the dimensions.

Example: Set analysis

Year	Quarter	Sum(Amount)	Sum({1} Amount)	Sum(Amount)/Sum({1} Amount)
		3000	10800	27,8%
2012	Q1	0	1100	0%
2012	Q3	0	1400	0%
2012	Q4	0	1800	0%
2012	Q2	1700	1700	100%

Year	Quarter	Sum(Amount)	Sum({1} Amount)	Sum(Amount)/Sum({1} Amount)
2013	Q1	0	1000	0%
2013	Q3	0	1100	0%
2013	Q4	0	1400	0%
2013	Q2	1300	1300	100%

Example: TOTAL qualifier and set analysis

The following example shows how set analysis and the TOTAL qualifier can be combined to make a comparison between data sets before any selection was made and across all dimensions. Assuming that Q2 has been selected, using set analysis with the set definition {1} and the TOTAL qualifier calculates the sum of all values disregarding any selections and disregarding the dimensions.

Example: TOTAL qualifier and set analysis

Year	Quarter	Sum (Amount)	Sum({1} TOTAL Amount)	Sum(Amount)/Sum({1} TOTAL Amount)
		3000	10800	27,8%
2012	Q2	1700	10800	15,7%
2013	Q2	1300	10800	12%

Data used in examples:

```
AggregationScope:
LOAD * inline [
Year Quarter Amount
2012 Q1 1100
2012 Q2 1700
2012 Q3 1400
2012 Q4 1800
2013 Q1 1000
2013 Q2 1300
2013 Q3 1100
2013 Q4 1400] (delimiter is ' ');
```

3.2 Set analysis

When you make a selection in an app, you define a subset of records in the data. Aggregation functions, such as `Sum()`, `Max()`, `Min()`, `Avg()`, and `Count()` are calculated based on this subset.

In other words, your selection defines the scope of the aggregation; it defines the set of records on which calculations are made.

Set analysis offers a way of defining a scope that is different from the set of records defined by the current selection. This new scope can also be regarded as an alternative selection.

This can be useful if you want to compare the current selection with a particular value, for example last year's value or the global market share.

Set expressions

Set expressions can be used inside and outside aggregation functions, and are enclosed in curly brackets.

Example: Inner set expression

```
Sum( {$<Year={2021}>} Sales )
```

Example: Outer set expression

```
{<Year={2021}>} Sum(Sales) / Count(distinct Customer)
```

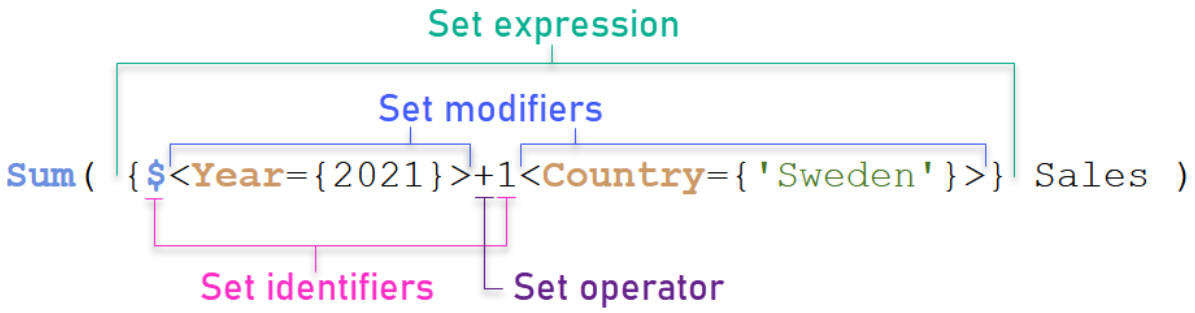
For more information about inner and outer set expressions, see *Inner and outer set expressions (page 245)*.

A set expression consists of a combination of the following elements:

- **Identifiers**. A set identifier represents a selection, defined elsewhere. It also represents a specific set of records in the data. It could be the current selection, a selection from a bookmark, or a selection from an alternate state. A simple set expression consists of a single identifier, such as the dollar sign, {$}, which means all records in the current selection.
  Examples: `$`, `1`, `BookMark1`, `State2`
- **Operators**. A set operator can be used to create unions, differences or intersections between different set identifiers. This way, you can create a subset or a superset of the selections defined by the set identifiers.
  Examples: `+`, `-`, `*`, `/`
- **Modifiers**. A set modifier can be added to the set identifier to change its selection. A modifier can also be used on its own and will then modify the default identifier. A modifier must be enclosed in angle brackets `<...>`.
  Examples: `<Year={2020}>`, `<Supplier={ACME}>`

The elements are combined to form set expressions.

Elements in a set expression



The set expression above, for example, is built from the aggregation `Sum(Sales)`.

The first operand returns sales for the year 2021 for the current selection, which is indicated by the $ set identifier and the modifier containing the selection of year 2021. The second operand returns Sales for Sweden, and ignores the current selection, which is indicated by the 1 set identifier.

Finally, the expression returns a set consisting of the records that belongs to any of the two set operands, as indicated by the + set operator.

Examples

Examples that combine the set expression elements above are available in the following topics:

Natural sets

Usually, a set expression represents both a set of records in the data model, and a selection that defines this subset of data. In this case, the set is called a natural set.

Set identifiers, with or without set modifiers, always represent natural sets.

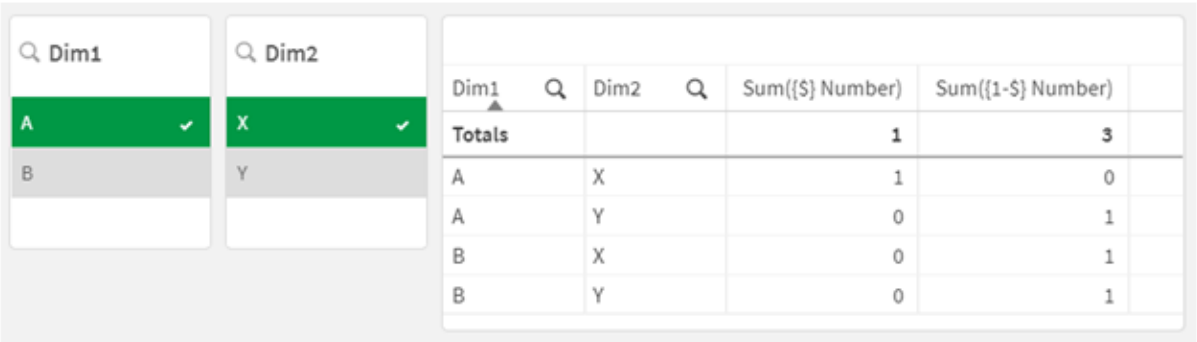
However, a set expression using set operators also represents a subset of the records, but can generally still not be described using a selection of field values. Such an expression is a non-natural set.

For example, the set given by {1-$} cannot always be defined by a selection. It is therefore not a natural set. This can be shown by loading the following data, adding it to a table, and then making selections using filter panes.

```
Load * Inline
[Dim1, Dim2, Number
A, X, 1
A, Y, 1
B, X, 1
B, Y, 1];
```

By making selections for Dim1 and Dim2, you get the view shown in the following table.

Table with natural and non-natural sets

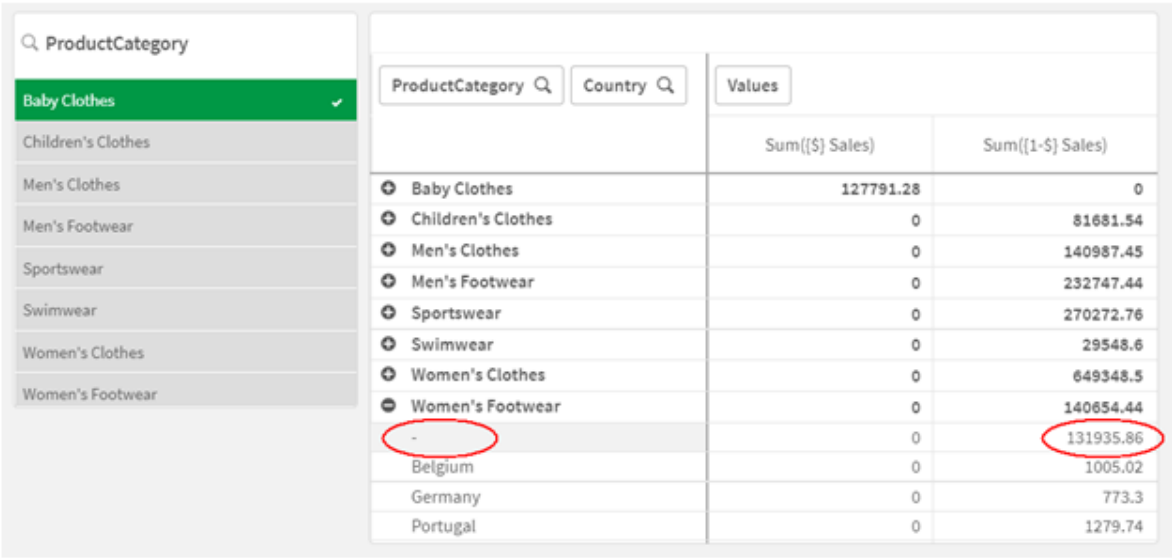


The set expression in the first measure uses a natural set: it corresponds to the selection that is made {$}.

The second measure is different. It uses {1-$}. It is not possible to make a selection that corresponds to this set, so it is a non-natural set.

This distinction has a number of consequences:

- Set modifiers can only be applied to set identifiers. They cannot be applied to an arbitrary set expression. For example, it is not possible to use a set expression such as:
  `{ (BM01 * BM02) <Field={x,y}> }`
  Here, the normal (round) brackets imply that the intersection between `BM01` and `BM02` should be evaluated before the set modifier is applied. The reason is that there is no element set that can be modified.
- You cannot use non-natural sets inside the `P()` and `E()` element functions. These functions return an element set, but it is not possible to deduce the element set from a non-natural set.
- A measure using a non-natural set cannot always be attributed to the right dimensional value if the data model has many tables. For example, in the following chart, some excluded sales numbers are attributed to the correct Country, whereas others have NULL as Country.
  Chart with non-natural set



Whether or not the assignment is made correctly depends on the data model. In this case, the number cannot be assigned if it pertains to a country that is excluded by the selection.

Identifier	Description
1	Represents the full set of all the records in the application, irrespective of any selections made.
$	Represents the records of the current selection. The set expression **{$}** is thus the equivalent to not stating a set expression.
$1	Represents the previous selection. $2 represents the previous selection-but-one, and so on.
$_1	Represents the next (forward) selection. $_2 represents the next selection-but-one, and so on.
BM01	You can use any bookmark ID or bookmark name.
MyAltState	You can reference the selections made in an alternate state by its state name.

Example	Result
sum ({1} Sales)	Returns total sales for the app, disregarding selections but not the dimension.
sum ({$} Sales)	Returns the sales for the current selection, that is, the same as sum(Sales).
sum ({$1} Sales)	Returns the sales for the previous selection.
sum ({BM01} Sales)	Returns the sales for the bookmark named *BM01*.

Example	Result
sum({$<OrderDate = DeliveryDate>} Sales)	Returns the sales for the current selection where OrderDate = DeliveryDate.
sum({1<Region = {US}>} Sales)	Returns the sales for region US, disregarding the current selection.
sum({$<Region = >} Sales)	Returns the sales for the selection, but with the selection in *Region* removed.
sum({<Region = >} Sales)	Returns the same as the example above. When the set to modify is omitted, $ is assumed.
sum({$<Year={2000}, Region={“U*”}>} Sales)	Returns the sales for the current selection, but with new selections both in *Year* and in *Region*.

Set identifiers

A set identifier represents a set of records in the data; either all the data or a subset of the data. It is the set of records defined by a selection. It could be the current selection, all data (no selection), a selection from a bookmark, or a selection from an alternate state.

In the example `Sum( {$<Year = {2009}>} Sales )`, the identifier is the dollar sign: `$`. This represents the current selection. It also represents all the possible records. This set can then altered by the modifier part of the set expression: the selection `2009` in `Year` is added.

In a more complex set expression, two identifiers can be used together with an operator to form a union, a difference, or an intersection of the two record sets.

The following table shows some common identifiers.

Examples with common identifiers

Identifier	Description
`1`	Represents the full set of all the records in the application, irrespective of any selections made.
`$`	Represents the records of the current selection in the default state. The set expression `{$}` is thus usually the equivalent to not stating a set expression.
`$1`	Represents the previous selection in the default state. `$2` represents the previous selection-but-one, and so on.

Identifier	Description
`$_1`	Represents the next (forward) selection. `$_2` represents the next selection-but-one, and so on.
`BM01`	You can use any bookmark ID or bookmark name.
`AltState`	You can reference an alternate state by its state name.
`AltState::BM01`	A bookmark contains the selections of all states, and you can reference a specific bookmark by qualifying the bookmark name.

The following table shows examples with different identifiers.

Examples with different identifiers

Example	Result
`Sum ({1} Sales)`	Returns total sales for the app, disregarding selections but not the dimension.
`Sum ({$} Sales)`	Returns the sales for the current selection, that is, the same as `Sum(Sales)`.
`Sum ({$1} Sales)`	Returns the sales for the previous selection.
`Sum ({BM01} Sales)`	Returns the sales for the bookmark named `BM01`.

Set operators

Set operators are used to include, exclude, or intersect data sets. All operators use sets as operands and return a set as result.

You can use set operators in two different situations:

- To perform a set operation on set identifiers, representing sets of records in data.
- To perform a set operation on the element sets, on the field values, or inside a set modifier.

The following table shows the operators that can be used in set expressions.

Operators

Operator	Description
+	Union. This binary operation returns a set consisting of the records or elements that belong to any of the two set operands.
-	Exclusion. This binary operation returns a set consisting of the records or elements that belong to the first but not the other of the two set operands. Also, when used as a unary operator, it returns the complement set.
*	Intersection. This binary operation returns a set consisting of the records or elements that belong to both set operands.
/	Symmetric difference (`XOR`). This binary operation returns a set consisting of the records or elements that belong to either, but not both set operands.

The following table shows examples with operators.

Examples with operators

Example	Result
`Sum ({1-$} Sales)`	Returns sales for everything excluded by current selection.
`Sum ({$*BM01} Sales)`	Returns sales for the intersection between the selection and bookmark `BM01`.
`Sum ({-($+BM01)} Sales)`	Returns sales excluded by the selection and bookmark `BM01`.
`Sum ({$<Year={2009}>+1<Country={'Sweden'}>} Sales)`	Returns sales for the year 2009 associated with the current selections and add the full set of data associated with the country `Sweden` across all years.
`Sum ({$<Country={"S*"}+{"*land"}>} Sales)`	Returns the sales for countries that begin with `S` or end with `land`.

Set modifiers

Set expressions are used to define the scope of a calculation. The central part of the set expression is the set modifier that specifies a selection. This is used to modify the user selection, or the selection in the set identifier, and the result defines a new scope for the calculation.

The set modifier consists of one or more field names, each followed by a selection that should be made on the field. The modifier is enclosed by angled brackets: `< >`

For example:

- `Sum ( {$<Year = {2015}>} Sales )`
- `Count ( {1<Country = {Germany}>} distinct OrderID )`
- `Sum ( {$<Year = {2015}, Country = {Germany}>} Sales )`

Element sets

An element set can be defined using the following:

- A list of values
- A search
- A reference to another field
- A set function

If the element set definition is omitted, the set modifier will clear any selection in this field. For example:

```
Sum( {$<Year = >} Sales )
```

Examples: Chart expressions for set modifiers based on element sets

Examples - chart expressions

Load script

Load the following data as an inline load in the data load editor to create the chart expression examples below.

```
MyTable:
Load * Inline [
Country, Year, Sales
Argentina, 2014, 66295.03
Argentina, 2015, 140037.89
Austria, 2014, 54166.09
Austria, 2015, 182739.87
Belgium, 2014, 182766.87
Belgium, 2015, 178042.33
Brazil, 2014, 174492.67
Brazil, 2015, 2104.22
Canada, 2014, 101801.33
Canada, 2015, 40288.25
Denmark, 2014, 45273.25
Denmark, 2015, 106938.41
Finland, 2014, 107565.55
Finland, 2015, 30583.44
France, 2014, 115644.26
France, 2015, 30696.98
Germany, 2014, 8775.18
Germany, 2015, 77185.68
];
```

Chart expressions

Create a table in a Qlik Sense sheet with the following chart expressions.

Table - Set modifiers based on element sets

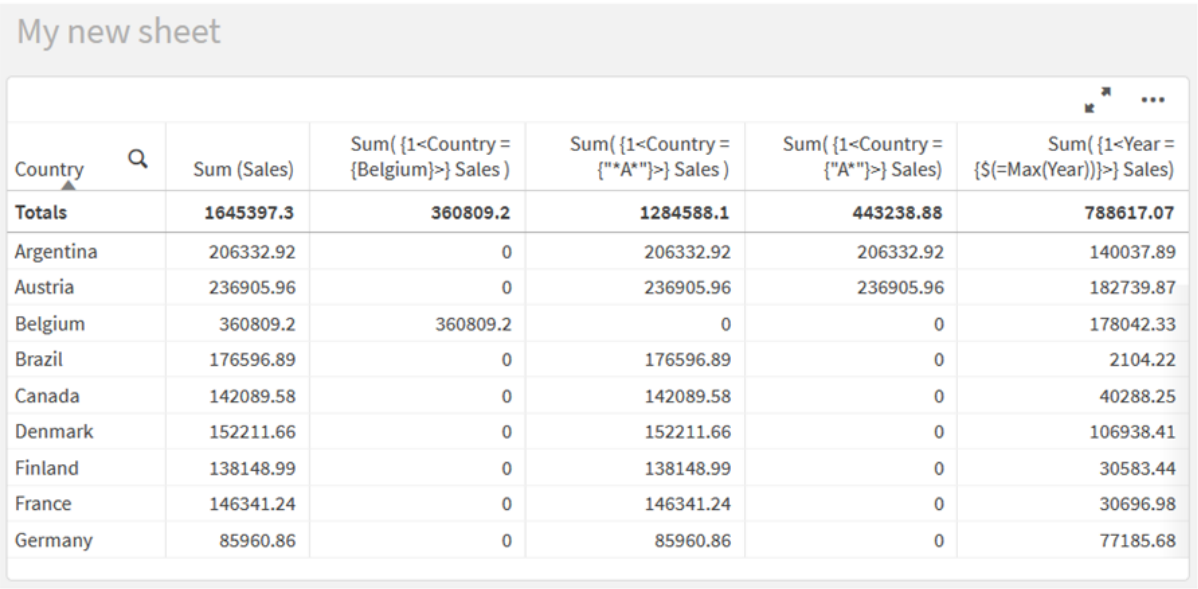
Country	Sum(Sales)	Sum ({1<Country= {Belgium}>} Sales)	Sum ({1<Country= {"*A*"}>} Sales)	Sum ({1<Country= {"A*"}>} Sales)	Sum({1<Year= {$(=Max (Year))}>} Sales)
Totals	1645397.3	360809.2	1284588.1	443238.88	788617.07
Argentina	206332.92	0	206332.92	206332.92	140037.89
Austria	236905.96	0	236905.96	236905.96	182739.87
Belgium	360809.2	360809.2	0	0	178042.33
Brazil	176596.89	0	176596.89	0	2104.22
Canada	142089.58	0	142089.58	0	40288.25

Country	Sum(Sales)	Sum ({1<Country={Belgium}>} Sales)	Sum ({1<Country={"*A*"}>} Sales)	Sum ({1<Country={"A*"}>} Sales)	Sum({1<Year={$(=Max(Year))}>} Sales)
Denmark	152211.66	0	152211.66	0	106938.41
Finland	138148.99	0	138148.99	0	30583.44
France	146341.24	0	146341.24	0	30696.98
Germany	85960.86	0	85960.86	0	77185.68

Explanation

- Dimensions:
  - `Country`
- Measures:
  - `Sum(Sales)`
    Sum `Sales` with no set expression.
  - `Sum({1<Country={Belgium}>}Sales)`
    Select `Belgium`, and then sum corresponding `Sales`.
  - `Sum({1<Country={"*A*"}>}Sales)`
    Select all countries that have an `A`, and then sum corresponding `Sales`.
  - `Sum({1<Country={"A*"}>}Sales)`
    Select all countries that begin with an `A`, and then sum corresponding `Sales`.
  - `Sum({1<Year={$(=Max(Year))}>}Sales)`
    Calculate the `Max(Year)`, which is `2015`, and then sum corresponding `Sales`.

Set modifiers based on element sets



Country	Sum (Sales)	Sum({1<Country = {Belgium}>} Sales)	Sum({1<Country = {"*A*"}>} Sales)	Sum({1<Country = {"A*"}>} Sales)	Sum({1<Year = {$(=Max(Year))}>} Sales)
Totals	**1645397.3**	**360809.2**	**1284588.1**	**443238.88**	**788617.07**
Argentina	206332.92	0	206332.92	206332.92	140037.89
Austria	236905.96	0	236905.96	236905.96	182739.87
Belgium	360809.2	360809.2	0	0	178042.33
Brazil	176596.89	0	176596.89	0	2104.22
Canada	142089.58	0	142089.58	0	40288.25
Denmark	152211.66	0	152211.66	0	106938.41
Finland	138148.99	0	138148.99	0	30583.44
France	146341.24	0	146341.24	0	30696.98
Germany	85960.86	0	85960.86	0	77185.68

Listed values

The most common example of an element set is one that is based on a list of field values enclosed in curly brackets. For example:

- `{$<Country = {Canada, Germany, Singapore}>}`
- `{$<Year = {2015, 2016}>}`

The inner curly brackets define the element set. The individual values are separated by commas.

Quotes and case sensitivity

If the values contain blanks or special characters, the values need to be quoted. Single quotes will be a literal, case-sensitive match with a single field value. Double quotes imply a case-insensitive match with one or several field values. For example:

- `<Country = {'New Zealand'}>`
  Matches `New Zealand` only.
- `<Country = {"New Zealand"}>`
  Matches `New Zealand`, `NEW ZEALAND`, and `new zealand`.

Dates must be enclosed in quotes and use the date format of the field in question. For example:

- `<ISO_Date = {'2021-12-31'}>`
- `<US_Date = {'12/31/2021'}>`
- `<UK_Date = {'31/12/2021'}>`

Double quotes can be substituted by square brackets or by grave accents.

Searches

Element sets can also be created through searches. For example:

- `<Country = {"C*"}>`
- `<Ingredient = {"*garlic*"}>`
- `<Year = {">2015"}>`
- `<Date = {">12/31/2015"}>`

Wildcards can be used in a text searches: An asterisk (*) represents any number of characters, and a question mark (?) represents a single character. Relational operators can be used to define numeric searches.

You should always use double quotes for searches. Searches are case-insensitive.

Dollar expansions

Dollar expansions are needed if you want to use a calculation inside your element set. For example, if you want to look at the last possible year only, you can use:

```
<Year = {$(=Max(Year))}>
```

Selected values in other fields

Modifiers can be based on the selected values of another field. For example:

```
<OrderDate = DeliveryDate>
```

This modifier will take the selected values from `DeliveryDate` and apply those as a selection on `OrderDate`. If there are many distinct values – more than a couple of hundred – then this operation is CPU intensive and should be avoided.

Element set functions

The element set can also be based on the set functions `P()` (possible values) and `E()` (excluded values).

For example, if you want to select countries where the product `Cap` has been sold, you can use:

```
<Country = P({1<Product={Cap}>} Country)>
```

Similarly, if you want to pick out the countries where the product `Cap` has not been sold, you can use:

```
<Country = E({1<Product={Cap}>} Country)>
```

Set modifiers with searches

You can create element sets through searches with set modifiers.

For example:

- `<Country = {"C*"}>`
- `<Year = {">2015"}>`
- `<Ingredient = {"*garlic*"}>`

Searches should always be enclosed in double quotes, square brackets or grave accents. You can use a list with a mixture of literal strings (single quotes) and searches (double quotes). For example:

```
<Product = {'Nut', "*Bolt", Washer}>
```

Text searches

Wildcards and other symbols can be used in text searches:

- An asterisk (*) will represent any number of characters.
- A question mark (?) will represent a single character.
- A circumflex accent (^) will mark the beginning of a word.

For example:

- `<Country = {"C*", "*land"}>`
  Match all countries beginning with a `C` or ending with `land`.
- `<Country = {"*^z*"}>`
  This will match all countries that have a word beginning with `z`, such as `New Zealand`.

Numeric searches

You can make numeric searches using these relational operators: >, >=, <, <=

A numeric search always begins with one of these operators. For example:

- `<Year = {">2015"}>`
  Match 2016 and subsequent years.
- `<Date = {">=1/1/2015<1/1/2016"}>`
  Match all dates during 2015. Note the syntax for describing a time range between two dates. The date format needs to match the date format of the field in question.

Expression searches

You can use expression searches to make more advanced searches. An aggregation is then evaluated for each field value in the search field. All values for which the search expression returns true are selected.

An expression search always begins with an equals sign: =

For example:

```
<Customer = {"=Sum(Sales)>1000"}>
```

This will return all customers with a sales value greater than 1000. `Sum(Sales)` is calculated on the current selection. This means that if you have a selection in another field, such as the `Product` field, you will get the customers that fulfilled the sales condition for the selected products only.

If you want the condition to be independent of the selection, you need to use set analysis inside the search string. For example:

```
<Customer = {"=Sum({1} Sales)>1000"}>
```

The expressions after the equals sign will be interpreted as a boolean value. This means that if it evaluates to something else, any non-zero number will be interpreted as true, while zero and strings are interpreted as false.

Quotes

Use quotation marks when the search strings contain blanks or special characters. Single quotes imply a literal, case-sensitive match with a single field value. Double quotes imply a case insensitive search that potentially matches multiple field values.

For example:

- `<Country = {'New Zealand'}>`
  Match `New Zealand` only.
- `<Country = {"New Zealand"}>`
  Match `New Zealand`, `NEW ZEALAND`, and `new zealand`

Double quotes can be substituted by square brackets or by grave accents.

> *In previous versions of Qlik Sense, there was no distinction between single quotes and double quotes, and all quoted strings were treated as searches. To maintain backward compatibility, apps created with older versions of Qlik Sense will continue to work as they did in previous versions. Apps created with Qlik Sense November 2017 or later will respect the difference between the two types of quotes.*

Examples: Chart expressions for set modifiers with searches

Examples - chart expressions

Load script

Load the following data as an inline load in the data load editor to create the chart expression examples below.

```
MyTable:
Load
Year(Date) as Year,
Date#(Date,'YYYY-MM-DD') as ISO_Date,
Date(Date#(Date,'YYYY-MM-DD'),'M/D/YYYY') as US_Date,
Country, Product, Amount
Inline
[Date, Country, Product, Amount
2018-02-20, Canada, Washer, 6
2018-07-08, Germany, Anchor bolt, 10
2018-07-14, Germany, Anchor bolt, 3
2018-08-31, France, Nut, 2
2018-09-02, Czech Republic, Bolt, 1
2019-02-11, Czech Republic, Bolt, 3
2019-07-31, Czech Republic, Washer, 6
2020-03-13, France, Anchor bolt, 1
2020-07-12, Canada, Anchor bolt, 8
2020-09-16, France, Washer, 1];
```

Example 1: Chart expressions with text searches

Create a table in a Qlik Sense sheet with the following chart expressions.

Table - Set modifiers with text searches

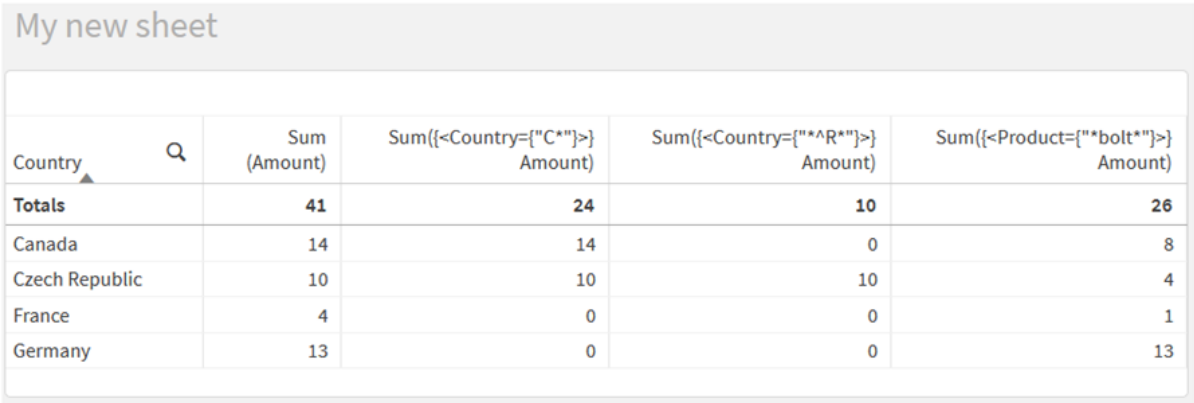
Country	Sum (Amount)	Sum({<Country={"C*"}>} Amount)	Sum({<Country={"*^R*"}>} Amount)	Sum({<Product={"*bolt*"}>} Amount)
Totals	**41**	**24**	**10**	**26**
Canada	14	14	0	8
Czech Republic	10	10	10	4
France	4	0	0	1
Germany	13	0	0	13

Explanation

- Dimensions:
  - `Country`

- Measures:
  - `Sum(Amount)`
    Sum `Amount` with no set expression.
  - `Sum({<Country={"C*"}>}Amount)`
    Sum `Amount` for all countries that start with `C`, such as `Canada` and `Czech Republic`.
  - `Sum({<Country={"*^R*"}>}Amount)`
    Sum `Amount` for all countries that have a word that starts with `R`, such as `Czech Republic`.
  - `Sum({<Product={"*bolt*"}>}Amount)`
    Sum `Amount` for all products that contain the string `bolt`, such as `Bolt` and `Anchor bolt`.

Set modifiers with text searches


My new sheet

Country	Sum (Amount)	Sum({<Country={"C*"}>} Amount)	Sum({<Country={"*^R*"}>} Amount)	Sum({<Product={"*bolt*"}>} Amount)
Totals	**41**	**24**	**10**	**26**
Canada	14	14	0	8
Czech Republic	10	10	10	4
France	4	0	0	1
Germany	13	0	0	13

Example 2: Chart expressions with numeric searches

Create a table in a Qlik Sense sheet with the following chart expressions.

Table - Set modifiers with numeric searches

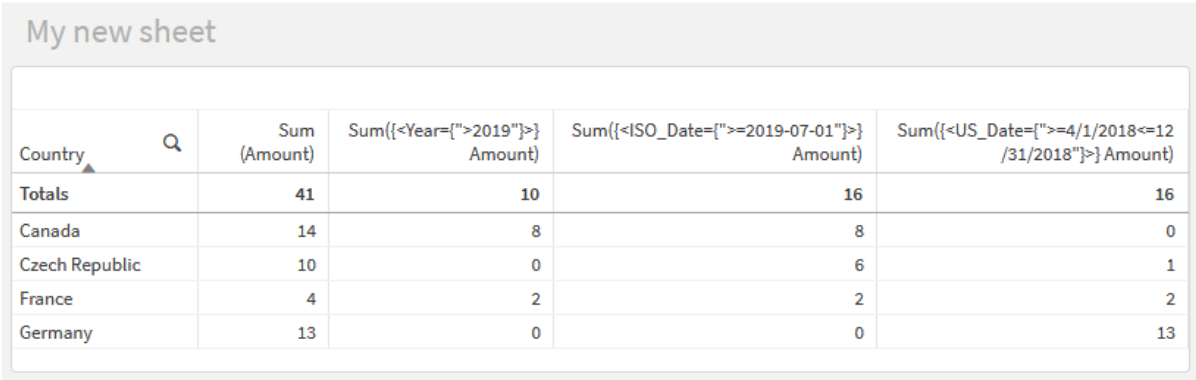
Country	Sum (Amount)	Sum({<Year={">2019"}>} Amount)	Sum({<ISO_Date={">=2019-07-01"}>} Amount)	Sum({<US_Date={">=4/1/2018<=12/31/2018"}>} Amount)
Totals	**41**	**10**	**16**	**16**
Canada	14	8	8	0
Czech Republic	10	0	6	1
France	4	2	2	2
Germany	13	0	0	13

Explanation

- Dimensions:
  - `Country`
- Measures:

- `Sum(Amount)`
  Sum `Amount` with no set expression.
- `Sum({<Year={">2019"}>}Amount)`
  Sum `Amount` for all years after 2019.
- `Sum({<ISO_Date={">=2019-07-01"}>}Amount)`
  Sum `Amount` for all dates on or after 2019-07-01. The format of the date in the search must match the format of the field.
- `Sum({<US_Date={">=4/1/2018<=12/31/2018"}>}Amount)`
  Sum `Amount` for all dates from 4/1/2018 to 12/31/2018, including the start and end dates. The format of the dates in the search must match the format of the field.

Set modifiers with numeric searches



Country	Sum (Amount)	Sum({<Year={">2019"}>} Amount)	Sum({<ISO_Date={">=2019-07-01"}>} Amount)	Sum({<US_Date={">=4/1/2018<=12/31/2018"}>} Amount)
Totals	41	10	16	16
Canada	14	8	8	0
Czech Republic	10	0	6	1
France	4	2	2	2
Germany	13	0	0	13

Example 3: Chart expressions with expression searches

Create a table in a Qlik Sense sheet with the following chart expressions.

Table - Set modifiers with expression searches

Country	Sum (Amount)	Sum({<Country={"=Sum (Amount)>10"}>} Amount)	Sum({<Country={"=Count(distinct Product)=1"}>} Amount)	Sum({<Product={"=Count (Amount)>3"}>} Amount)
Totals	**41**	**27**	**13**	**22**
Canada	14	14	0	8
Czech Republic	10	0	0	0
France	4	0	0	1
Germany	13	13	13	13

Explanation

- Dimensions:
  - `Country`

- Measures:
  - `Sum(Amount)`
    Sum `Amount` with no set expression.
  - `Sum({<Country={"=Sum(Amount)>10"}>}Amount)`
    Sum `Amount` for all countries that have an aggregated sum of `Amount` greater than 10.
  - `Sum({<Country={"=Count(distinct Product)=1"}>}Amount)`
    Sum `Amount` for all countries that are associated with exactly one distinct product.
  - `Sum({<Product={"=Count(Amount)>3"}>}Amount)`
    Sum `Amount` for all countries that have more than three transactions in the data.

Set modifiers with expression searches

My new sheet

Country	Sum (Amount)	Sum({<Country={"=Sum(Amount)>10"}>} Amount)	Sum({<Country={"=Count(distinct Product)=1"}>} Amount)	Sum({<Product={"=Count(Amount)>3"}>} Amount)
Totals	**41**	**27**	**13**	**22**
Canada	14	14	0	8
Czech Republic	10	0	0	0
France	4	0	0	1
Germany	13	13	13	13

Examples	Results
sum({$–1<Product = {“*Internal*”, “*Domestic*”}>} Sales)	Returns the sales for current selection, excluding transactions pertaining to products with the string 'Internal' or 'Domestic' in the product name.
sum({$<Customer = {“=Sum ({1<Year = {2007}>} Sales) > 1000000”}>} Sales)	Returns the sales for current selection, but with a new selection in the 'Customer' field: only customers who during 2007 had a total sales of more than 1000000.

Set modifiers with dollar-sign expansions

Dollar-sign expansions are constructs that are calculated before the expression is parsed and evaluated. The result is then injected into the expression instead of the `$(...)`. The calculation of the expression is then made using the result of the dollar expansion.

The expression editor shows a dollar expansion preview so that you can verify what your dollar-sign expansion evaluates to.

Dollar-sign expansion preview in expression editor

Use dollar-sign expansions when you want to use a calculation inside your element set.

For example, if you want to look at the last possible year only, you can use the following construction:

```
<Year = {$(=Max(Year))}>
```

`Max(Year)` is calculated first, and the result would be injected in the expression instead of the `$(…)`.

The result after the dollar expansion will be an expression such as the following:

```
<Year = {2021}>
```

The expression inside the dollar expansion is calculated based on the current selection. This means that if you have a selection in another field, the result of the expression will be affected.

If you want the calculation to be independent of the selection, use set analysis inside the dollar expansion. For example:

```
<Year = {$(=Max({1} Year))}>
```

Strings

When you want the dollar expansion to result in a string, normal quoting rules apply. For example:

```
<Country = {'$(=FirstSortedValue(Country,Date)'}>
```

The result after the dollar expansion will be an expression such as the following:

```
<Country = {'New Zealand'}>
```

You will get a syntax error if you do not use the quotation marks.

Numbers

When you want the dollar expansion to result in a number, ensure that the expansion gets the same formatting as the field. This means that you sometimes need to wrap the expression in a formatting function.

For example:

```
<Amount = {$(=Num(Max(Amount), '###0.00'))}>
```

The result after the dollar expansion will be an expression such as the following:

```
<Amount = {12362.00}>
```

Use a hash to force the expansion to always use decimal point and no thousand separator . For example:

```
<Amount = {$(#=Max(Amount))}>
```

Dates

When you want the dollar expansion to result in a date, ensure that the expansion has the correct formatting. This means that you sometimes need to wrap the expression in a formatting function.

For example:

```
<Date = {'$(=Date(Max(Date)))'}>
```

The result after the dollar expansion will be an expression such as the following:

```
<Date = {'12/31/2015'}>
```

Just as with strings, you need to use the correct quotes.

A common use case is that you want your calculation to be limited to the last month (or year). Then you can use a numeric search in combination with the `AddMonths()` function.

For example:

```
<Date = {">=$(=AddMonths(Today(),-1))"}>
```

The result after the dollar expansion will be an expression such as the following:

```
<Date = {">=9/31/2021"}>
```

This will pick out all events that have occurred the last month.

Example: Chart expressions for set modifiers with dollar-sign expansions

Example - chart expressions

Load script

Load the following data as an inline load in the data load editor to create the chart expression examples below.

```
Let vToday = Today();
MyTable:
Load
Year(Date) as Year,
Date#(Date,'YYYY-MM-DD') as ISO_Date,
Date(Date#(Date,'YYYY-MM-DD'),'M/D/YYYY') as US_Date,
Country, Product, Amount
Inline
[Date, Country, Product, Amount
2018-02-20, Canada, Washer, 6
2018-07-08, Germany, Anchor bolt, 10
2018-07-14, Germany, Anchor bolt, 3
2018-08-31, France, Nut, 2
```

```
2018-09-02, Czech Republic, Bolt, 1
2019-02-11, Czech Republic, Bolt, 3
2019-07-31, Czech Republic, Washer, 6
2020-03-13, France, Anchor bolt, 1
2020-07-12, Canada, Anchor bolt, 8
2021-10-15, France, Washer, 1];
```

Chart expressions with dollar-sign expansions

Create a table in a Qlik Sense sheet with the following chart expressions.

Table - Set modifiers with dollar-sign expansions

Country	Sum (Amount)	Sum({<US_Date={'$(vToday)'}>} Amount)	Sum({<ISO_Date={"$(=Date(Min(ISO_Date),'YYYY-MM-DD'))"}>} Amount)	Sum({<US_Date={">=$(=AddYears(Max(US_Date),-1))"}>} Amount)
Totals	**41**	**1**	**6**	**1**
Canada	14	0	6	0
Czech Republic	10	0	0	0
France	4	1	0	1
Germany	13	0	0	0

Explanation

- Dimensions:
  - `Country`
- Measures:
  - `Sum(Amount)`
    Sum `Amount` with no set expression.
  - `Sum({<US_Date={'$(vToday)'}>}Amount)`
    Sum `Amount` for all records where the `US_Date` is the same as in the variable `vToday`.
  - `Sum({<ISO_Date={"$(=Date(Min(ISO_Date),'YYYY-MM-DD'))"}>}Amount)`
    Sum `Amount` for all records where the `ISO_Date` is the same as the first (smallest) possible `ISO_Date`. The `Date()` function is needed to ensure that the format of the date matches that of the field.
  - `Sum({<US_Date={">=$(=AddYears(Max(US_Date),-1))"}>}Amount)`
    Sum `Amount` for all records that have a `US_Date` after or on the date a year before the latest (largest) possible `US_Date`. The `AddYears()` function will return a date in the format specified by the variable `DateFormat`, and this needs to match the format of the field `US_Date`.

Set modifiers with dollar-sign expansions

My new sheet

Country	Sum (Amount)	Sum({<US_Date=['$(vToday)']>} Amount)	Sum({<ISO_Date={"$(=Date(Min(ISO_Date),'YYYY-MM-DD'))"}>} Amount)	Sum({<US_Date={">=$(=AddYears(Max(US_Date),-1))"}>} Amount)
Totals	**41**	**1**	**6**	**1**
Canada	14	0	6	0
Czech Republic	10	0	0	0
France	4	1	0	1
Germany	13	0	0	0

Examples	Results
sum({$<Year = {$(#vLastYear)}>} Sales)	Returns the sales for the previous year in relation to current selection. Here, a variable vLastYear containing the relevant year is used in a dollar-sign expansion.
sum({$<Year = {$(#=Only(Year)-1)}>} Sales)	Returns the sales for the previous year in relation to current selection. Here, a dollar-sign expansion is used to calculate previous year.

Set modifiers with set operators

Set operators are used to include, exclude, or intersect different element sets. They combine the different methods to define element sets.

The operators are the same as those used for set identifiers.

Operators

Operator	Description
+	Union. This binary operation returns a set consisting of the records or elements that belong to any of the two set operands.
-	Exclusion. This binary operation returns a set consisting of the records or elements that belong to the first but not the other of the two set operands. Also, when used as a unary operator, it returns the complement set.
*	Intersection. This binary operation returns a set consisting of the records or elements that belong to both set operands.
/	Symmetric difference (XOR). This binary operation returns a set consisting of the records or elements that belong to either, but not both set operands.

For example, the following two modifiers define the same set of field values:

- `<Year = {1997, "20*"}>`
- `<Year = {1997} + {"20*"}>`

Both expressions select 1997 and the years that begin with 20. In other words, this is the union of the two conditions.

Set operators also allow for more complex definitions. For example:

```
<Year = {1997, "20*"} - {2000}>
```

This expression will select the same years as those above, but in addition exclude year 2000.

.

Examples: Chart expressions for set modifiers with set operators

Examples - chart expressions

Load script

Load the following data as an inline load in the data load editor to create the chart expression examples below.

```
MyTable:
Load
Year(Date) as Year,
Date#(Date,'YYYY-MM-DD') as ISO_Date,
Date(Date#(Date,'YYYY-MM-DD'),'M/D/YYYY') as US_Date,
Country, Product, Amount
Inline
[Date, Country, Product, Amount
2018-02-20, Canada, Washer, 6
2018-07-08, Germany, Anchor bolt, 10
2018-07-14, Germany, Anchor bolt, 3
2018-08-31, France, Nut, 2
2018-09-02, Czech Republic, Bolt, 1
2019-02-11, Czech Republic, Bolt, 3
2019-07-31, Czech Republic, Washer, 6
2020-03-13, France, Anchor bolt, 1
2020-07-12, Canada, Anchor bolt, 8
2020-09-16, France, Washer, 1];
```

Chart expressions

Create a table in a Qlik Sense sheet with the following chart expressions.

Table - Set modifiers with set operators

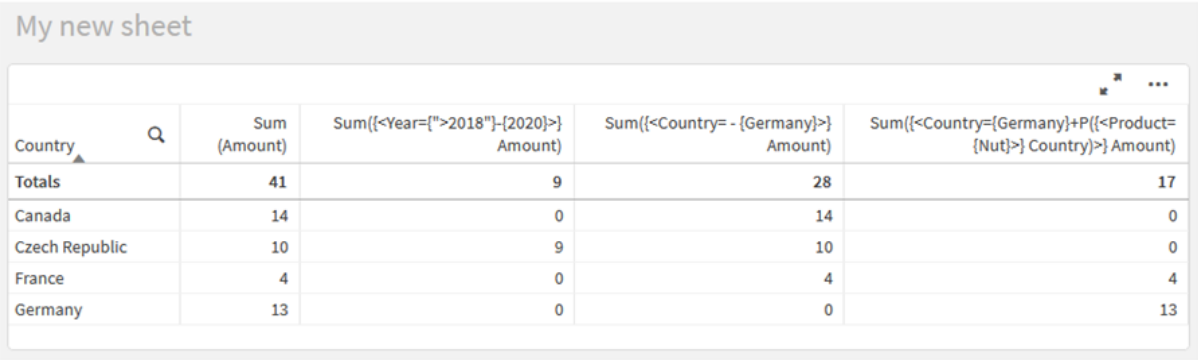
Country	Sum (Amount)	Sum({<Year={">2018"}-{2020}>} Amount)	Sum({<Country=-{Germany}>} Amount)	Sum({<Country={Germany}+P({<Product={Nut}>}Country)>} Amount)
Totals	**41**	**9**	**28**	**17**
Canada	14	0	14	0

Country	Sum (Amount)	Sum({<Year={">2018"}-{2020}>} Amount)	Sum({<Country=-{Germany}>} Amount)	Sum({<Country={Germany}+P({<Product={Nut}>}Country)>} Amount)
Totals	**41**	**9**	**28**	**17**
Czech Republic	10	9	10	0
France	4	0	4	4
Germany	13	0	0	13

Explanation

- Dimensions:
  - `Country`
- Measures:
  - `Sum(Amount)`
    Sum `Amount` with no set expression.
  - `Sum({<Year={">2018"}-{2020}>}Amount)`
    Sum `Amount` for all years after 2018, except 2020.
  - `Sum({<Country=-{Germany}>}Amount)`
    Sum `Amount` for all countries except `Germany`. Note the unary exclusion operator.
  - `Sum({<Country={Germany}+P({<Product={Nut}>}Country)>}Amount)`
    Sum `Amount` for `Germany` and all countries associated with the product `Nut`.

Set modifiers with set operators



Country	Sum (Amount)	Sum({<Year={">2018"}-{2020}>} Amount)	Sum({<Country= - {Germany}>} Amount)	Sum({<Country={Germany}+P({<Product={Nut}>} Country)>} Amount)
Totals	41	9	28	17
Canada	14	0	14	0
Czech Republic	10	9	10	0
France	4	0	4	4
Germany	13	0	0	13

Examples	Results
sum({$<Product = Product + {OurProduct1} – {OurProduct2} >} Sales)	Returns the sales for the current selection, but with the product “OurProduct1” added to the list of selected products and “OurProduct2” removed from the list of selected products.

Examples	Results
sum({$<Year = Year + ({“20*”,1997} – {2000}) >} Sales)	Returns the sales for the current selection but with additional selections in the field “Year”: 1997 and all that begin with “20” – however, not 2000. Note that if 2000 is included in the current selection, it will still be included after the modification.
sum({$<Year = (Year + {“20*”,1997}) – {2000} >} Sales)	Returns almost the same as above, but here 2000 will be excluded, also if it initially is included in the current selection. The example shows the importance of sometimes using brackets to define an order of precedence.
sum({$<Year = {“*”} – {2000}, Product = {“*bearing*”} >} Sales)	Returns the sales for the current selection but with a new selection in “Year”: all years except 2000; and only for products containing the string 'bearing'.

Set modifiers with implicit set operators

The standard way to write selections in a set modifier is to use an equals sign. For example:

```
Year = {">2015"}
```

The expression to the right of the equals sign in the set modifier is called an element set. It defines a set of distinct field values, in other words a selection.

This notation defines a new selection, disregarding the current selection in the field. So, if the set identifier contains a selection in this field, the old selection will be replaced by the one in the element set.

When you want to base your selection on the current selection in the field, you need to use a different expression

For example, if you want to respect the old selection, and add the requirement that the year is after 2015, you can write the following:

```
Year = Year * {">2015"}
```

The asterisk is a set operator defining an intersection, so you will get the intersection between the current selection in `Year`, and the additional requirement that the year be after `2015`. An alternative way to write this is the following:

```
Year *= {">2015"}
```

That is, the assignment operator (`*=`) implicitly defines an intersection.

Similarly, implicit unions, exclusions and symmetric differences can be defined using the following: `+=`, `–=`, `/=`

Examples: Chart expressions for set modifiers with implicit set operators

Examples - chart expressions

Load script

Load the following data as an inline load in the data load editor to create the chart expression examples below.

```
MyTable:
Load
Year(Date) as Year,
Date#(Date,'YYYY-MM-DD') as ISO_Date,
Date(Date#(Date,'YYYY-MM-DD'),'M/D/YYYY') as US_Date,
Country, Product, Amount
Inline
[Date, Country, Product, Amount
2018-02-20, Canada, Washer, 6
2018-07-08, Germany, Anchor bolt, 10
2018-07-14, Germany, Anchor bolt, 3
2018-08-31, France, Nut, 2
2018-09-02, Czech Republic, Bolt, 1
2019-02-11, Czech Republic, Bolt, 3
2019-07-31, Czech Republic, Washer, 6
2020-03-13, France, Anchor bolt, 1
2020-07-12, Canada, Anchor bolt, 8
2020-09-16, France, Washer, 1];
```

Chart expressions with implicit set operators

Create a table in a Qlik Sense sheet with the following chart expressions.

Select `Canada` and `Czech Republic` from a list of countries.

Table - Chart expressions with implicit set operators

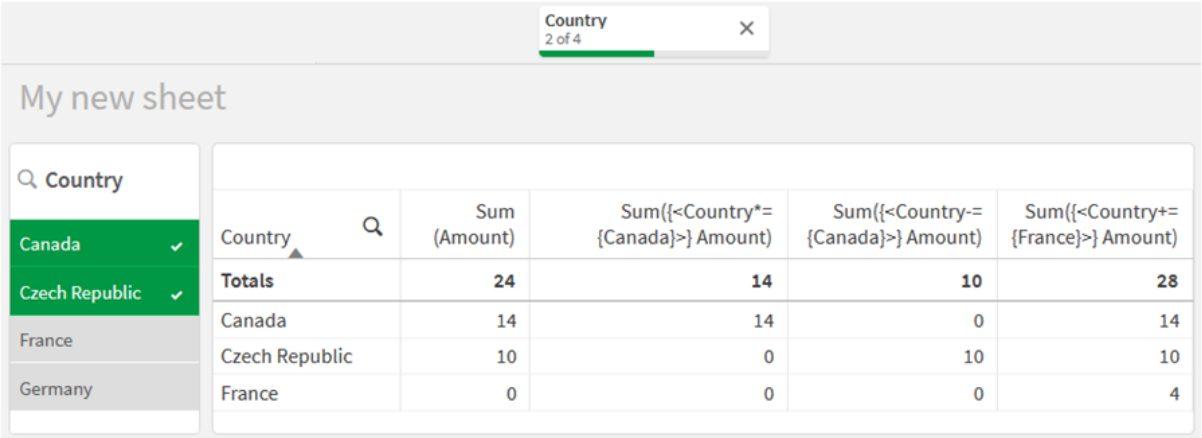
Country	Sum (Amount)	Sum({<Country*= {Canada}>} Amount)	Sum({<Country-= {Canada}>} Amount)	Sum({<Country+= {France}>} Amount)
Totals	**24**	**14**	**10**	**28**
Canada	14	14	0	14
Czech Republic	10	0	10	10
France	0	0	0	4

Explanation

- Dimensions:
  - `Country`

- Measures:
  - `Sum(Amount)`
    Sum `Amount` for the current selection. Note that only `Canada` and `Czech Republic` have non-zero values.
  - `Sum({<Country*={Canada}>}Amount)`
    Sum `Amount` for the current selection, intersected with the requirement that the `Country` be `Canada`. If `Canada` is not part of the user selection, the set expression returns an empty set, and the column will have `0` on all rows.
  - `Sum({<Country-={Canada}>}Amount)`
    Sum `Amount` for the current selection, but first exclude `Canada` from the `Country` selection. If `Canada` is not part of the user selection, the set expression will not change any numbers.
  - `Sum({<Country+={France}>}Amount)`
    Sum `Amount` for the current selection, but first add `France` to the `Country` selection. If `France` is already part of the user selection, the set expression will not change any numbers.

Set modifiers with implicit set operators



Examples	Results
sum({$<Product += {OurProduct1, OurProduct2} >} Sales)	Returns the sales for the current selection, but using an implicit union to add the products 'OurProduct1' and 'OurProduct2' to the list of selected products.
sum({$<Year += {“20*”,1997} – {2000} >} Sales)	Returns the sales for the current selection but using an implicit union to add a number of years in the selection: 1997 and all that begin with “20” – however, not 2000. Note that if 2000 is included in the current selection, it will still be included after the modification. Same as `<Year=Year + ({“20*”,1997}-{2000})>`.
sum({$<Product *= {OurProduct1} >} Sales)	Returns the sales for the current selection, but only for the intersection of currently selected products and the product OurProduct1.

Set modifiers using set functions

Sometimes you need to define a set of field values using a nested set definition. For example, you may want to select all customers that have bought a specific product, without selecting the product.

In such cases, use the element set functions `P()` and `E()`. These return the element sets of possible values and excluded values of a field, respectively. Inside the brackets, you can specify the field in question, and a set expression that defines the scope. For example:

```
P({1<Year = {2021}>} Customer)
```

This will return the set of customers that had transactions in 2021. You can then use this in a set modifier. For example:

```
Sum({<Customer = P({1<Year = {2021}>} Customer)>} Amount)
```

This set expression will select these customers, but it will not restrict the selection to 2021.

These functions cannot be used in other expressions.

Additionally, only natural sets can be used inside the element set functions. That is, a set of records that can be defined by a simple selection.

For example, the set given by `{1-$}` cannot always be defined through a selection, and is therefore not a natural set. Using these functions on non-natural sets will return unexpected results.

Examples: Chart expressions for set modifiers using set functions

Examples - chart expressions

Load script

Load the following data as an inline load in the data load editor to create the chart expression examples below.

```
MyTable:
Load
Year(Date) as Year,
Date#(Date,'YYYY-MM-DD') as ISO_Date,
Date(Date#(Date,'YYYY-MM-DD'),'M/D/YYYY') as US_Date,
Country, Product, Amount
Inline
[Date, Country, Product, Amount
2018-02-20, Canada, Washer, 6
2018-07-08, Germany, Anchor bolt, 10
2018-07-14, Germany, Anchor bolt, 3
2018-08-31, France, Nut, 2
2018-09-02, Czech Republic, Bolt, 1
2019-02-11, Czech Republic, Bolt, 3
2019-07-31, Czech Republic, Washer, 6
```

```
2020-03-13, France, Anchor bolt, 1
2020-07-12, Canada, Anchor bolt, 8
2020-09-16, France, Washer, 1];
```

Chart expressions

Create a table in a Qlik Sense sheet with the following chart expressions.

Table - Set modifiers using set functions

Country	Sum (Amount)	Sum({<Country=P ({<Year={2019}>}Country)>} Amount)	Sum({<Product=P ({<Year={2019}>}Product)>} Amount)	Sum({<Country=E ({<Product={Washer}>}Country)>} Amount)
Totals	**41**	**10**	**17**	**13**
Canada	14	0	6	0
Czech Republic	10	10	10	0
France	4	0	1	0
Germany	13	0	0	13

Explanation

- Dimensions:
  - `Country`
- Measures:
  - `Sum(Amount)`
    Sum `Amount` with no set expression.
  - `Sum({<Country=P({<Year={2019}>} Country)>} Amount)`
    Sum `Amount` for the countries that are associated with year `2019`. It will however not limit the calculation to `2019`.
  - `Sum({<Product=P({<Year={2019}>} Product)>} Amount)`
    Sum `Amount` for the products that are associated with year `2019`. It will however not limit the calculation to `2019`.
  - `Sum({<Country=E({<Product={Washer}>} Country)>} Amount)`
    Sum `Amount` for the countries that are not associated with the product `Washer`.

Set modifiers using set functions

My new sheet

Country	Sum (Amount)	Sum({<Country=P({<Year={2019}>} Country)>} Amount)	Sum({<Product=P({<Year={2019}>} Product)>} Amount)	Sum({<Country=E({<Product={Washer}>} Country)>} Amount)
Totals	**41**	**10**	**17**	**13**
Canada	14	0	6	0
Czech Republic	10	10	10	0
France	4	0	1	0
Germany	13	0	0	13

Examples	**Results**
sum({$<Customer = P({1<Product={'Shoe'}>} Customer)>} Sales)	Returns the sales for current selection, but only those customers that ever have bought the product 'Shoe'. The element function P( ) here returns a list of possible customers; those that are implied by the selection 'Shoe' in the field Product.
sum({$<Customer = P({1<Product={'Shoe'}>})>} Sales)	Same as above. If the field in the element function is omitted, the function will return the possible values of the field specified in the outer assignment.
sum({$<Customer = P({1<Product={'Shoe'}>} Supplier)>} Sales)	Returns the sales for current selection, but only those customers that ever have supplied the product 'Shoe', that is, the customer is also a supplier. The element function P( ) here returns a list of possible suppliers; those that are implied by the selection 'Shoe' in the field Product. The list of suppliers is then used as a selection in the field Customer.
sum({$<Customer = E({1<Product={'Shoe'}>})>} Sales)	Returns the sales for current selection, but only those customers that never bought the product 'Shoe'. The element function E( ) here returns the list of excluded customers; those that are excluded by the selection 'Shoe' in the field Product.

Inner and outer set expressions

Set expressions can be used inside and outside aggregation functions, and are enclosed in curly brackets.

When you use a set expression inside an aggregation function, it can look like this:

Example: Inner set expression

```
Sum( {$<Year={2021}>} Sales )
```

Use a set expression outside the aggregation function if you have expressions with multiple aggregations and want to avoid writing the same set expression in every aggregation function.

If you use an outer set expression, it must be placed at the beginning of the scope.

Example: Outer set expression

```
{<Year={2021}>} Sum(Sales) / Count(distinct Customer)
```

If you use a set expression outside the aggregation function, you can also apply it on existing master measures.

Example: Outer set expression applied to master measure

```
{<Year={2021}>} [Master Measure]
```

A set expression used outside aggregation functions affects the entire expression, unless it is enclosed in brackets then the brackets define the scope. In the lexical scoping example below, the set expression is only applied to the aggregation inside the brackets.

Example: Lexical scoping

```
( {<Year={2021}>} Sum(Amount) / Count(distinct Customer) ) – Avg(CustomerSales)
```

Rules

Lexical scope

The set expression affects the entire expression, unless it is enclosed in brackets. If so, the brackets define the lexical scope.

Position

The set expression must be placed in the beginning of the lexical scope.

Context

The context is the selection that is relevant for the expression. Traditionally, the context has always been the default state of current selection. But if an object is set to an alternate state, the context is the alternate state of the current selection.

You can also define a context in the form of an outer set expression.

Inheritance

Inner set expressions have precedence over outer set expressions. If the inner set expression contains a set identifier, it replaces the context. Otherwise, the context and the set expression will be merged.

- `{$<SetExpression>}` - overrides the outer set expression
- `{<SetExpression>}` - is merged with the outer set expression

Element set assignment

The element set assignment determines how the two selections are merged. If a normal equals sign is used, the selection in the inner set expression has precedence. Otherwise, the implicit set operator will be used.

- `{<Field={value}>}` - this inner selection replaces any outer selection in "`Field`".
- `{<Field+={value}>}` - this inner selection is merged with the outer selection in "`Field`", using the union operator.
- `{<Field*={value}>}` - this inner selection is merged with the outer selection in "`Field`", using the intersection operator.

Inheritance in multiple steps

The inheritance can occur in multiple steps. Examples:

- Current Selection → `Sum(Amount)`
  The aggregation function will use the context, which here is the current selection.
- Current Selection → `{<Set1>} Sum(Amount)`
  `Set1` will inherit from current selection, and the result will be the context for the aggregation function.
- Current Selection → `{<Set1>} ({<Set2>} Sum(Amount))`
  `Set2` will inherit from `Set1`, which in turn inherits from current selection, and the result will be the context for the aggregation function.

The Aggr() function

The `Aggr()` function creates a nested aggregation that has two independent aggregations. In the example below, a `Count()` is calculated for each value of `Dim`, and the resulting array is aggregated using the `Sum()` function.

Example:

```
Sum(Aggr(Count(X),Dim))
```

`Count()` is the inner aggregation and `Sum()` is the outer aggregation.

- The inner aggregation does not inherit any context from the outer aggregation.
- The inner aggregation inherits the context from the `Aggr()` function, which may contain a set expression.
- Both the `Aggr()` function and the outer aggregation function inherit the context from an outer set expression.

Tutorial - Creating a set expression

You can build set expressions to support data analysis. In this context, the analysis is often referred to as set analysis. Set analysis offers a way of defining a scope that is different from the set of records defined by the current selection in an app.

What you will learn

This tutorial provides the data and chart expressions to build set expressions using set modifiers, identifiers and operators.

Who should complete this tutorial

This tutorial is for app developers who are comfortable working with the script editor and chart expressions.

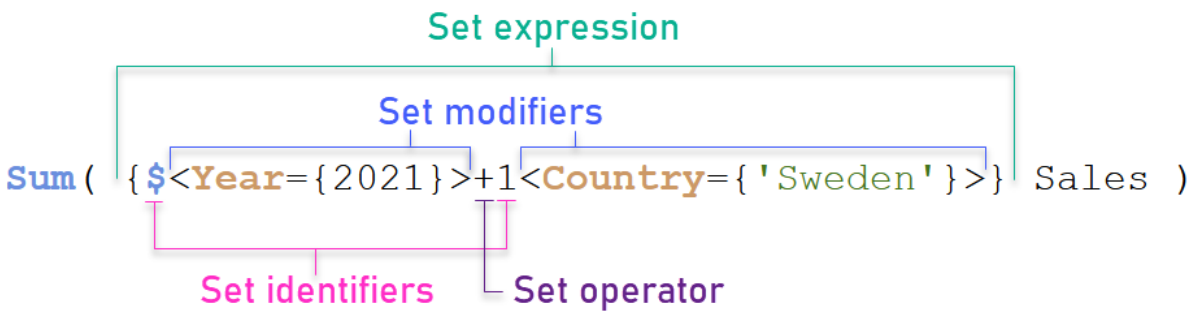
What you need to do before you start

A Qlik Sense Enterprise professional access allocation, which allows you to load data and create apps.

Elements in a set expression

Set expressions are enclosed in an aggregation function, such as `Sum()`, `Max()`, `Min()`, `Avg()`, or `Count()`. Set expressions are constructed from building blocks known as elements. These elements are set modifiers, identifiers, and operators.

Elements in a set expression



The set expression above, for example, is built from the aggregation `Sum(Sales)`. The set expression is enclosed in the outer curly brackets: `{ }`

The first operand in the expression is: `$<Year={2021}>`

This operand returns sales for the year `2021` for the current selection. The modifier, `<Year={2021}>`, contains the selection of the year 2021. The `$` set identifier indicates that the set expression is based on current selection.

The second operand in the expression is: `1<Country={'Sweden'}>`

This operand returns Sales for Sweden. The modifier, `<Country={'Sweden'}>`, contains the selection of the country Sweden. The `1` set identifier indicates that selections made in the app will be ignored.

Finally, the `+` set operator indicates that the expression returns a set consisting of the records that belongs to any of the two set operands.

Creating a set expression tutorial

Complete the following procedures to create the set expressions shown in this tutorial.

Create a new app and load data

Do the following:

1. Create a new app.
2. Click **Script editor**. Alternatively, click **Prepare** > **Data load editor** in the navigation bar.
3. Create a new section in the **Data load editor**.
4. Copy the following data and paste it into the new section: *Set expression tutorial data (page 256)*
5. Click **Load data**. The data is loaded as an inline load.

Create set expressions with modifiers

The set modifier consists of one or more field names, each followed by a selection that should be made on the field. The modifier is enclosed by angled brackets. For example, in this set expression:

```
Sum ( {<Year = {2015}>} Sales )
```

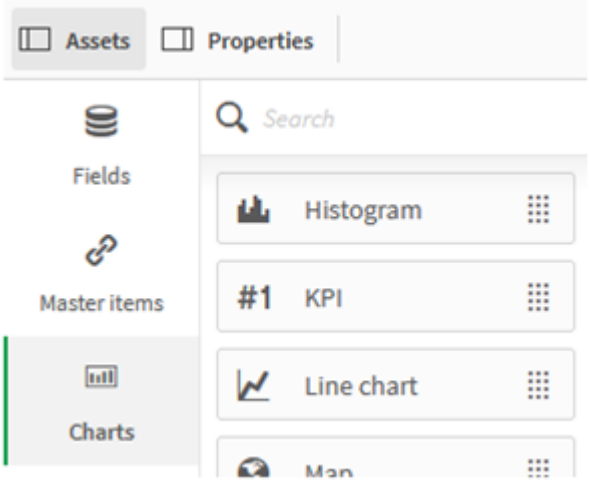
The modifier is:

```
<Year = {2015}>
```

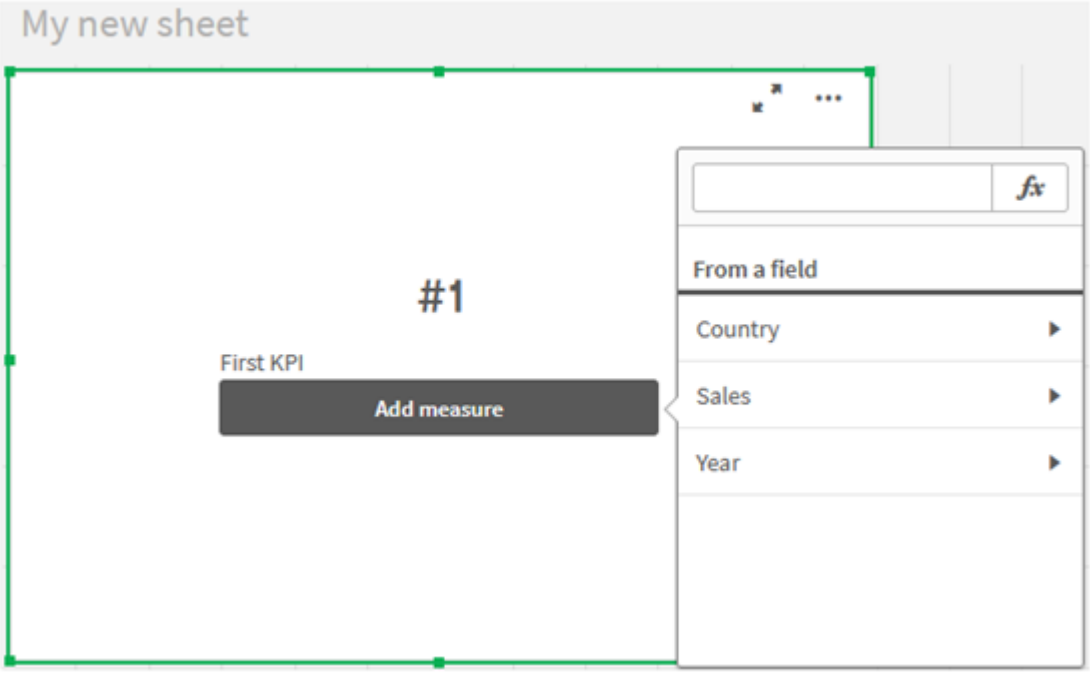
This modifier specifies that data from the year 2015 will be selected. The curly brackets in which the modifier is enclosed indicate a set expression.

Do the following:

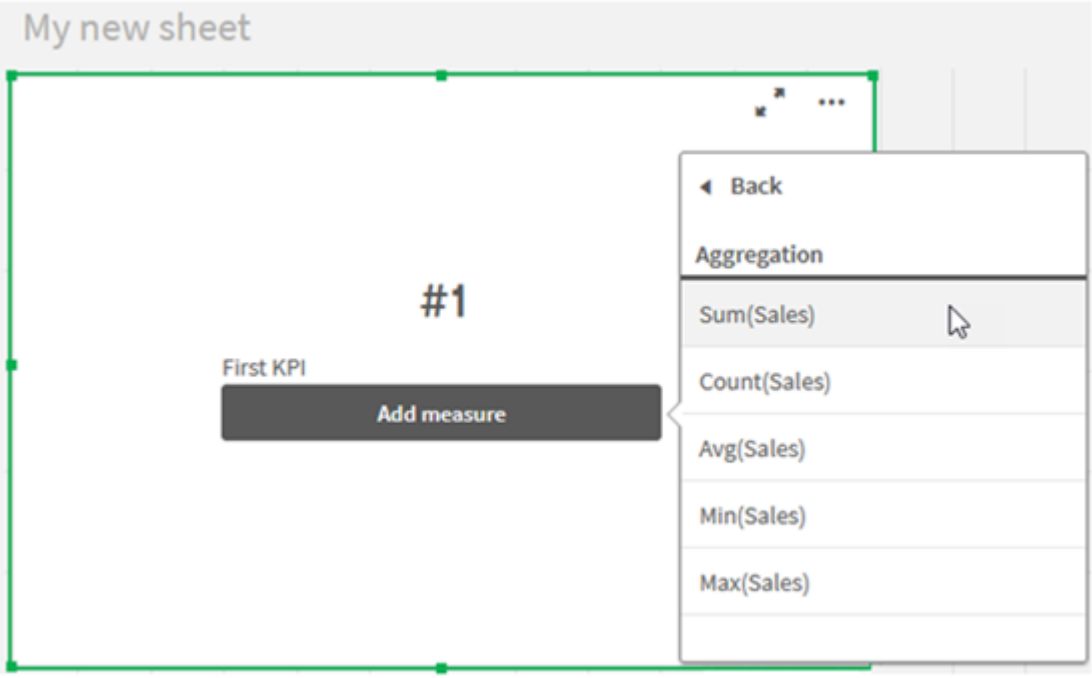
1. In a sheet, open the **Assets** panel from the navigation bar, and then click **Charts**.



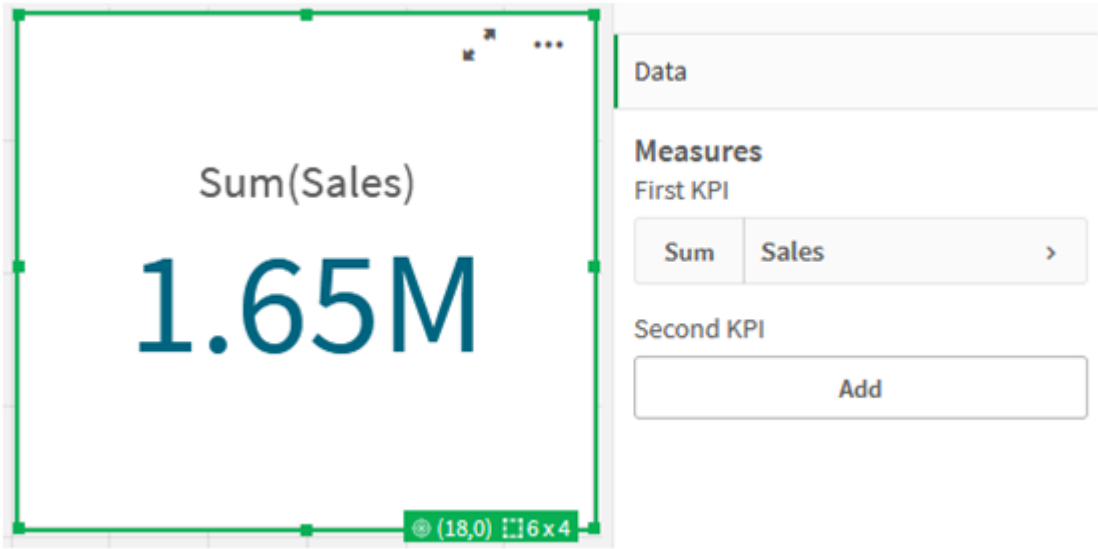
2. Drag a **KPI** onto the sheet, and then click **Add measure**.



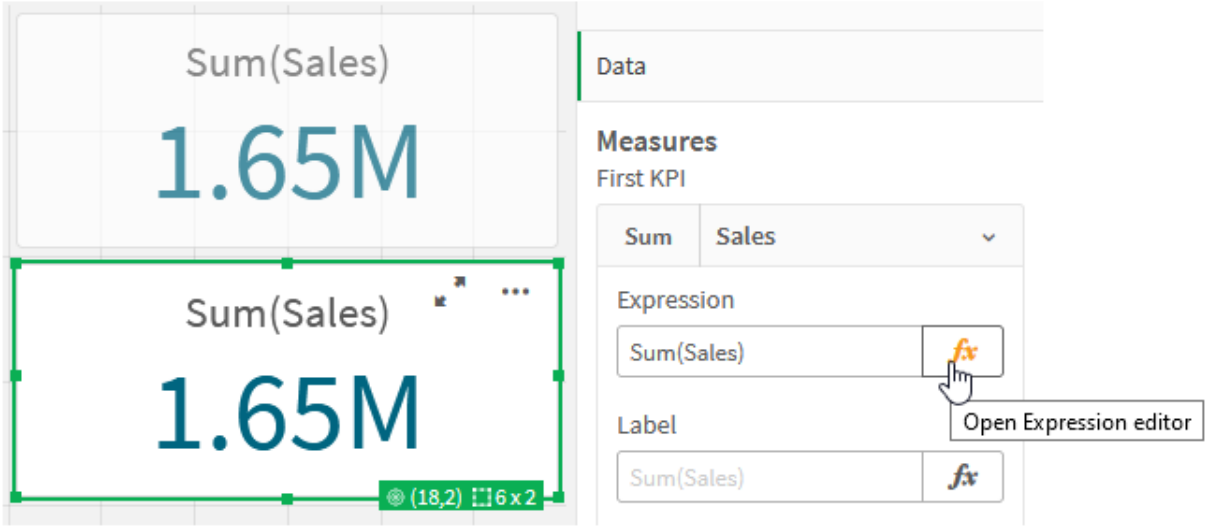
3. Click `Sales`, and then select `Sum(Sales)` for the aggregation.



The KPI shows the sum of sales for all years.

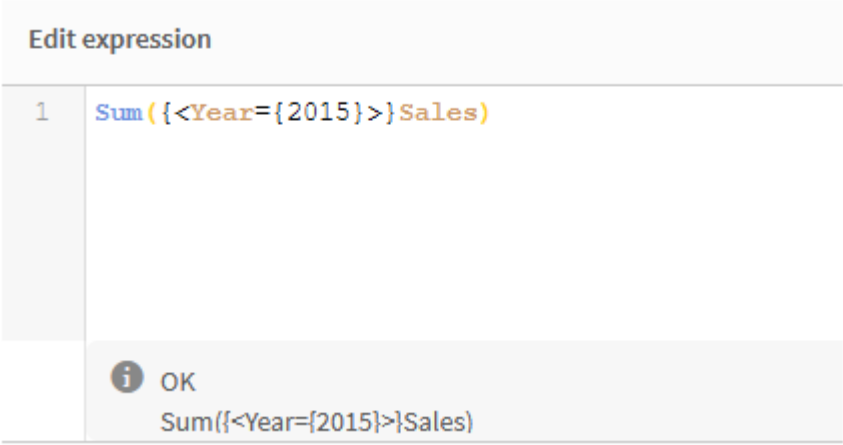


4. Copy and paste the KPI to create a new KPI.
5. Click the new KPI, click **Sales** under **Measures**, and then click **Open Expression editor**.



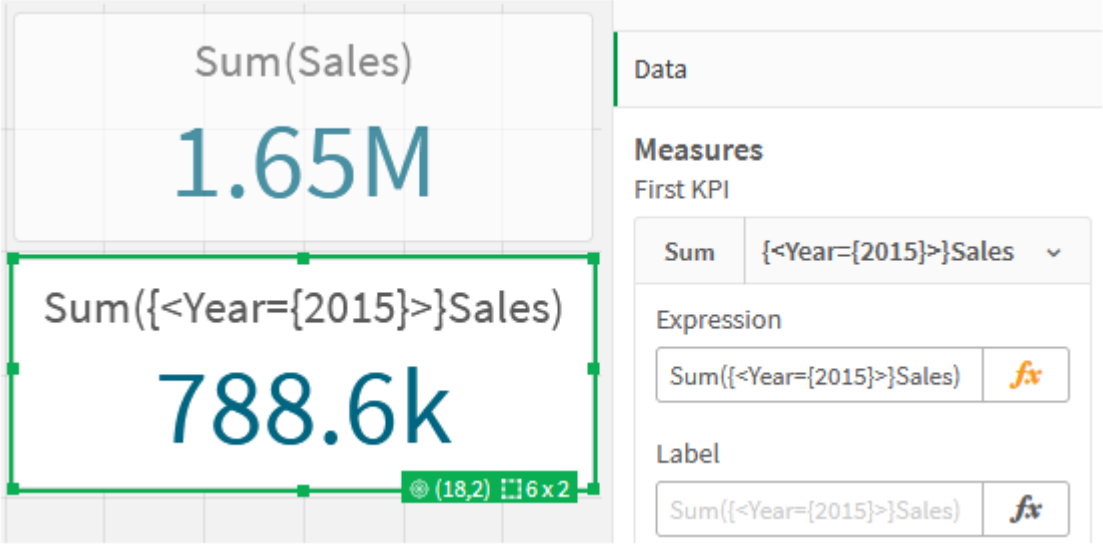
The expression editor open with the aggregation `Sum(Sales)`.

6. In the expression editor, create an expression to sum Sales for 2015 only:
   i. Add curly brackets to indicate a set expression: `Sum({}Sales)`
   i. Add angle brackets to indicate a set modifier: `Sum({<>}Sales)`
   ii. In the angle brackets, add the field to be selected, in this case the field is `Year`, followed by an equal sign. Next, enclose 2015 in another set of curly brackets. The resulting set modifier is: `{<Year={2015}>}`.
   The entire expression is:
   `Sum({<Year={2015}>}Sales)`



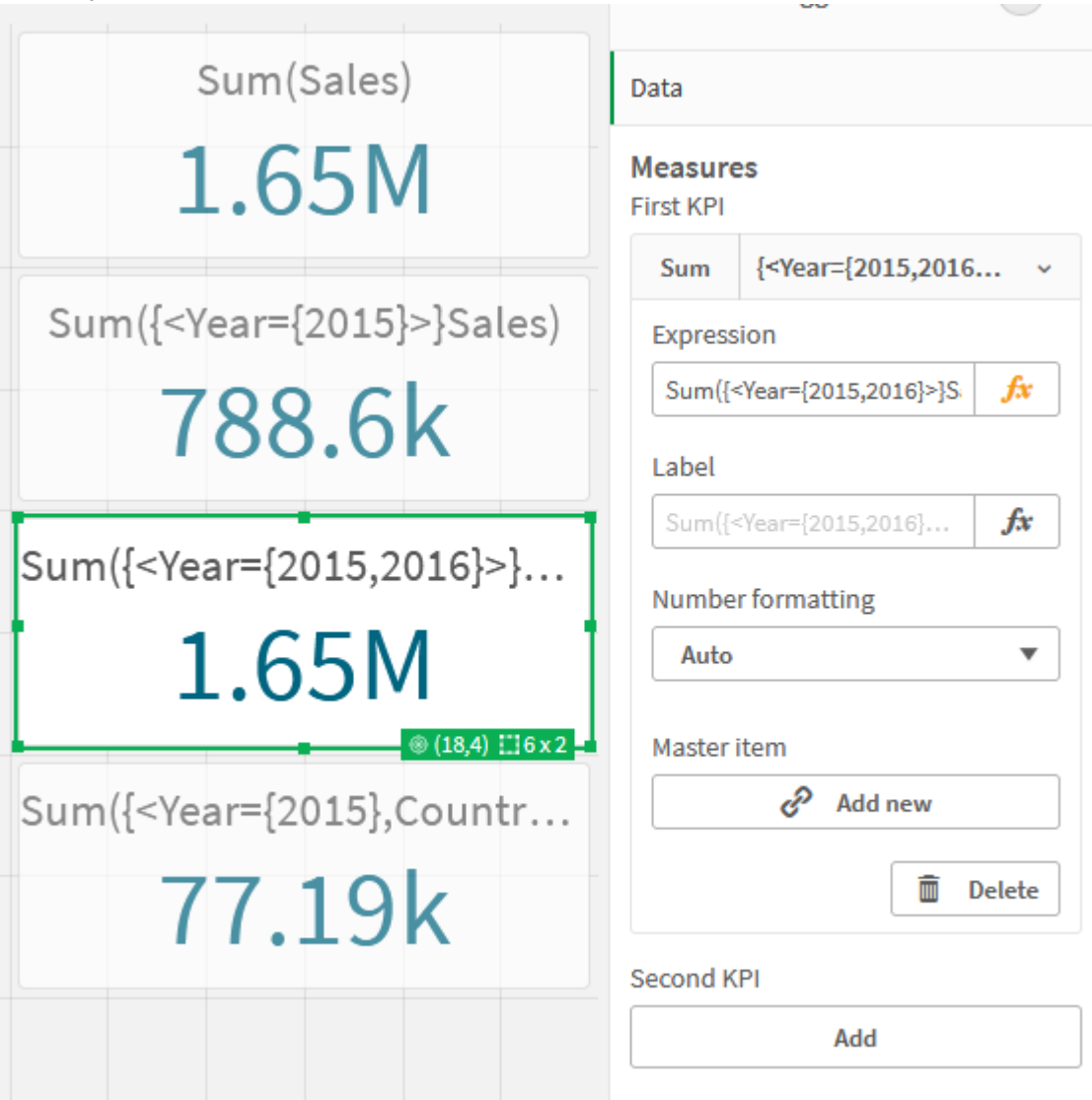
   iii. Click **Apply** to save the expression and to close the expression editor. The sum of Sales for 2015

is shown in the KPI.



7. Create two more KPIs with the following expressions:
   `Sum({<Year={2015,2016}>}Sales)`
   The modifier in the above is `<Year={2015,2016}>`. The expression will return the sum of Sales for 2015 and 2016.
   `Sum({<Year={2015},Country={'Germany'}>} Sales)`
   The modifier in the above is `<Year={2015}, Country={'Germany'}>`. The expression will return the sum of Sales for 2015, where 2015 intersects with Germany.

KPIs using set modifiers



Add set identifiers

The set expressions above will use current selections as base, because an identifier was not used. Next, add identifiers to specify the behavior when selections are made.

Do the following:

On your sheet, build or copy the following set expressions:

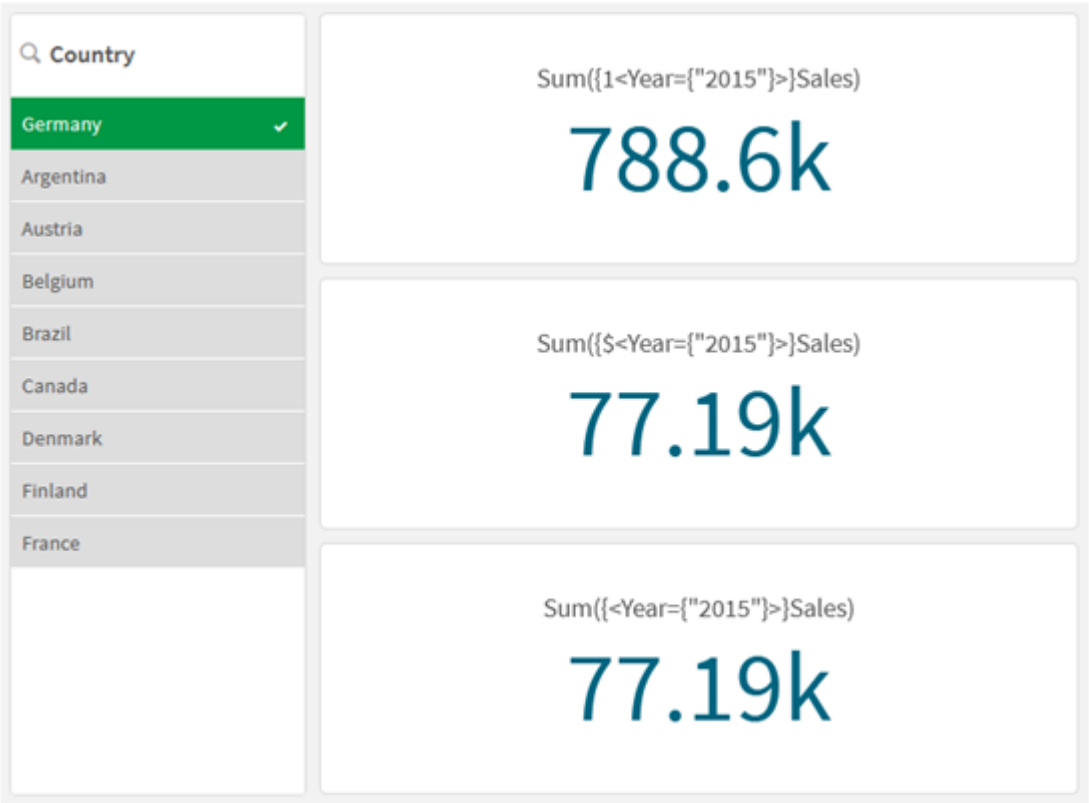
```
Sum({$<Year={"2015"}>}Sales
```

The $ identifier will base the set expression on the current selections made in the data. This is also the default behavior when an identifier is not used.

```
Sum({1<Year={"2015"}>}Sales)
```

The 1 identifier will cause the aggregation of `Sum(Sales)` on 2015 to ignore the current selection. The value of the aggregation will not change when the user makes other selections. For example, when Germany is selected below, the value for the aggregate sum of 2015 does not change.

KPIs using set modifiers and identifiers



Add operators

Set operators are used to include, exclude, or intersect data sets. All operators use sets as operands and return a set as result.

You can use set operators in two different situations:

- To perform a set operation on set identifiers, representing sets of records in data.
- To perform a set operation on the element sets, on the field values, or inside a set modifier.

Do the following:

On your sheet, build or copy the following set expression:

```
Sum({$<Year={2015}>+1<Country={'Germany'}>}Sales)
```

The plus sign `(+)` operator produces a union of the data sets for 2015 and Germany. As explained with set identifiers above, the dollar sign `($)` identifier means current selections will be used for the first operand, `<Year={2015}>`, will be respected. The 1 identifier means selection will be ignored for the second operand, `<Country={'Germany'}>`.

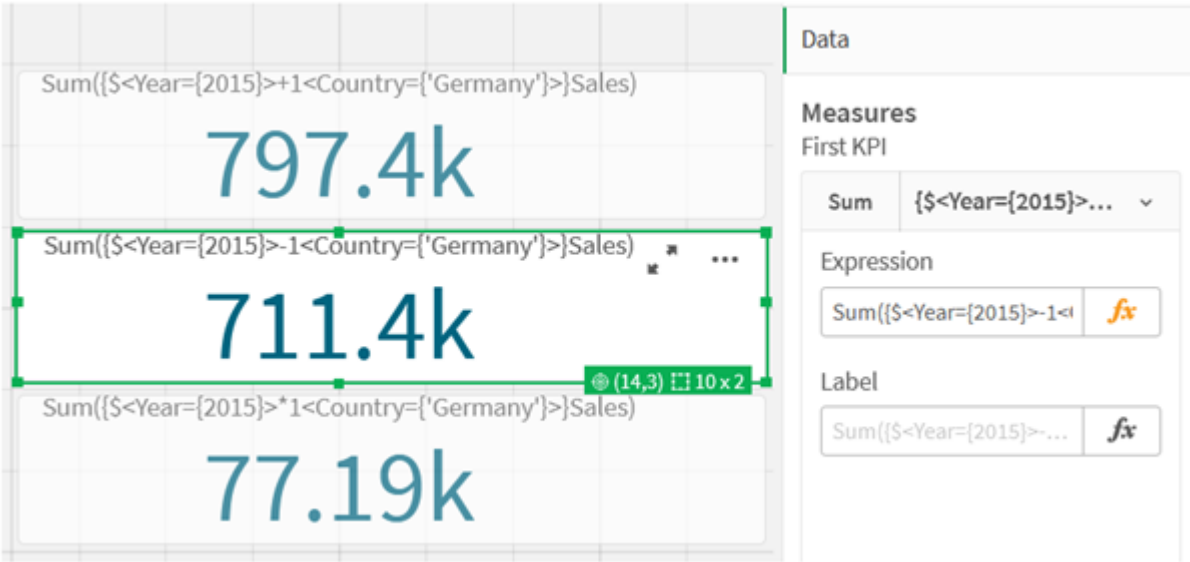
KPI using plus sign (+) operator

Alternatively, use a minus sign (-) to return a data set that consists of the records that belong to 2015 but not Germany. Or, use an asterisk (*) to return a set consisting of the records that belong to both sets.

```
Sum({$<Year={2015}>-1<Country={'Germany'}>}Sales)
```

```
Sum({$<Year={2015}>*1<Country={'Germany'}>}Sales)
```

KPIs using operators



Set expression tutorial data

Load script

Load the following data as an inline load and then create the chart expressions in the tutorial.

```
//Create table SalesByCountry
SalesByCountry:
Load * Inline [
Country, Year, Sales
Argentina, 2016, 66295.03
Argentina, 2015, 140037.89
Austria, 2016, 54166.09
Austria, 2015, 182739.87
```

```
Belgium, 2016, 182766.87
Belgium, 2015, 178042.33
Brazil, 2016, 174492.67
Brazil, 2015, 2104.22
Canada, 2016, 101801.33
Canada, 2015, 40288.25
Denmark, 2016, 45273.25
Denmark, 2015, 106938.41
Finland, 2016, 107565.55
Finland, 2015, 30583.44
France, 2016, 115644.26
France, 2015, 30696.98
Germany, 2016, 8775.18
Germany, 2015, 77185.68
];
```

Syntax for set expressions

The full syntax (not including the optional use of standard brackets to define precedence) is described using Backus-Naur Formalism:

```
set_expression ::= { set_entity { set_operator set_entity } }
set_entity ::= set_identifier [ set_modifier ] | set_modifier
set_identifier ::= 1 | $ | $N | $_N | bookmark_id | bookmark_name
set_operator ::= + | - | * | /
set_modifier ::= < field_selection {, field_selection } >
field_selection ::= field_name [ = | += | -= | *= | /= ] element_set_
expression
element_set_expression ::= [ - ] element_set { set_operator element_set }
element_set ::= [ field_name ] | { element_list } | element_function
element_list ::= element { , element }
element_function ::= ( P | E ) ( [set_expression] [field_name] )
element ::= field_value | " search_mask "
```

3.3 General syntax for chart expressions

The following general syntax structure can be used for chart expressions, with many optional parameters:

```
expression ::= ( constant | expressionname | operator1 expression | expression operator2
expression | function | aggregation function | (expression ) )
```

where:

constant is a string (a text, a date or a time) enclosed by single straight quotation marks, or a number. Constants are written without thousands separator and with a decimal point as decimal separator.

expressionname is the name (label) of another expression in the same chart.

operator1 is a unary operator (working on one expression, the one to the right).

operator2 is a binary operator (working on two expressions, one on each side).

```
function ::= functionname ( parameters )
parameters ::= expression { , expression }
```

The number and types of parameters are not arbitrary. They depend on the function used.

```
aggregationfunction ::= aggregationfunctionname ( parameters2 )
parameters2 ::= aggrexpression { , aggrexpression }
```

The number and types of parameters are not arbitrary. They depend on the function used.

3.4 General syntax for aggregations

The following general syntax structure can be used for aggregations, with many optional parameters:

```
aggrexpression ::= ( fieldref | operator1 aggrexpression | aggrexpression operator2
aggrexpression | functioninaggr | ( aggrexpression ) )
```

fieldref is a field name.

```
functionaggr ::= functionname ( parameters2 )
```

Expressions and functions can thus be nested freely, as long as **fieldref** is always enclosed by exactly one aggregation function and provided the expression returns an interpretable value, Qlik Sense does not give any error messages.

4 Operators

This section describes the operators that can be used in Qlik Sense. There are two types of operators:

- Unary operators (take only one operand)
- Binary operators (take two operands)

Most operators are binary.

The following operators can be defined:

- Bit operators
- Logical operators
- Numeric operators
- Relational operators
- String operators

4.1 Bit operators

All bit operators convert (truncate) the operands to signed integers (32 bit) and return the result in the same way. All operations are performed bit by bit. If an operand cannot be interpreted as a number, the operation will return NULL.

Bit operators

Operator	Full name	Description
bitnot	Bit inverse.	Unary operator. The operation returns the logical inverse of the operand performed bit by bit. **Example:** `bitnot 17` returns -18
bitand	Bit and.	The operation returns the logical AND of the operands performed bit by bit. **Example:** `17 bitand 7` returns 1
bitor	Bit or.	The operation returns the logical OR of the operands performed bit by bit. **Example:** `17 bitor 7` returns 23

Operator	Full name	Description
bitxor	Bit exclusive or.	The operation returns the logical exclusive or of the operands performed bit by bit. **Example:** `17 bitxor 7` returns 22
>>	Bit right shift.	The operation returns the first operand shifted to the right. The number of steps is defined in the second operand. **Example:** `8 >> 2` returns 2
<<	Bit left shift.	The operation returns the first operand shifted to the left. The number of steps is defined in the second operand. **Example:** `8 << 2` returns 32

4.2 Logical operators

All logical operators interpret the operands logically and return True (-1) or False (0) as result.

Logical operators

Operator	Description
not	Logical inverse. One of the few unary operators. The operation returns the logical inverse of the operand.
and	Logical and. The operation returns the logical and of the operands.
or	Logical or. The operation returns the logical or of the operands.
Xor	Logical exclusive or. The operation returns the logical exclusive or of the operands. I.e. like logical or, but with the difference that the result is False if both operands are True.

4.3 Numeric operators

All numeric operators use the numeric values of the operands and return a numeric value as result.

Numeric operators

Operator	Description
+	Sign for positive number (unary operator) or arithmetic addition. The binary operation returns the sum of the two operands.
-	Sign for negative number (unary operator) or arithmetic subtraction. The unary operation returns the operand multiplied by -1, and the binary the difference between the two operands.
*	Arithmetic multiplication. The operation returns the product of the two operands.
/	Arithmetic division. The operation returns the ratio between the two operands.

4.4 Relational operators

All relational operators compare the values of the operands and return True (-1) or False (0) as the result. All relational operators are binary.

Relational operators

Operator	Description
<	Less than. A numeric comparison is made if both operands can be interpreted numerically. The operation returns the logical value of the evaluation of the comparison.
<=	Less than or equal. A numeric comparison is made if both operands can be interpreted numerically. The operation returns the logical value of the evaluation of the comparison.
>	Greater than. A numeric comparison is made if both operands can be interpreted numerically. The operation returns the logical value of the evaluation of the comparison.
>=	Greater than or equal. A numeric comparison is made if both operands can be interpreted numerically. The operation returns the logical value of the evaluation of the comparison.
=	Equals. A numeric comparison is made if both operands can be interpreted numerically. The operation returns the logical value of the evaluation of the comparison.
<>	Not equivalent to. A numeric comparison is made if both operands can be interpreted numerically. The operation returns the logical value of the evaluation of the comparison.

Operator	Description
precedes	Unlike the **<** operator no attempt is made to make a numeric interpretation of the argument values before the comparison. The operation returns true if the value to the left of the operator has a text representation which, in string comparison, comes before the text representation of the value on the right. **Example:** `'1 ' precedes ' 2'` returns FALSE `' 1' precedes ' 2'` returns TRUE as the ASCII value of a space (' ') is of less value than the ASCII value of a number. Compare this to: `'1 ' < ' 2'` returns TRUE `' 1' < ' 2'` returns TRUE
follows	Unlike the **>** operator no attempt is made to make a numeric interpretation of the argument values before the comparison. The operation returns true if the value to the left of the operator has a text representation which, in string comparison, comes after the text representation of the value on the right. **Example:** `' 2' follows '1 '` returns FALSE ' 2' follows ' 1' returns TRUE as the ASCII value of a space (' ') is of less value than the ASCII value of a number. Compare this to: `' 2' > ' 1'` returns TRUE `' 2' > '1 '` returns TRUE

4.5 String operators

There are two string operators. One uses the string values of the operands and return a string as result. The other one compares the operands and returns a boolean value to indicate match.

&

String concatenation. The operation returns a text string, that consists of the two operand strings, one after another.

Example:

`'abc' & 'xyz'` returns 'abcxyz'

like

String comparison with wildcard characters. The operation returns a boolean True (-1) if the string before the operator is matched by the string after the operator. The second string may contain the wildcard characters * (any number of arbitrary characters) or ? (one arbitrary character).

Example:

`'abc' like 'a*'` returns True (-1)

`'abcd' like 'a?c*'` returns True (-1)

`'abc' like 'a??bc'` returns False (0)

5 Script and chart functions

Transform and aggregate data using functions in data load scripts and chart expressions.

Many functions can be used in the same way in both data load scripts and chart expressions, but there are a number of exceptions:

- Some functions can only be used in data load scripts, denoted by - script function.
- Some functions can only be used in chart expressions, denoted by - chart function.
- Some functions can be used in both data load scripts and chart expressions, but with differences in parameters and application. These are described in separate topics denoted by - script function or - chart function.

5.1 Analytic connections for server-side extensions (SSE)

Functions enabled by analytic connections will only be visible if you have configured the analytic connections and Qlik Sense has started.

You configure the analytic connections in the QMC, see the topic " Creating an analytic connection" in the guide Manage Qlik Sense sites.

In Qlik Sense Desktop, you configure the analytic connections by editing the *Settings.ini* file, see the topic " Configuring analytic connections in Qlik Sense Desktop" in the guide Qlik Sense Desktop.

5.2 Aggregation functions

The family of functions known as aggregation functions consists of functions that take multiple field values as their input and return a single result per group, where the grouping is defined by a chart dimension or a **group by** clause in the script statement.

Aggregation functions include **Sum()**, **Count()**, **Min()**, **Max()**, and many more.

Most aggregation functions can be used in both the data load script and chart expressions, but the syntax differs.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

When naming an entity, avoid assigning the same name to more than one field, variable, or measure. There is a strict order of precedence for resolving conflicts between entities with identical names. This order is reflected in any objects or contexts in which these entities are used. This order of precedence is as follows:

- Inside an aggregation, a field has precedence over a variable. Measure labels are not relevant in aggregations and are not prioritized.

- Outside an aggregation, a measure label has precedence over a variable, which in turn has precedence over a field name.
- Additionally, outside an aggregation, a measure can be re-used by referencing its label, unless the label is in fact a calculated one. In that situation, the measure drops in significance in order to reduce risk of self-reference, and in this case the name will always be interpreted first as a measure label, second as a field name, and third as a variable name.

Using aggregation functions in a data load script

Aggregation functions can only be used inside **LOAD** and **SELECT** statements.

Using aggregation functions in chart expressions

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

An aggregation function aggregates over the set of possible records defined by the selection. However, an alternative set of records can be defined by using a set expression in set analysis.

How aggregations are calculated

An aggregation loops over the records of a specific table, aggregating the records in it. For example, **Count** (<Field>) will count the number of records in the table where <Field> resides. Should you want to aggregate just the distinct field values, you need to use the **distinct** clause, such as **Count**(**distinct** <Field>).

If the aggregation function contains fields from different tables, the aggregation function will loop over the records of the cross product of the tables of the constituent fields. This has a performance penalty, and for this reason such aggregations should be avoided, particularly when you have large amounts of data.

Aggregation of key fields

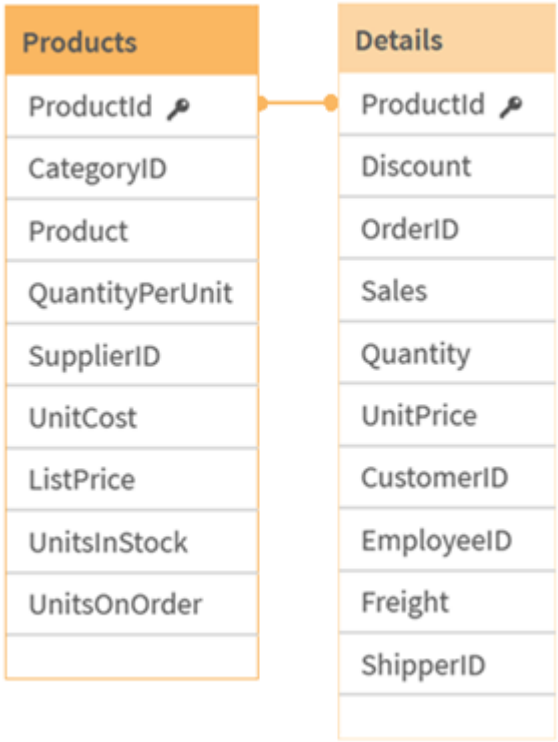
The way aggregations are calculated means that you cannot aggregate key fields because it is not clear which table should be used for the aggregation. For example, if the field <Key> links two tables, it is not clear whether **Count**(<Key>) should return the number of records from the first or the second table.

However, if you use the **distinct** clause, the aggregation is well-defined and can be calculated.

So, if you use a key field inside an aggregation function without the **distinct** clause, Qlik Sense will return a number which may be meaningless. The solution is to either use the **distinct** clause, or use a copy of the key – a copy that resides in one table only.

For example, in the following tables, ProductID is the key between the tables.

ProductID key between Products and Details tables



Count(ProductID) can be counted either in the Products table (which has only one record per product – ProductID is the primary key) or it can be counted in the Details table (which most likely has several records per product). If you want to count the number of distinct products, you should use Count(distinct ProductID). If you want to count the number of rows in a specific table, you should not use the key.

Basic aggregation functions

Basic aggregation functions overview

Basic aggregation functions are a group of the most common aggregation functions.

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Basic aggregation functions in the data load script

FirstSortedValue

FirstSortedValue() returns the value from the expression specified in **value** that corresponds to the result of sorting the **sort_weight** argument, for example, the name of the product with the lowest unit price. The nth value in the sort order, can be specified in **rank**. If more than one resulting value shares the same **sort_weight** for the specified **rank**, the function returns NULL. The sorted values are iterated over a number of records, as defined by a **group by** clause, or aggregated across the full data set if no **group by** clause is defined.

```
FirstSortedValue ([ distinct ] expression, sort_weight [, rank ])
```

Max

Max() finds the highest numeric value of the aggregated data in the expression, as defined by a **group by** clause. By specifying a **rank** n, the nth highest value can be found.

```
Max ( expression[, rank])
```

Min

Min() returns the lowest numeric value of the aggregated data in the expression, as defined by a **group by** clause. By specifying a **rank** n, the nth lowest value can be found.

```
Min ( expression[, rank])
```

Mode

Mode() returns the most commonly-occurring value, the mode value, of the aggregated data in the expression, as defined by a **group by** clause. The **Mode()** function can return numeric values as well as text values.

```
Mode (expression )
```

Only

Only() returns a value if there is one and only one possible result from the aggregated data. If records contain only one value then that value is returned, otherwise NULL is returned. Use the **group by** clause to evaluate over multiple records. The **Only()** function can return numeric and text values.

```
Only (expression )
```

Sum

Sum() calculates the total of the values aggregated in the expression, as defined by a **group by** clause.

```
Sum ([distinct]expression)
```

Basic aggregation functions in chart expressions

Chart aggregation functions can only be used on fields in chart expressions. The argument expression of one aggregation function must not contain another aggregation function.

FirstSortedValue

FirstSortedValue() returns the value from the expression specified in **value** that corresponds to the result of sorting the **sort_weight** argument, for example, the name of the product with the lowest unit price. The nth value in the sort order, can be specified in **rank**. If more than one resulting value shares the same **sort_weight** for the specified **rank**, the function returns NULL.

```
FirstSortedValue - chart function([{SetExpression}] [DISTINCT] [TOTAL [<fld
{,fld}>]] value, sort_weight [,rank])
```

Max

Max() finds the highest value of the aggregated data. By specifying a **rank** n, the nth highest value can be found.

```
Max - chart functionMax() finds the highest value of the aggregated data. By
specifying a rank n, the nth highest value can be found. You might also want
to look at FirstSortedValue and rangemax, which have similar functionality to
the Max function. Max([{SetExpression}] [TOTAL [<fld {,fld}>]] expr [,rank])
```

```
numeric ArgumentsArgumentDescriptionexprThe expression or field containing
the data to be measured.rankThe default value of rank is 1, which corresponds
to the highest value. By specifying rank as 2, the second highest value is
returned. If rank is 3, the third highest value is returned, and so
on.SetExpressionBy default, the aggregation function will aggregate over the
set of possible records defined by the selection. An alternative set of
records can be defined by a set analysis expression. TOTALIf the word TOTAL
occurs before the function arguments, the calculation is made over all
possible values given the current selections, and not just those that pertain
to the current dimensional value, that is, it disregards the chart
dimensions. By using TOTAL [<fld {.fld}>], where the TOTAL qualifier is
followed by a list of one or more field names as a subset of the chart
dimension variables, you create a subset of the total possible
values. DataCustomerProductUnitSalesUnitPrice
AstridaAA416AstridaAA1015AstridaBB99BetacabBB510BetacabCC220BetacabDD-
25CanutilityAA815CanutilityCC-19Examples and resultsExamplesResultsMax
(UnitSales)10, because this is the highest value in UnitSales.The value of an
order is calculated from the number of units sold in (UnitSales) multiplied
by the unit price.Max(UnitSales*UnitPrice)150, because this is the highest
value of the result of calculating all possible values of (UnitSales)*
(UnitPrice).Max(UnitSales, 2)9, which is the second highest value.Max
(TOTAL UnitSales)10, because the TOTAL qualifier means the highest possible
value is found, disregarding the chart dimensions. For a chart with Customer
as dimension, the TOTAL qualifier will ensure the maximum value across the
full dataset is returned, instead of the maximum UnitSales for each
customer.Make the selection Customer B.Max({1} TOTAL UnitSales)10,
independent of the selection made, because the Set Analysis expression {1}
defines the set of records to be evaluated as ALL, no matter what selection
is made.Data used in examples:ProductData:LOAD * inline
[Customer|Product|UnitSales|UnitPriceAstrida|AA|4|16Astrida|AA|10|15Astrida|B
B|9|9Betacab|BB|5|10Betacab|CC|2|20Betacab|DD||25Canutility|AA|8|15Canutility
|CC||19] (delimiter is '|'); FirstSortedValue RangeMax ([{SetExpression}]
[DISTINCT] [TOTAL [<fld {,fld}>]] expr [,rank])
```

Min

Min() finds the lowest value of the aggregated data. By specifying a **rank** n, the nth lowest value can be found.

```
Min - chart function([{SetExpression}] [DISTINCT] [TOTAL [<fld {,fld}>]] expr
[,rank])
```

Mode

Mode() finds the most commonly-occurring value, the mode value, in the aggregated data. The **Mode()** function can process text values as well as numeric values.

```
Mode - chart function ({[SetExpression] [TOTAL [<fld {,fld}>]]} expr)
```

Only

Only() returns a value if there is one and only one possible result from the aggregated data. For example, searching for the only product where the unit price =9 will return NULL if more than one product has a unit price of 9.

```
Only - chart function([{SetExpression}] [DISTINCT] [TOTAL [<fld {,fld}>]]
expr)
```

Sum

Sum() calculates the total of the values given by the expression or field across the aggregated data.

```
Sum - chart function([{SetExpression}] [DISTINCT] [TOTAL [<fld {,fld}>]]
expr])
```

FirstSortedValue

FirstSortedValue() returns the value from the expression specified in **value** that corresponds to the result of sorting the **sort_weight** argument, for example, the name of the product with the lowest unit price. The nth value in the sort order, can be specified in **rank**. If more than one resulting value shares the same **sort_weight** for the specified **rank**, the function returns NULL. The sorted values are iterated over a number of records, as defined by a **group by** clause, or aggregated across the full data set if no **group by** clause is defined.

Syntax:

```
FirstSortedValue ([ distinct ] value, sort-weight [, rank ])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
value Expression	The function finds the value of the expression **value** that corresponds to the result of sorting **sort_weight**.
sort-weight Expression	The expression containing the data to be sorted. The first (lowest) value of **sort_weight** is found, from which the corresponding value of the **value** expression is determined. If you place a minus sign in front of **sort_weight**, the function returns the last (highest) sorted value instead.
rank Expression	By stating a **rank** "n" larger than 1, you get the nth sorted value.
distinct	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

Scripting examples

Example	Result

`Temp:`
`LOAD * inline [`
`Customer|Product|OrderNumber|UnitSales|CustomerID`
`Astrida|AA|1|10|1`
`Astrida|AA|7|18|1`
`Astrida|BB|4|9|1`
`Astrida|CC|6|2|1`
`Betacab|AA|5|4|2`
`Betacab|BB|2|5|2`
`Betacab|DD|12|25|2`
`Canutility|AA|3|8|3`
`Canutility|CC|13|19|3`
`Divadip|AA|9|16|4`
`Divadip|AA|10|16|4`
`Divadip|DD|11|10|4`
`] (delimiter is '|');`
 	
`FirstSortedValue:`
`LOAD Customer,FirstSortedValue(Product, UnitSales)`
`as MyProductWithSmallestOrderByCustomer Resident`
`Temp Group By Customer;`
	`Customer`
	`MyProductWithSmallestOrderByCustomer`
	`Astrida CC`
	`Betacab AA`
	`Canutility AA`
	`Divadip DD`
	The function sorts UnitSales from smallest to largest, looking for the value of Customer with the smallest value of UnitSales, the smallest order.
	
	Because CC corresponds to the smallest order (value of UnitSales=2) for customer Astrida. AA corresponds to the smallest order (4) for customer Betacab, AA corresponds to the smallest order (8) for customer Canutility, and DD corresponds to the smallest order (10) for customer Divadip..

Given that the **Temp** table is loaded as in the previous example:
 	
`LOAD Customer,FirstSortedValue(Product, -UnitSales)`
`as MyProductWithLargestOrderByCustomer Resident`
`Temp Group By Customer;`
	`Customer`
	`MyProductWithLargestOrderByCustomer`
	`Astrida AA`
	`Betacab DD`
	`Canutility CC`
	`Divadip -`
	A minus sign precedes the sort_weight argument, so the function sorts the largest first.
	
	Because AA corresponds to the largest order (value of UnitSales:18) for customer Astrida, DD corresponds to the largest order (12) for customer Betacab, and CC corresponds to the largest order (13) for customer Canutility. There are two identical values for the largest order (16) for customer Divadip, therefore this produces a null result.

Example	Result

Given that the **Temp** table is loaded as in the previous example:
`LOAD Customer,FirstSortedValue(distinct Product, -UnitSales) as MyProductWithSmallestOrderByCustomer Resident Temp Group By Customer;`
	`Customer MyProductWithLargestOrderByCustomer`
	`Astrida AA`
	`Betacab DD`
	`Canutility CC`
	`Divadip AA`
	This is the same as the previous example, except the distinct qualifier is used. This causes the duplicate result for Divadip to be disregarded, allowing a non-null value to be returned.

FirstSortedValue - chart function

FirstSortedValue() returns the value from the expression specified in **value** that corresponds to the result of sorting the **sort_weight** argument, for example, the name of the product with the lowest unit price. The nth value in the sort order, can be specified in **rank**. If more than one resulting value shares the same **sort_weight** for the specified **rank**, the function returns NULL.

Syntax:

```
FirstSortedValue([{SetExpression}] [DISTINCT] [TOTAL [<fld {,fld}>]] value, sort_weight [,rank])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
value	Output field. The function finds the value of the expression **value** that corresponds to the result of sorting **sort_weight**.
sort_weight	Input field. The expression containing the data to be sorted. The first (lowest) value of **sort_weight** is found, from which the corresponding value of the **value** expression is determined. If you place a minus sign in front of **sort_weight**, the function returns the last (highest) sorted value instead.
rank	By stating a **rank** "n" larger than 1, you get the nth sorted value.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.

Argument	Description
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Data

Customer	Product	UnitSales	UnitPrice
Astrida	AA	4	16
Astrida	AA	10	15
Astrida	BB	9	9
Betacab	BB	5	10
Betacab	CC	2	20
Betacab	DD	-	25
Canutility	AA	8	15
Canutility	CC	-	19

Examples and results

Example	Result
`firstsortedvalue (Product, UnitPrice)`	BB, which is the `Product`with the lowest `UnitPrice`(9).
`firstsortedvalue (Product, UnitPrice, 2)`	BB, which is the `Product`with the second-lowest `UnitPrice`(10).
`firstsortedvalue (Customer, -UnitPrice, 2)`	Betacab, which is the `Customer`with the `Product` that has second-highest `UnitPrice`(20).
`firstsortedvalue (Customer, UnitPrice, 3)`	NULL, because there are two values of `Customer` (Astrida and Canutility) with the samerank (third-lowest) `UnitPrice`(15). Use the distinct qualifier to make sure unexpected null results do not occur.
`firstsortedvalue (Customer, -UnitPrice*UnitSales, 2)`	Canutility, which is the `Customer` with the second-highest sales order value `UnitPrice` multiplied by `UnitSales` (120).

Data used in examples:

```
ProductData:
LOAD * inline [
Customer|Product|UnitSales|UnitPrice
Astrida|AA|4|16
Astrida|AA|10|15
Astrida|BB|9|9
Betacab|BB|5|10
Betacab|CC|2|20
Betacab|DD||25
Canutility|AA|8|15
Canutility|CC||19
] (delimiter is '|');
```

Max

Max() finds the highest numeric value of the aggregated data in the expression, as defined by a **group by** clause. By specifying a **rank** n, the nth highest value can be found.

Syntax:

```
Max ( expr [, rank])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr Expression	The expression or field containing the data to be measured.
rank Expression	The default value of **rank** is 1, which corresponds to the highest value. By specifying **rank** as 2, the second highest value is returned. If **rank** is 3, the third highest value is returned, and so on.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

Example:

```
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|CustomerID
Astrida|AA|1|10|1
Astrida|AA|7|18|1
Astrida|BB|4|9|1
Astrida|CC|6|2|1
Betacab|AA|5|4|2
Betacab|BB|2|5|2
```

```
Betacab|DD
Canutility|DD|3|8
Canutility|CC
] (delimiter is '|');

Max:
LOAD Customer, Max(UnitSales) as MyMax Resident Temp Group By Customer;
```

Resulting table

Customer	MyMax
Astrida	18
Betacab	5
Canutility	8

Example:

Given that the **Temp** table is loaded as in the previous example:

```
LOAD Customer, Max(UnitSales,2) as MyMaxRank2 Resident Temp Group By Customer;
```

Resulting table

Customer	MyMaxRank2
Astrida	10
Betacab	4
Canutility	-

Max - chart function

Max() finds the highest value of the aggregated data. By specifying a **rank** n, the nth highest value can be found.

*You might also want to look at **FirstSortedValue** and **rangemax**, which have similar functionality to the **Max** function.*

Syntax:

```
Max([{SetExpression}] [TOTAL [<fld {,fld}>]] expr [,rank])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.

Argument	Description
rank	The default value of **rank** is 1, which corresponds to the highest value. By specifying **rank** as 2, the second highest value is returned. If **rank** is 3, the third highest value is returned, and so on.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Data

Customer	Product	UnitSales	UnitPrice
Astrida	AA	4	16
Astrida	AA	10	15
Astrida	BB	9	9
Betacab	BB	5	10
Betacab	CC	2	20
Betacab	DD	-	25
Canutility	AA	8	15
Canutility	CC	-	19

Examples and results

Examples	Results
`Max(UnitSales)`	10, because this is the highest value in `UnitSales`.

Examples	Results
The value of an order is calculated from the number of units sold in (`UnitSales`) multiplied by the unit price. `Max (UnitSales*UnitPrice)`	150, because this is the highest value of the result of calculating all possible values of (`UnitSales`)*(`UnitPrice`).
`Max(UnitSales, 2)`	9, which is the second highest value.
`Max(TOTAL UnitSales)`	10, because the TOTAL qualifier means the highest possible value is found, disregarding the chart dimensions. For a chart with Customer as dimension, the TOTAL qualifier will ensure the maximum value across the full dataset is returned, instead of the maximum UnitSales for each customer.
Make the selection `Customer B`. `Max({1} TOTAL UnitSales)`	10, independent of the selection made, because the Set Analysis expression {1} defines the set of records to be evaluated as ALL, no matter what selection is made.

Data used in examples:

```
ProductData:
LOAD * inline [
Customer|Product|UnitSales|UnitPrice
Astrida|AA|4|16
Astrida|AA|10|15
Astrida|BB|9|9
Betacab|BB|5|10
Betacab|CC|2|20
Betacab|DD||25
Canutility|AA|8|15
Canutility|CC||19
] (delimiter is '|');
```

See also:

FirstSortedValue - chart function (page 271)
RangeMax (page 915)

Min

Min() returns the lowest numeric value of the aggregated data in the expression, as defined by a **group by** clause. By specifying a **rank** n, the nth lowest value can be found.

Syntax:

```
Min ( expr [, rank])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr Expression	The expression or field containing the data to be measured.
rank Expression	The default value of **rank** is 1, which corresponds to the lowest value. By specifying **rank** as 2, the second lowest value is returned. If **rank** is 3, the third lowest value is returned, and so on.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

Example:

```
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|CustomerID
Astrida|AA|1|10|1
Astrida|AA|7|18|1
Astrida|BB|4|9|1
Astrida|CC|6|2|1
Betacab|AA|5|4|2
Betacab|BB|2|5|2
Betacab|DD
Canutility|DD|3|8
Canutility|CC
] (delimiter is '|');
Min:
LOAD Customer, Min(UnitSales) as MyMin Resident Temp Group By Customer;
```

Resulting table

Customer	MyMin
Astrida	2
Betacab	4
Canutility	8

Example:

Given that the **Temp** table is loaded as in the previous example:

```
LOAD Customer, Min(UnitSales,2) as MyMinRank2 Resident Temp Group By Customer;
```

Resulting table

Customer	MyMinRank2
Astrida	9
Betacab	5
Canutility	-

Min - chart function

Min() finds the lowest value of the aggregated data. By specifying a **rank** n, the nth lowest value can be found.

> *You might also want to look at **FirstSortedValue** and **rangemin**, which have similar functionality to the **Min** function.*

Syntax:

```
Min({[SetExpression] [TOTAL [<fld {,fld}>]]} expr [,rank])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
rank	The default value of **rank** is 1, which corresponds to the lowest value. By specifying **rank** as 2, the second lowest value is returned. If **rank** is 3, the third lowest value is returned, and so on.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Data

Customer	Product	UnitSales	UnitPrice
Astrida	AA	4	16
Astrida	AA	10	15
Astrida	BB	9	9
Betacab	BB	5	10
Betacab	CC	2	20
Betacab	DD	-	25
Canutility	AA	8	15
Canutility	CC	-	19

The Min() function must return a non-NULL value from the array of values given by the expression, if there is one. So in the examples, because there are NULL values in the data, the function returns the first non-NULL value evaluated from the expression.

Examples and results

Examples	Results
`Min(UnitSales)`	2, because this is the lowest non-NULL value in `UnitSales`.
The value of an order is calculated from the number of units sold in (`UnitSales`) multiplied by the unit price. `Min (UnitSales*UnitPrice)`	40, because this is the lowest non-NULL value result of calculating all possible values of (`UnitSales`)*(`UnitPrice`).
`Min(UnitSales, 2)`	4, which is the second lowest value (after the NULL values).
`Min(TOTAL UnitSales)`	2, because the TOTAL qualifier means the lowest possible value is found, disregarding the chart dimensions. For a chart with Customer as dimension, the TOTAL qualifier will ensure the minimum value across the full dataset is returned, instead of the minimum UnitSales for each customer.
Make the selection Customer B. `Min({1} TOTAL UnitSales)`	2, which is independent of the selection of Customer B. The Set Analysis expression {1} defines the set of records to be evaluated as ALL, no matter what selection is made.

Data used in examples:

```
ProductData:
LOAD * inline [
Customer|Product|UnitSales|UnitPrice
Astrida|AA|4|16
Astrida|AA|10|15
Astrida|BB|9|9
Betacab|BB|5|10
Betacab|CC|2|20
Betacab|DD||25
Canutility|AA|8|15
Canutility|CC||19
] (delimiter is '|');
```

See also:

FirstSortedValue - chart function (page 271)
RangeMin (page 918)

Mode

Mode() returns the most commonly-occurring value, the mode value, of the aggregated data in the expression, as defined by a **group by** clause. The **Mode()** function can return numeric values as well as text values.

Syntax:

```
Mode ( expr)
```

Return data type: dual

Arguments

Argument	Description
expr Expression	The expression or field containing the data to be measured.

Limitations:

If more than one value is equally commonly occurring, NULL is returned.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

Scripting examples

Example	Result

`Temp:`
`LOAD * inline [`
`Customer|Product|OrderNumber|UnitSales|CustomerID`
`Astrida|AA|1|10|1`
`Astrida|AA|7|18|1`
`Astrida|BB|4|9|1`
`Astrida|CC|6|2|1`
`Betacab|AA|5|4|2`
`Betacab|BB|2|5|2`
`Betacab|DD`
`Canutility|DD|3|8`
`Canutility|CC`
`] (delimiter is '|');`
 	
`Mode:`
`LOAD Customer, Mode(Product) as MyMostOftenSoldProduct`
`Resident Temp Group By Customer;`
	MyMostOftenSoldProduct
	
	AA
	
	because AA is the only product sold more than once.

Mode - chart function

Mode() finds the most commonly-occurring value, the mode value, in the aggregated data. The **Mode()** function can process text values as well as numeric values.

Syntax:

```
Mode({[SetExpression] [TOTAL [<fld {,fld}>]]} expr)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Data

Customer	Product	UnitSales	UnitPrice
Astrida	AA	4	16
Astrida	AA	10	15
Astrida	BB	9	9
Betacab	BB	5	10
Betacab	CC	2	20
Betacab	DD	-	25
Canutility	AA	8	15
Canutility	CC	-	19

Examples and results

Examples	Results
`Mode(UnitPrice)` Make the selection `Customer A`.	15, because this is the most commonly-occurring value in `UnitSales`. Returns NULL (-). No single value occurs more often than another.
`Mode(Product)` Make the selection `Customer A`	AA, because this is the most commonly occurring value in `Product`. Returns NULL (-). No single value occurs more often than another.
`Mode (TOTAL UnitPrice)`	15, because the TOTAL qualifier means the most commonly occurring value is still 15, even disregarding the chart dimensions.
Make the selection `Customer B`. `Mode({1} TOTAL UnitPrice)`	15, independent of the selection made, because the Set Analysis expression {1} defines the set of records to be evaluated as ALL, no matter what selection is made.

Data used in examples:

```
ProductData:
LOAD * inline [
Customer|Product|UnitSales|UnitPrice
Astrida|AA|4|16
Astrida|AA|10|15
Astrida|BB|9|9
Betacab|BB|5|10
Betacab|CC|2|20
Betacab|DD||25
Canutility|AA|8|15
Canutility|CC||19
```

```
] (delimiter is '|');
```

See also:

Avg - chart function (page 324)
Median - chart function (page 361)

Only

Only() returns a value if there is one and only one possible result from the aggregated data. If records contain only one value then that value is returned, otherwise NULL is returned. Use the **group by** clause to evaluate over multiple records. The **Only()** function can return numeric and text values.

Syntax:

```
Only ( expr )
```

Return data type: dual

Arguments

Argument	Description
expr Expression	The expression or field containing the data to be measured.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

```
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|CustomerID
Astrida|AA|1|10|1
Astrida|AA|7|18|1
Astrida|BB|4|9|1
Astrida|CC|6|2|1
Betacab|AA|5|4|2
Betacab|BB|2|5|2
Betacab|DD
Canutility|DD|3|8
Canutility|CC
] (delimiter is '|');
Only:
LOAD Customer, Only(CustomerID) as MyUniqIDCheck Resident Temp Group By Customer;
```

Resulting table

Customer	MyUniqIDCheck
Astrida	1 because only customer Astrida has complete records that include CustomerID.

Only - chart function

Only() returns a value if there is one and only one possible result from the aggregated data. For example, searching for the only product where the unit price =9 will return NULL if more than one product has a unit price of 9.

Syntax:

```
Only([{SetExpression}] [TOTAL [<fld {,fld}>]] expr)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Use Only() when you want a NULL result if there are multiple possible values in the sample data.

Examples and results:

Data

Customer	Product	UnitSales	UnitPrice
Astrida	AA	4	16
Astrida	AA	10	15

Customer	Product	UnitSales	UnitPrice
Astrida	BB	9	9
Betacab	BB	5	10
Betacab	CC	2	20
Betacab	DD	-	25
Canutility	AA	8	15
Canutility	CC	-	19

Examples and results

Examples	Results
`Only({<UnitPrice={9}>} Product)`	BB, because this is the only `Product`that has a `UnitPrice` of '9'.
`Only({<Product={DD}>} Customer)`	Betacab, because it is the only `Customer` selling a `Product` called 'DD'.
`Only({<UnitPrice={20}>} UnitSales)`	The number of `UnitSales` where `UnitPrice` is 20 is 2, because there is only one value of `UnitSales` where the `UnitPrice` =20.
`Only({<UnitPrice={15}>} UnitSales)`	NULL, because there are two values of `UnitSales` where the `UnitPrice` =15.

Data used in examples:

```
ProductData:
LOAD * inline [
Customer|Product|UnitSales|UnitPrice
Astrida|AA|4|16
Astrida|AA|10|15
Astrida|BB|9|9
Betacab|BB|5|10
Betacab|CC|2|20
Betacab|DD||25
Canutility|AA|8|15
Canutility|CC||19
] (delimiter is '|');
```

Sum

Sum() calculates the total of the values aggregated in the expression, as defined by a **group by** clause.

Syntax:

```
sum ( [ distinct] expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
distinct	If the word **distinct** occurs before the expression, all duplicates will be disregarded.
expr Expression	The expression or field containing the data to be measured.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

```
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|CustomerID
Astrida|AA|1|10|1
Astrida|AA|7|18|1
Astrida|BB|4|9|1
Astrida|CC|6|2|1
Betacab|AA|5|4|2
Betacab|BB|2|5|2
Betacab|DD
Canutility|DD|3|8
Canutility|CC
] (delimiter is '|');
Sum:
LOAD Customer, Sum(UnitSales) as MySum Resident Temp Group By Customer;
```

Resulting table

Customer	MySum
Astrida	39
Betacab	9
Canutility	8

Sum - chart function

Sum() calculates the total of the values given by the expression or field across the aggregated data.

Syntax:

```
Sum([{SetExpression}] [DISTINCT] [TOTAL [<fld {,fld}>]] expr])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded. *Although the DISTINCT qualifier is supported, use it only with extreme caution because it may mislead the reader into thinking a total value is shown when some data has been omitted.*
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Data

Customer	Product	UnitSales	UnitPrice
Astrida	AA	4	16
Astrida	AA	10	15
Astrida	BB	9	9
Betacab	BB	5	10
Betacab	CC	2	20
Betacab	DD	-	25
Canutility	AA	8	15
Canutility	CC	-	19

Examples and results

Examples	Results
`Sum(UnitSales)`	38. The total of the values in `UnitSales`.
`Sum(UnitSales*UnitPrice)`	505. The total of `UnitPrice` multiplied by `UnitSales` aggregated.
`Sum (TOTAL UnitSales*UnitPrice)`	505 for all rows in the table as well as the total, because the TOTAL qualifier means the sum is still 505, disregarding the chart dimensions.
Make the selection `Customer B`. `Sum({1} TOTAL UnitSales*UnitPrice)`	505, independent of the selection made, because the Set Analysis expression {1} defines the set of records to be evaluated as ALL, no matter what selection is made.

Data used in examples:

```
ProductData:
LOAD * inline [
Customer|Product|UnitSales|UnitPrice
Astrida|AA|4|16
Astrida|AA|10|15
Astrida|BB|9|9
Betacab|BB|5|10
Betacab|CC|2|20
Betacab|DD||25
Canutility|AA|8|15
Canutility|CC||19
] (delimiter is '|');
```

Counter aggregation functions

Counter aggregation functions return various types of counts of an expression over a number of records in a data load script, or a number of values in a chart dimension.

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Counter aggregation functions in the data load script

Count

Count() returns the number of values aggregated in expression, as defined by a **group by** clause.

```
Count ([distinct ] expression | * )
```

MissingCount

MissingCount() returns the number of missing values aggregated in the expression, as defined by a **group by** clause.

```
MissingCount ([ distinct ] expression)
```

NullCount

NullCount() returns the number of NULL values aggregated in the expression, as defined by a **group by** clause.

```
NullCount ([ distinct ] expression)
```

NumericCount

NumericCount() returns the number of numeric values found in the expression, as defined by a **group by** clause.

```
NumericCount ([ distinct ] expression)
```

TextCount

TextCount() returns the number of field values that are non-numeric aggregated in the expression, as defined by a **group by** clause.

```
TextCount ([ distinct ] expression)
```

Counter aggregation functions in chart expressions

The following counter aggregation functions can be used in charts:

Count

Count() is used to aggregate the number of values, text and numeric, in each chart dimension.

```
Count - chart function({[SetExpression] [DISTINCT] [TOTAL [<fld {,fld}>]]}
expr)
```

MissingCount

MissingCount() is used to aggregate the number of missing values in each chart dimension. Missing values are all non-numeric values.

```
MissingCount - chart function({[SetExpression] [DISTINCT] [TOTAL [<fld
{,fld}>]] expr)
```

NullCount

NullCount() is used to aggregate the number of NULL values in each chart dimension.

```
NullCount - chart function({[SetExpression][DISTINCT] [TOTAL [<fld {,fld}>]]}
expr)
```

NumericCount

NumericCount() aggregates the number of numeric values in each chart dimension.

```
NumericCount - chart function({[SetExpression] [DISTINCT] [TOTAL [<fld
{,fld}>]]} expr)
```

TextCount

TextCount() is used to aggregate the number of field values that are non-numeric in each chart dimension.

```
TextCount - chart function({[SetExpression] [DISTINCT] [TOTAL [<fld
{,fld}>]]} expr)
```

Count

Count() returns the number of values aggregated in expression, as defined by a **group by** clause.

Syntax:

```
Count( [distinct ] expr)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
distinct	If the word **distinct** occurs before the expression, all duplicates are disregarded.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

Scripting examples

Example	Result

`Temp:`
`LOAD * inline [`
`Customer|Product|OrderNumber|UnitSales|UnitPrice`
`Astrida|AA|1|4|16`
`Astrida|AA|7|10|15`
`Astrida|BB|4|9|9`
`Betacab|CC|6|5|10`
`Betacab|AA|5|2|20`
`Betacab|BB|1|25| 25`
`Canutility|AA|3|8|15`
`Canutility|CC|||19`
`Divadip|CC|2|4|16`
`Divadip|DD|3|1|25`
`] (delimiter is '|');`
 	
`Count1:`
`LOAD Customer,Count(OrderNumber) as`
`OrdersByCustomer Resident Temp Group By Customer;`
	`Customer OrdersByCustomer`
	`Astrida 3`
	`Betacab 3`
	`Canutility 2`
	`Divadip 2`
	As long as the dimension Customer is included in the table on the sheet, otherwise the result for OrdersByCustomer is 3, 2.

Example	Result
Given that the **Temp** table is loaded as in the previous example: `LOAD Count(OrderNumber) as TotalOrderNumber Resident Temp;`	`TotalOrderNumber` 10
Given that the **Temp** table is loaded as in the first example: `LOAD Count(distinct OrderNumber) as TotalOrderNumber Resident Temp;`	`TotalOrderNumber` 8 Because there are two values of OrderNumber with the same value, 1, and one null value.

Count - chart function

Count() is used to aggregate the number of values, text and numeric, in each chart dimension.

Syntax:

```
Count({[SetExpression] [DISTINCT] [TOTAL [<fld {,fld}>]]} expr)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Data

Customer	Product	OrderNumber	UnitSales	Unit Price
Astrida	AA	1	4	16
Astrida	AA	7	10	15
Astrida	BB	4	9	9
Betacab	BB	6	5	10
Betacab	CC	5	2	20
Betacab	DD	1	25	25
Canutility	AA	3	8	15
Canutility	CC			19
Divadip	AA	2	4	16
Divadip	DD	3		25

The following examples assume that all customers are selected, except where stated.

Examples and results

Example	Result
`Count(OrderNumber)`	10, because there are 10 fields that could have a value for OrderNumber, and all records, even empty ones, are counted. *"0" counts as a value and not an empty cell. However, if a measure aggregates to 0 for a dimension, that dimension will not be included in charts.*
`Count(Customer)`	10, because Count evaluates the number of occurrences in all fields.
`Count(DISTINCT [Customer])`	4, because using the Distinct qualifier, Count only evaluates unique occurrences.
Given that customer Canutility is selected `Count (OrderNumber)/Count({1} TOTAL OrderNumber)`	0.2, because the expression returns the number of orders from the selected customer as a percentage of orders from all customers. In this case 2 / 10.

Example	Result
Given that customers Astrida and Canutility are selected `Count(TOTAL <Product> OrderNumber)`	5, because that is the number of orders placed on products for the selected customers only and empty cells are counted.

Data used in examples:

```
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|UnitPrice
Astrida|AA|1|4|16
Astrida|AA|7|10|15
Astrida|BB|4|9|9
Betacab|CC|6|5|10
Betacab|AA|5|2|20
Betacab|BB|1|25| 25
Canutility|AA|3|8|15
Canutility|CC|||19
Divadip|CC|2|4|16
Divadip|DD|3|1|25
] (delimiter is '|');
```

MissingCount

MissingCount() returns the number of missing values aggregated in the expression, as defined by a **group by** clause.

Syntax:

```
MissingCount ( [ distinct ] expr)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
expr Expression	The expression or field containing the data to be measured.
distinct	If the word **distinct** occurs before the expression, all duplicates are disregarded.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

Scripting examples

Example	Result

`Temp:`
`LOAD * inline [`
`Customer|Product|OrderNumber|UnitSales|UnitPrice`
`Astrida|AA|1|4|16`
`Astrida|AA|7|10|15`
`Astrida|BB|4|9|9`
`Betacab|CC|6|5|10`
`Betacab|AA|5|2|20`
`Betacab|BB||| 25`
`Canutility|AA|||15`
`Canutility|CC| ||19`
`Divadip|CC|2|4|16`
`Divadip|DD|3|1|25`
`] (delimiter is '|');`
`MissCount1:`
`LOAD Customer,MissingCount(OrderNumber) as MissingOrdersByCustomer Resident Temp Group By Customer;`
 	
`Load MissingCount(OrderNumber) as TotalMissingCount Resident Temp;`
	`Customer MissingOrdersByCustomer`
	`Astrida 0`
	`Betacab 1`
	`Canutility 2`
	`Divadip 0`
	
	The second statement gives:
	
	`TotalMissingCount`
	`3`
	in a table with that dimension.

Given that the **Temp** table is loaded as in the previous example:
 	
`LOAD MissingCount(distinct OrderNumber) as TotalMissingCountDistinct Resident Temp;`
	`TotalMissingCountDistinct`
	`1`
	Because there is only oneOrderNumber one missing value.

MissingCount - chart function

MissingCount() is used to aggregate the number of missing values in each chart dimension. Missing values are all non-numeric values.

Syntax:

```
MissingCount({[SetExpression] [DISTINCT] [TOTAL [<fld {,fld}>]]} expr)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.

Argument	Description
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Data

Customer	Product	OrderNumber	UnitSales	Unit Price
Astrida	AA	1	4	16
Astrida	AA	7	10	15
Astrida	BB	4	9	9
Betacab	BB	6	5	10
Betacab	CC	5	2	20
Betacab	DD			25
Canutility	AA			15
Canutility	CC			19
Divadip	AA	2	4	16
Divadip	DD	3		25

Examples and results

Example	Result
`MissingCount([OrderNumber])`	3 because 3 of the 10 OrderNumber fields are empty *"0" counts as a value and not an empty cell. However, if a measure aggregates to 0 for a dimension, that dimension will not be included in charts.*
`MissingCount ([OrderNumber])/MissingCount ({1} Total [OrderNumber])`	The expression returns the number of incomplete orders from the selected customer as a fraction of incomplete orders from all customers. There is a total of 3 missing values for OrderNumber for all customers. So, for each Customer that has a missing value for Product the result is 1/3.

Data used in example:

```
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|UnitPrice
Astrida|AA|1|4|16
Astrida|AA|7|10|15
Astrida|BB|4|9|9
Betacab|CC|6|5|10
Betacab|AA|5|2|20
Betacab|BB||| 25
Canutility|AA|||15
Canutility|CC| ||19
Divadip|CC|2|4|16
Divadip|DD|3|1|25
] (delimiter is '|');
```

NullCount

NullCount() returns the number of NULL values aggregated in the expression, as defined by a **group by** clause.

Syntax:

```
NullCount ( [ distinct ] expr)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
expr Expression	The expression or field containing the data to be measured.
distinct	If the word **distinct** occurs before the expression, all duplicates are disregarded.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

Scripting examples

Example	Result

`Set NULLINTERPRET = NULL;`
`Temp:`
`LOAD * inline [`
`Customer|Product|OrderNumber|UnitSales|CustomerID`
`Astrida|AA|1|10|1`
`Astrida|AA|7|18|1`
`Astrida|BB|4|9|1`
`Astrida|CC|6|2|1`
`Betacab|AA|5|4|2`
`Betacab|BB|2|5|2`
`Betacab|DD|||`
`Canutility|AA|3|8|`
`Canutility|CC|NULL||`
`] (delimiter is '|');`
`Set NULLINTERPRET=;`
`NullCount1:`
`LOAD Customer,NullCount(OrderNumber) as NullOrdersByCustomer Resident Temp Group By Customer;`
 	
`LOAD NullCount(OrderNumber) as TotalNullCount Resident Temp;`
	`Customer NullOrdersByCustomer`
	`Astrida 0`
	`Betacab 0`
	`Canutility 1`
	
	The second statement gives:
	
	`TotalNullCount`
	`1`
	in a table with that dimension, because only one record contains a null value.

NullCount - chart function

NullCount() is used to aggregate the number of NULL values in each chart dimension.

Syntax:

```
NullCount({[SetExpression][DISTINCT] [TOTAL [<fld {,fld}>]]} expr)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.

Argument	Description
set_ expression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Examples and results

Example	Result
NullCount ([OrderNumber])	1 because we have introduced a null value using NullInterpret in the inline **LOAD** statement.

Data used in example:

```
Set NULLINTERPRET = NULL;
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|CustomerID
Astrida|AA|1|10|1
Astrida|AA|7|18|1
Astrida|BB|4|9|1
Astrida|CC|6|2|1
Betacab|AA|5|4|2
Betacab|BB|2|5|2
Betacab|DD|||
Canutility|AA|3|8|
Canutility|CC|NULL||
] (delimiter is '|');
Set NULLINTERPRET=;
```

NumericCount

NumericCount() returns the number of numeric values found in the expression, as defined by a **group by** clause.

Syntax:

```
NumericCount ( [ distinct ] expr)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
expr Expression	The expression or field containing the data to be measured.
distinct	If the word **distinct** occurs before the expression, all duplicates are disregarded.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

Scripting example

Example	Result
`LOAD NumericCount(OrderNumber) as TotalNumericCount Resident Temp;`	The second statement gives: TotalNumericCount 7 in a table with that dimension.
Given that the **Temp** table is loaded as in the previous example: `LOAD NumericCount(distinct OrderNumber) as TotalNumericCountDistinct Resident Temp;`	TotalNumericCountDistinct 6 Because there is one OrderNumber that duplicates another, so the result is 6 that are not duplicates..

Example:

```
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|UnitPrice
Astrida|AA|1|4|16
Astrida|AA|7|10|15
Astrida|BB|4|9|9
Betacab|CC|6|5|10
Betacab|AA|5|2|20
Betacab|BB||| 25
Canutility|AA|||15
Canutility|CC| ||19
Divadip|CC|2|4|16
Divadip|DD|7|1|25
] (delimiter is '|');
NumCount1:
LOAD Customer,NumericCount(OrderNumber) as NumericCountByCustomer Resident Temp Group By
Customer;
```

Resulting table

Customer	NumericCountByCustomer
Astrida	3
Betacab	2
Canutility	0
Divadip	2

NumericCount - chart function

NumericCount() aggregates the number of numeric values in each chart dimension.

Syntax:

```
NumericCount({[SetExpression] [DISTINCT] [TOTAL [<fld {,fld}>]]} expr)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
set_ expression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Data

Customer	Product	OrderNumber	UnitSales	Unit Price
Astrida	AA	1	4	16
Astrida	AA	7	10	15
Astrida	BB	4	9	1

Customer	Product	OrderNumber	UnitSales	Unit Price
Betacab	BB	6	5	10
Betacab	CC	5	2	20
Betacab	DD			25
Canutility	AA			15
Canutility	CC			19
Divadip	AA	2	4	16
Divadip	DD	3		25

The following examples assume that all customers are selected, except where stated.

Examples and results

Example	Result
`NumericCount ([OrderNumber])`	7 because three of the 10 fields in OrderNumber are empty. *"0" counts as a value and not an empty cell. However, if a measure aggregates to 0 for a dimension, that dimension will not be included in charts.*
`NumericCount ([Product])`	0 because all product names are in text. Typically you could use this to check that no text fields have been given numeric content.
`NumericCount (DISTINCT [OrderNumber])/Count (DISTINCT [OrderNumber)]`	Counts all the number of distinct numeric order numbers and divides it by the number of order numbers numeric and non-numeric. This will be 1 if all field values are numeric. Typically you could use this to check that all field values are numeric. In the example, there are 7 distinct numeric values for OrderNumber of 8 distinct numeric and non-numerid, so the expression returns 0.875.

Data used in example:

```
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|UnitPrice
Astrida|AA|1|4|16
Astrida|AA|7|10|15
Astrida|BB|4|9|9
Betacab|CC|6|5|10
Betacab|AA|5|2|20
Betacab|BB||| 25
Canutility|AA|||15
Canutility|CC| ||19
Divadip|CC|2|4|16
Divadip|DD|3|1|25
```

```
] (delimiter is '|');
```

TextCount

TextCount() returns the number of field values that are non-numeric aggregated in the expression, as defined by a **group by** clause.

Syntax:

```
TextCount ( [ distinct ] expr)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
expr Expression	The expression or field containing the data to be measured.
distinct	If the word **distinct** occurs before the expression, all duplicates are disregarded.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

Example:

```
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|UnitPrice
Astrida|AA|1|4|16
Astrida|AA|7|10|15
Astrida|BB|4|9|9
Betacab|CC|6|5|10
Betacab|AA|5|2|20
Betacab|BB||| 25
Canutility|AA|||15
Canutility|CC| ||19
Divadip|CC|2|4|16
Divadip|DD|3|1|25
] (delimiter is '|');
TextCount1:
LOAD Customer,TextCount(Product) as ProductTextCount Resident Temp Group By Customer;
```

Resulting table

Customer	ProductTextCount
Astrida	3
Betacab	3
Canutility	2
Divadip	2

Example:

```
LOAD Customer,TextCount(OrderNumber) as OrderNumberTextCount Resident Temp Group By Customer;
```

Resulting table

Customer	OrderNumberTextCount
Astrida	0
Betacab	1
Canutility	2
Divadip	0

TextCount - chart function

TextCount() is used to aggregate the number of field values that are non-numeric in each chart dimension.

Syntax:

```
TextCount({[SetExpression] [DISTINCT] [TOTAL [<fld {,fld}>]]} expr)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.

Argument	Description
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Data

Customer	Product	OrderNumber	UnitSales	Unit Price
Astrida	AA	1	4	16
Astrida	AA	7	10	15
Astrida	BB	4	9	1
Betacab	BB	6	5	10
Betacab	CC	5	2	20
Betacab	DD			25
Canutility	AA			15
Canutility	CC			19
Divadip	AA	2	4	16
Divadip	DD	3		25

Examples and results

Example	Result
`TextCount ([Product])`	10 because all of the 10 fields in Product are text. *"0" counts as a value and not an empty cell. However, if a measure aggregates to 0 for a dimension, that dimension will not be included in charts. Empty cells are evaluated as being non text and are not counted by TextCount.*
`TextCount ([OrderNumber])`	3, because empty cells are counted. Typically, you would use this to check that no numeric fields have been given text values or are non-zero.

Example	Result
`TextCount (DISTINCT [Product])/Count ([Product)]`	Counts all the number of distinct text values of Product (4), and divides it by the total number of values in Product (10). The result is 0.4.

Data used in example:

```
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|UnitPrice
Astrida|AA|1|4|16
Astrida|AA|7|1|15
Astrida|BB|4|9|9
Betacab|CC|6|5|10
Betacab|AA|5|2|20
Betacab|BB|||| 25
Canutility|AA|||15
Canutility|CC|||19
Divadip|CC|2|4|16
Divadip|DD|3|1|25
] (delimiter is '|');
```

Financial aggregation functions

This section describes aggregation functions for financial operations regarding payments and cash flow.

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Financial aggregation functions in the data load script

IRR

IRR() returns the aggregated internal rate of return for a series of cash flows represented by the numbers in the expression iterated over a number of records as defined by a group by clause.

```
IRR (expression)
```

XIRR

XIRR() returns the aggregated internal rate of return for a schedule of cash flows (that is not necessarily periodic) represented by paired numbers in **pmt** and **date** iterated over a number of records as defined by a group by clause. All payments are discounted based on a 365-day year.

```
XIRR (valueexpression, dateexpression )
```

NPV

NPV() returns the aggregated net present value of an investment based on a **discount_rate** per period and a series of future payments (negative values) and incomes (positive values), represented by the numbers in **value**, iterated over a number of records, as defined by a group by clause. The payments and incomes are assumed to occur at the end of each period.

```
NPV (rate, expression)
```

XNPV

XNPV() returns the aggregated net present value for a schedule of cashflows (not necessarily periodic) represented by paired numbers in **pmt** and **date**, iterated over a number of records as defined by a group by clause. Rate is the interest rate per period. All payments are discounted based on a 365-day year.

```
XNPV (rate, valueexpression, dateexpression)
```

Financial aggregation functions in chart expressions

These financial aggregation functions can be used in charts.

IRR

IRR() returns the aggregated internal rate of return for a series of cash flows represented by the numbers in the expression given by **value** iterated over the chart dimensions.

```
 IRR - chart function[TOTAL [<fld {,fld}>]] value)
```

NPV

NPV() returns the aggregated net present value of an investment based on a **discount_rate** per period and a series of future payments (negative values) and incomes (positive values,) represented by the numbers in **value**, iterated over the chart dimensions. The payments and incomes are assumed to occur at the end of each period.

```
NPV - chart function([TOTAL [<fld {,fld}>]] discount_rate, value)
```

XIRR

XIRR()returns the aggregated internal rate of return for a schedule of cash flows (that is not necessarily periodic) represented by paired numbers in the expressions given by **pmt** and **date** iterated over the chart dimensions. All payments are discounted based on a 365-day year.

```
XIRR - chart function (page 313)([TOTAL [<fld {,fld}>]] pmt, date)
```

XNPV

XNPV() returns the aggregated net present value for a schedule of cash flows (not necessarily periodic) represented by paired numbers in the expressions given by **pmt** and **date** iterated over the chart dimensions. All payments are discounted based on a 365-day year.

```
XNPV - chart function([TOTAL [<fld{,fld}>]] discount_rate, pmt, date)
```

IRR

IRR() returns the aggregated internal rate of return for a series of cash flows represented by the numbers in the expression iterated over a number of records as defined by a group by clause.

These cash flows do not have to be even, as they would be for an annuity. However, the cash flows must occur at regular intervals, such as monthly or annually. The internal rate of return is the interest rate received for an investment consisting of payments (negative values) and income (positive values) that occur at regular periods.The function needs at least one positive and one negative value to calculate.

Syntax:

```
IRR(value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The expression or field containing the data to be measured.

Limitations:

Text values, NULL values and missing values are disregarded.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Examples and results:

Examples and results

Example	Year	IRR2013

`Cashflow:`
`LOAD 2013 as Year, * inline [`
`Date|Discount|Payments`
`2013-01-01|0.1|-10000`
`2013-03-01|0.1|3000`
`2013-10-30|0.1|4200`
`2014-02-01|0.2|6800`
`] (delimiter is '|');`
 	
`Cashflow1:`
`LOAD Year,IRR(Payments) as IRR2013 Resident Cashflow Group By Year;`
	2013	0.1634

IRR - chart function

IRR() returns the aggregated internal rate of return for a series of cash flows represented by the numbers in the expression given by **value** iterated over the chart dimensions.

These cash flows do not have to be even, as they would be for an annuity. However, the cash flows must occur at regular intervals, such as monthly or annually. The internal rate of return is the interest rate received for an investment consisting of payments (negative values) and income (positive values) that occur at regular periods. The function needs at least one positive and one negative value to calculate.

Syntax:

```
IRR([TOTAL [<fld {,fld}>]] value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The expression or field containing the data to be measured.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values are disregarded.

Examples and results:

Examples and results

Example	Result
`IRR (Payments)`	0.1634 The payments are assumed to be periodic in nature, for example monthly. *The Date field is used in the XIRR example where payments can be non-periodical as long as you provide the dates on which payments were made.*

Data used in examples:

```
Cashflow:
LOAD 2013 as Year, * inline [
Date|Discount|Payments
2013-01-01|0.1|-10000
2013-03-01|0.1|3000
2013-10-30|0.1|4200
2014-02-01|0.2|6800
] (delimiter is '|');
```

See also:

XIRR - chart function (page 313)
Aggr - chart function (page 459)

NPV

NPV() returns the aggregated net present value of an investment based on a **discount_rate** per period and a series of future payments (negative values) and incomes (positive values), represented by the numbers in **value**, iterated over a number of records, as defined by a group by clause. The payments and incomes are assumed to occur at the end of each period.

Syntax:

```
NPV(discount_rate, value)
```

Return data type: numeric. The result has a default number format of money.

Arguments:

Arguments

Argument	Description
discount_rate	**discount_rate** is the rate of discount over the length of the period.
value	The expression or field containing the data to be measured.

Text values, NULL values and missing values are disregarded.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Examples and results

Example	Year	NPV1_2013

`Cashflow:`
`LOAD 2013 as Year, * inline [`
`Date|Discount|Payments`
`2013-01-01|0.1|-10000`
`2013-03-01|0.1|3000`
`2013-10-30|0.1|4200`
`2014-02-01|0.2|6800`
`] (delimiter is '|');`
 	
`Cashflow1:`
`LOAD Year,NPV(0.2, Payments) as NPV1_2013 Resident Cashflow Group By Year;`
	2013	-$540.12

Examples and results

Example	Year	Discount	NPV2_ 2013

Given that the **Cashflow** table is loaded as in the previous example:
`LOAD Year,NPV(Discount, Payments) as NPV2_2013 Resident Cashflow Group By Year, Discount;`
Note that the Group By clause sorts the results by Year and Discount. The first argument, discount_rate, is given as a field (Discount), rather than a specific number, and therefore, a second sorting criterion is required. A field can contain a different values, so the aggregated records must be sorted to allow for different values of Year and Discount.
`;`
	2013
	2013
		0.1
		0.2
			-$3456.05
			$5666.67

NPV - chart function

NPV() returns the aggregated net present value of an investment based on a **discount_rate** per period and a series of future payments (negative values) and incomes (positive values,) represented by the numbers in **value**, iterated over the chart dimensions. The payments and incomes are assumed to occur at the end of each period.

Syntax:

```
NPV([TOTAL [<fld {,fld}>]] discount_rate, value)
```

Return data type: numeric The result has a default number format of money.

Arguments:

Arguments

Argument	Description
discount_ rate	**discount_rate** is the rate of discount over the length of the period.**discount_rate** is the percentage rate of discount applied.
value	The expression or field containing the data to be measured.

Argument	Description
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values. The **TOTAL** qualifier may be followed by a list of one or more field names within angle brackets. These field names should be a subset of the chart dimension variables. In this case, the calculation is made disregarding all chart dimension variables except those listed, that is, one value is returned for each combination of field values in the listed dimension fields. Also, fields that are not currently a dimension in a chart may be included in the list. This may be useful in the case of group dimensions, where the dimension fields are not fixed. Listing all of the variables in the group causes the function to work when the drill-down level changes.

Limitations:

discount_rate and **value** must not contain aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values are disregarded.

Examples and results:

Examples and results

Example	Result
`NPV(Discount, Payments)`	-$540.12

Data used in examples:

```
Cashflow:
LOAD 2013 as Year, * inline [
Date|Discount|Payments
2013-01-01|0.1|-10000
2013-03-01|0.1|3000
2013-10-30|0.1|4200
2014-02-01|0.2|6800
] (delimiter is '|');
```

See also:

XNPV - chart function (page 315)
Aggr - chart function (page 459)

XIRR

XIRR() returns the aggregated internal rate of return for a schedule of cash flows (that is not necessarily periodic) represented by paired numbers in **pmt** and **date** iterated over a number of records as defined by a group by clause. All payments are discounted based on a 365-day year.

Syntax:

```
XIRR(pmt, date )
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
pmt	Payments.The expression or field containing the cash flows corresponding to the payment schedule given in **date**.
date	The expression or field containing the schedule of dates corresponding to the cash flow payments given in **pmt**.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair will result in the entire data-pair to be disregarded.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Examples and results

Example	Year	XIRR2013
Cashflow: LOAD 2013 as Year, * inline [Date|Discount|Payments 2013-01-01|0.1|-10000 2013-03-01|0.1|3000 2013-10-30|0.1|4200 2014-02-01|0.2|6800] (delimiter is '|'); Cashflow1: LOAD Year,XIRR(Payments, Date) as XIRR2013 Resident Cashflow Group By Year;	2013	0.5385

XIRR - chart function

XIRR()returns the aggregated internal rate of return for a schedule of cash flows (that is not necessarily periodic) represented by paired numbers in the expressions given by **pmt** and **date** iterated over the chart dimensions. All payments are discounted based on a 365-day year.

Syntax:

```
XIRR([TOTAL [<fld {,fld}>]] pmt, date)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
pmt	Payments.The expression or field containing the cash flows corresponding to the payment schedule given in **date**.
date	The expression or field containing the schedule of dates corresponding to the cash flow payments given in **pmt**.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

pmt and **date** must not contain aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

Examples and results:

Examples and results

Example	Result
`XIRR(Payments, Date)`	0.5385

Data used in examples:

```
Cashflow:
LOAD 2013 as Year, * inline [
Date|Discount|Payments
```

```
2013-01-01|0.1|-10000
2013-03-01|0.1|3000
2013-10-30|0.1|4200
2014-02-01|0.2|6800
] (delimiter is '|');
```

See also:

IRR - chart function (page 307)
Aggr - chart function (page 459)

XNPV

XNPV() returns the aggregated net present value for a schedule of cashflows (not necessarily periodic) represented by paired numbers in **pmt** and **date**, iterated over a number of records as defined by a group by clause. Rate is the interest rate per period. All payments are discounted based on a 365-day year.

Syntax:

```
XNPV(discount_rate, pmt, date)
```

Return data type: numeric. The result has a default number format of money.

Arguments:

Arguments

Argument	Description
discount_rate	**discount_rate** is the rate of discount over the length of the period.
value	The expression or field containing the data to be measured.
date	The expression or field containing the schedule of dates corresponding to the cash flow payments given in **pmt**.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair will result in the entire data-pair to be disregarded.

Examples:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Examples and results

Example	Year	XNPV1_2013

`Cashflow:`
`LOAD 2013 as Year, * inline [`
`Date|Discount|Payments`
`2013-01-01|0.1|-10000`
`2013-03-01|0.1|3000`
`2013-10-30|0.1|4200`
`2014-02-01|0.2|6800`
`] (delimiter is '|');`
 	
`Cashflow1:`
`LOAD Year,XNPV(0.2, Payments, Date) as XNPV1_2013 Resident Cashflow Group By Year;`
	2013	$2104.37

Examples and results

Example	Year	Discount	XNPV2_2013
Given that the **Cashflow** table is loaded as in the previous example: `LOAD Year,XNPV(Discount, Payments, Date) as XNPV2_2013 Resident Cashflow Group By Year, Discount;` Note that the Group By clause sorts the results by Year and Discount. The first argument, discount_rate, is given as a field (Discount), rather than a specific number, and therefore, a second sorting criterion is required. A field can contain a different values, so the aggregated records must be sorted to allow for different values of Year and Discount.	2013 2013	0.1 0.2	-$3164.35 $6800.00

XNPV - chart function

XNPV() returns the aggregated net present value for a schedule of cash flows (not necessarily periodic) represented by paired numbers in the expressions given by **pmt** and **date** iterated over the chart dimensions. All payments are discounted based on a 365-day year.

Syntax:

```
XNPV([TOTAL [<fld{,fld}>]] discount_rate, pmt, date)
```

Return data type: numeric The result has a default number format of money.

Arguments:

Arguments

Argument	Description
discount_rate	**discount_rate** is the rate of discount over the length of the period.**discount_rate** is the percentage rate of discount applied.

Argument	Description
pmt	Payments.The expression or field containing the cash flows corresponding to the payment schedule given in **date**.
date	The expression or field containing the schedule of dates corresponding to the cash flow payments given in **pmt**.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

discount_rate, **pmt** and **date** must not contain aggregation functions, unless these inner aggregations contain the **TOTAL** or **ALL** qualifiers. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

Examples and results:

Examples and results

Example	Result
`XNPV(Discount, Payments, Date)`	-$3164.35

Data used in examples:

```
Cashflow:
LOAD 2013 as Year, * inline [
Date|Discount|Payments
2013-01-01|0.1|-10000
2013-03-01|0.1|3000
2013-10-30|0.1|4200
2014-02-01|0.2|6800
] (delimiter is '|');
```

See also:

NPV - chart function (page 310)
Aggr - chart function (page 459)

Statistical aggregation functions

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Statistical aggregation functions in the data load script

The following statistical aggregation functions can be used in scripts.

Avg

Avg() finds the average value of the aggregated data in the expression over a number of records as defined by a **group by** clause.

```
Avg  ([distinct] expression)
```

Correl

Correl() returns the aggregated correlation coefficient for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
Correl (x-expression, y-expression)
```

Fractile

Fractile() finds the value that corresponds to the inclusive fractile (quantile) of the aggregated data in the expression over a number of records as defined by a **group by** clause.

```
Fractile (expression, fractile)
```

FractileExc

FractileExc() finds the value that corresponds to the exclusive fractile (quantile) of the aggregated data in the expression over a number of records as defined by a **group by** clause.

```
FractileExc (expression, fractile)
```

Kurtosis

Kurtosis() returns the kurtosis of the data in the expression over a number of records as defined by a **group by** clause.

```
Kurtosis  ([distinct ] expression )
```

LINEST_B

LINEST_B() returns the aggregated b value (y-intercept) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
LINEST_B (y-expression, x-expression [, y0 [, x0 ]])
```

LINEST_df

LINEST_DF() returns the aggregated degrees of freedom of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
LINEST_DF  (y-expression, x-expression [, y0 [, x0 ]])
```

LINEST_f

This script function returns the aggregated F statistic ($r^2/(1-r^2)$) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
LINEST_F (y-expression, x-expression [, y0 [, x0 ]])
```

LINEST_m

LINEST_M() returns the aggregated m value (slope) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
LINEST_M  (y-expression, x-expression [, y0 [, x0 ]])
```

LINEST_r2

LINEST_R2() returns the aggregated r^2 value (coefficient of determination) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
LINEST_R2 (y-expression, x-expression [, y0 [, x0 ]])
```

LINEST_seb

LINEST_SEB() returns the aggregated standard error of the b value of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
LINEST_SEB  (y-expression, x-expression [, y0 [, x0 ]])
```

LINEST_sem

LINEST_SEM() returns the aggregated standard error of the m value of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
LINEST_SEM  (y-expression, x-expression [, y0 [, x0 ]])
```

LINEST_sey

LINEST_SEY() returns the aggregated standard error of the y estimate of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
LINEST_SEY  (y-expression, x-expression [, y0 [, x0 ]])
```

LINEST_ssreg

LINEST_SSREG() returns the aggregated regression sum of squares of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
LINEST_SSREG  (y-expression, x-expression [, y0 [, x0 ]])
```

Linest_ssresid

LINEST_SSRESID() returns the aggregated residual sum of squares of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
LINEST_SSRESID  (y-expression, x-expression [, y0 [, x0 ]])
```

Median

Median() returns the aggregated median of the values in the expression over a number of records as defined by a **group by** clause.

```
Median (expression)
```

Skew

Skew() returns the skewness of expression over a number of records as defined by a **group by** clause.

```
Skew ([ distinct] expression)
```

Stdev

Stdev() returns the standard deviation of the values given by the expression over a number of records as defined by a **group by** clause.

```
Stdev ([distinct] expression)
```

Sterr

Sterr() returns the aggregated standard error (stdev/sqrt(n)) for a series of values represented by the expression iterated over a number of records as defined by a **group by** clause.

```
Sterr  ([distinct] expression)
```

STEYX

STEYX() returns the aggregated standard error of the predicted y-value for each x-value in the regression for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

```
STEYX  (y-expression, x-expression)
```

Statistical aggregation functions in chart expressions

The following statistical aggregation functions can be used in charts.

Avg

Avg() returns the aggregated average of the expression or field iterated over the chart dimensions.

```
Avg - chart function({[SetExpression] [DISTINCT] [TOTAL [<fld{, fld}>]]}
expr)
```

Correl

Correl() returns the aggregated correlation coefficient for two data sets. The correlation function is a measure of the relationship between the data sets and is aggregated for (x,y) value pairs iterated over the chart dimensions.

```
Correl - chart function({[SetExpression] [TOTAL [<fld {, fld}>]]} value1,
value2 )
```

Fractile

Fractile() finds the value that corresponds to the inclusive fractile (quantile) of the aggregated data in the range given by the expression iterated over the chart dimensions.

```
Fractile - chart function({[SetExpression] [TOTAL [<fld {, fld}>]]} expr,
fraction)
```

FractileExc

FractileExc() finds the value that corresponds to the exclusive fractile (quantile) of the aggregated data in the range given by the expression iterated over the chart dimensions.

```
FractileExc - chart function({[SetExpression] [TOTAL [<fld {, fld}>]]} expr,
fraction)
```

Kurtosis

Kurtosis() finds the kurtosis of the range of data aggregated in the expression or field iterated over the chart dimensions.

```
Kurtosis - chart function({[SetExpression] [DISTINCT] [TOTAL [<fld{, fld}>]]}
expr)
```

LINEST_b

LINEST_B() returns the aggregated b value (y-intercept) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in the expressions given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

```
LINEST_R2 - chart function({[SetExpression] [TOTAL [<fld{ ,fld}>]] }y_value,
x_value[, y0_const[, x0_const]])
```

LINEST_df

LINEST_DF() returns the aggregated degrees of freedom of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in the expressions given by **x_value** and **y_value**, iterated over the chart dimensions.

```
LINEST_DF - chart function({[SetExpression] [TOTAL [<fld{, fld}>]]} y_value,
x_value [, y0_const [, x0_const]])
```

LINEST_f

LINEST_F() returns the aggregated F statistic (r2/(1-r2)) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in the expressions given by **x_value** and the **y_value**, iterated over the chart dimensions.

```
LINEST_F - chart function({[SetExpression] [TOTAL[<fld{, fld}>]]} y_value, x_value [, y0_const [, x0_const]])
```

LINEST_m

LINEST_M() returns the aggregated m value (slope) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

```
LINEST_M - chart function({[SetExpression] [TOTAL[<fld{, fld}>]]} y_value, x_value [, y0_const [, x0_const]])
```

LINEST_r2

LINEST_R2() returns the aggregated r2 value (coefficient of determination) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

```
LINEST_R2 - chart function({[SetExpression] [TOTAL [<fld{ ,fld}>]] }y_value, x_value[, y0_const[, x0_const]])
```

LINEST_seb

LINEST_SEB() returns the aggregated standard error of the b value of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

```
LINEST_SEB - chart function({[SetExpression] [TOTAL [<fld{ ,fld}>]] }y_value, x_value[, y0_const[, x0_const]])
```

LINEST_sem

LINEST_SEM() returns the aggregated standard error of the m value of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

```
LINEST_SEM - chart function([{set_expression}][ distinct ] [total [<fld {,fld}>] ] y-expression, x-expression [, y0 [, x0 ]] )
```

LINEST_sey

LINEST_SEY() returns the aggregated standard error of the y estimate of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

```
LINEST_SEY - chart function({[SetExpression] [TOTAL [<fld{ ,fld}>]] }y_value, x_value[, y0_const[, x0_const]])
```

LINEST_ssreg

LINEST_SSREG() returns the aggregated regression sum of squares of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

```
LINEST_SSREG - chart function({[SetExpression] [TOTAL [<fld{ ,fld}>]] }y_
value, x_value[, y0_const[, x0_const]])
```

LINEST_ssresid

LINEST_SSRESID() returns the aggregated residual sum of squares of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in the expressions given by **x_value** and **y_value**, iterated over the chart dimensions.

```
LINEST_SSRESID - chart functionLINEST_SSRESID() returns the aggregated
residual sum of squares of a linear regression defined by the equation y=mx+b
for a series of coordinates represented by paired numbers in the expressions
given by x_value and y_value, iterated over the chart dimensions.  LINEST_
SSRESID([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value, x_value
[, y0_const[, x0_const]]) numeric ArgumentsArgumentDescriptiony_valueThe
expression or field containing the range of y-values to be measured.x_
valueThe expression or field containing the range of x-values to be
measured.y0, x0An optional value y0 may be stated forcing the regression line
to pass through the y-axis at a given point. By stating both y0 and x0 it is
possible to force the regression line to pass through a single fixed
coordinate. Unless both y0 and x0 are stated, the function requires at least
two valid data-pairs to calculate. If y0 and x0 are stated, a single data
pair will do. SetExpressionBy default, the aggregation function will
aggregate over the set of possible records defined by the selection. An
alternative set of records can be defined by a set analysis expression.
DISTINCTIf the word DISTINCT occurs before the function arguments, duplicates
resulting from the evaluation of the function arguments are disregarded.
TOTALIf the word TOTAL occurs before the function arguments, the calculation
is made over all possible values given the current selections, and not just
those that pertain to the current dimensional value, that is, it disregards
the chart dimensions. By using TOTAL [<fld {.fld}>], where the TOTAL
qualifier is followed by a list of one or more field names as a subset of the
chart dimension variables, you create a subset of the total possible
values.An optional value y0 may be stated forcing the regression line to pass
through the y-axis at a given point. By stating both y0 and x0 it is possible
to force the regression line to pass through a single fixed coordinate.  The
parameter of the aggregation function must not contain other aggregation
functions, unless these inner aggregations contain the TOTAL qualifier. For
more advanced nested aggregations, use the advanced function Aggr, in
combination with a specified dimension. Text values, NULL values and missing
values in any or both pieces of a data-pair result in the entire data-pair
```

```
being disregarded.  An example of how to use linest functionsavg
(({[SetExpression] [TOTAL [<fld{ ,fld}>]] }y_value, x_value[, y0_const[, x0_
const]])
```

Median

Median() returns the median value of the range of values aggregated in the expression iterated over the chart dimensions.

```
Median - chart function({[SetExpression] [TOTAL [<fld{, fld}>]]} expr)
```

MutualInfo

MutualInfo calculates the mutual information (MI) between two fields or between aggregated values in **Aggr ()**.

```
MutualInfo - chart function (page 362){[SetExpression] [DISTINCT] [TOTAL
target, driver [, datatype [, breakdownbyvalue [, samplesize ]]])
```

Skew

Skew() returns the aggregated skewness of the expression or field iterated over the chart dimensions.

```
Skew - chart function{[SetExpression] [DISTINCT] [TOTAL [<fld{ ,fld}>]]}
expr)
```

Stdev

Stdev() finds the standard deviation of the range of data aggregated in the expression or field iterated over the chart dimensions.

```
Stdev - chart function({[SetExpression] [DISTINCT] [TOTAL [<fld{, fld}>]]}
expr)
```

Sterr

Sterr() finds the value of the standard error of the mean, (stdev/sqrt(n)), for the series of values aggregated in the expression iterated over the chart dimensions.

```
Sterr - chart function({[SetExpression] [DISTINCT] [TOTAL[<fld{, fld}>]]}
expr)
```

STEYX

STEYX() returns the aggregated standard error when predicting y-values for each x-value in a linear regression given by a series of coordinates represented by paired numbers in the expressions given by **y_value** and **x_ value**.

```
STEYX - chart function{[SetExpression] [TOTAL [<fld{, fld}>]]} y_value, x_
value)
```

Avg

Avg() finds the average value of the aggregated data in the expression over a number of records as defined by a **group by** clause.

Syntax:

```
Avg([DISTINCT] expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
DISTINCT	If the word **distinct** occurs before the expression, all duplicates will be disregarded.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Resulting data

Example	Result

`Temp:`
`crosstable (Month, Sales) load * inline [`
`Customer|Jan|Feb|Mar||Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec`
`Astrida|46|60|70|13|78|20|45|65|78|12|78|22`
`Betacab|65|56|22|79|12|56|45|24|32|78|55|15`
`Canutility|77|68|34|91|24|68|57|36|44|90|67|27`
`Divadip|36|44|90|67|27|57|68|47|90|80|94`
`] (delimiter is '|');`
 	
`Avg1:`
`LOAD Customer, Avg(Sales) as MyAverageSalesByCustomer`
`Resident Temp Group By Customer;`
	`Customer`
	`MyAverageSalesByCustomer`
	`Astrida 48.916667`
	`Betacab 44.916667`
	`Canutility 56.916667`
	`Divadip 63.083333`
	This can be checked in the sheet by creating a table including the measure:
	`Sum(Sales)/12`

Given that the **Temp** table is loaded as in the previous example:
 	
`LOAD Customer,Avg(DISTINCT Sales) as MyAvgSalesDistinct`
`Resident Temp Group By Customer;`
	`Customer`
	`MyAverageSalesByCustomer`
	`Astrida 43.1`
	`Betacab 43.909091`
	`Canutility 55.909091`
	`Divadip 61`
	Only the distinct values are counted. Divide the total by the number of non-duplicate values.

Avg - chart function

Avg() returns the aggregated average of the expression or field iterated over the chart dimensions.

Syntax:

```
Avg([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Examples and results:

Example table

Customer	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Astrida	46	60	70	13	78	20	45	65	78	12	78	22
Betacab	65	56	22	79	12	56	45	24	32	78	55	15
Canutility	77	68	34	91	24	68	57	36	44	90	67	27
Divadip	57	36	44	90	67	27	57	68	47	90	80	94

Function examples

Example	Result
`Avg(Sales)`	For a table including the dimension `Customer` and the measure `Avg([Sales])`, if **Totals** are shown, the result is 2566.

Example	Result
`Avg([TOTAL (Sales))`	53.458333 for all values of `Customer`, because the TOTAL qualifier means that dimensions are disregarded.
`Avg(DISTINCT (Sales))`	51.862069 for the total, because using the Distinct qualifier means only unique values in `Sales` for each `Customer` are evaluated.

Data used in examples:

```
Monthnames:
LOAD *, Dual(MonthText,MonthNumber) as Month INLINE [
MonthText, MonthNumber
Jan, 1
Feb, 2
Mar, 3
Apr, 4
May, 5
Jun, 6
Jul, 7
Aug, 8
Sep, 9
Oct, 10
Nov, 11
Dec, 12
];

Sales2013:
Crosstable (MonthText, Sales) LOAD * inline [
Customer|Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec
Astrida|46|60|70|13|78|20|45|65|78|12|78|22
Betacab|65|56|22|79|12|56|45|24|32|78|55|15
Canutility|77|68|34|91|24|68|57|36|44|90|67|27
Divadip|57|36|44|90|67|27|57|68|47|90|80|94
] (delimiter is '|');
```

See also:

Aggr - chart function (page 459)

Correl

Correl() returns the aggregated correlation coefficient for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
Correl(value1, value2)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value1, value2	The expressions or fields containing the two sample sets for which the correlation coefficient is to be measured.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Resulting data

Example	Result

`Salary:`
`Load *, 1 as Grp;`
`LOAD * inline [`
`"Employee name"|Gender|Age|Salary`
`Aiden Charles|Male|20|25000`
`Brenda Davies|Male|25|32000`
`Charlotte Edberg|Female|45|56000`
`Daroush Ferrara|Male|31|29000`
`Eunice Goldblum|Female|31|32000`
`Freddy Halvorsen|Male|25|26000`
`Gauri Indu|Female|36|46000`
`Harry Jones|Male|38|40000`
`Ian Underwood|Male|40|45000`
`Jackie Kingsley|Female|23|28000`
`] (delimiter is '|');`
 	
`Correl1:`
`LOAD Grp,`
`Correl(Age,Salary) as Correl_`
`Salary Resident Salary Group By`
`Grp;`
	In a table with the dimension `Correl_Salary`, the result of the Correl() calculation in the data load script will be shown:
	0.9270611

Correl - chart function

Correl() returns the aggregated correlation coefficient for two data sets. The correlation function is a measure of the relationship between the data sets and is aggregated for (x,y) value pairs iterated over the chart dimensions.

Syntax:

```
Correl([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] value1, value2 )
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value1, value2	The expressions or fields containing the two sample sets for which the correlation coefficient is to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

Examples and results:

Function examples

Example	Result
`Correl(Age, Salary)`	For a table including the dimension `Employee name` and the measure `Correl(Age, Salary)`, the result is 0.9270611. The result is only displayed for the totals cell.

Example	Result
`Correl (TOTAL Age, Salary))`	0.927. This and the following results are shown to three decimal places for readability. If you create a filter pane with the dimension Gender, and make selections from it, you see the result 0.951 when Female is selected and 0.939 if Male is selected. This is because the selection excludes all results that do not belong to the other value of Gender.
`Correl({1} TOTAL Age, Salary))`	0.927. Independent of selections. This is because the set expression {1} disregards all selections and dimensions.
`Correl (TOTAL <Gender> Age, Salary))`	0.927 in the total cell, 0.939 for all values of Male, and 0.951 for all values of Female. This corresponds to the results from making the selections in a filter pane based on Gender.

Data used in examples:

```
Salary:
LOAD * inline [
"Employee name"|Gender|Age|Salary
Aiden Charles|Male|20|25000
Brenda Davies|Male|25|32000
Charlotte Edberg|Female|45|56000
Daroush Ferrara|Male|31|29000
Eunice Goldblum|Female|31|32000
Freddy Halvorsen|Male|25|26000
Gauri Indu|Female|36|46000
Harry Jones|Male|38|40000
Ian Underwood|Male|40|45000
Jackie Kingsley|Female|23|28000
] (delimiter is '|');
```

See also:

Aggr - chart function (page 459)
Avg - chart function (page 324)
RangeCorrel (page 906)

Fractile

Fractile() finds the value that corresponds to the inclusive fractile (quantile) of the aggregated data in the expression over a number of records as defined by a **group by** clause.

You can use FractileExc (page 333) to calculate the exclusive fractile.

Syntax:

```
Fractile(expr, fraction)
```

Return data type: numeric

The function returns the value corresponding to the rank as defined by `rank = fraction * (N-1) + 1` where `N` is the number of values in `expr`. If `rank` is a non-integer number, an interpolation is made between the two closest values.

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to use when calculating the fractile.
fraction	A number between 0 and 1 corresponding to the fractile (quantile expressed as a fraction) to be calculated.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Resulting data

Example	Result

`Table1:`
`crosstable LOAD recno() as ID, *`
`inline [`
`Observation|Comparison`
`35|2`
`40|27`
`12|38`
`15|31`
`21|1`
`14|19`
`46|1`
`10|34`
`28|3`
`48|1`
`16|2`
`30|3`
`32|2`
`48|1`
`31|2`
`22|1`
`12|3`
`39|29`
`19|37`
`25|2 ] (delimiter is '|');`
 	
`Fractile1:`
`LOAD Type,`
`Fractile(Value,0.75) as MyFractile`
`Resident Table1 Group By Type;`
	In a table with the dimensions `Type` and `MyFractile`, the results of the Fractile() calculations in the data load script are:
	
	`Type MyFractile`
	`Comparison 27.5`
	`Observation 36`

Fractile - chart function

Fractile() finds the value that corresponds to the inclusive fractile (quantile) of the aggregated data in the range given by the expression iterated over the chart dimensions.

> *You can use FractileExc - chart function (page 334) to calculate the exclusive fractile.*

Syntax:

```
Fractile([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] expr, fraction)
```

Return data type: numeric

The function returns the value corresponding to the rank as defined by `rank = fraction * (N-1) + 1` where `N` is the number of values in `expr`. If `rank` is a non-integer number, an interpolation is made between the two closest values.

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to use when calculating the fractile.
fraction	A number between 0 and 1 corresponding to the fractile (quantile expressed as a fraction) to be calculated.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Examples and results:

Example table

Customer	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Astrida	46	60	70	13	78	20	45	65	78	12	78	22
Betacab	65	56	22	79	12	56	45	24	32	78	55	15
Canutility	77	68	34	91	24	68	57	36	44	90	67	27
Divadip	57	36	44	90	67	27	57	68	47	90	80	94

Function examples

Example	Result
`Fractile (Sales, 0.75)`	For a table including the dimension `Customer` and the measure `Fractile([Sales])`, if **Totals** are shown, the result is 71.75. This is the point in the distribution of values of `Sales` that 75% of the values fall beneath.
`Fractile (TOTAL Sales, 0.75))`	71.75 for all values of `Customer`, because the TOTAL qualifier means that dimensions are disregarded.
`Fractile (DISTINCT Sales, 0.75)`	70 for the total, because using the DISTINCT qualifier means only unique values in `Sales` for each `Customer` are evaluated.

Data used in examples:

```
Monthnames:
LOAD *, Dual(MonthText,MonthNumber) as Month INLINE [
MonthText, MonthNumber
Jan, 1
Feb, 2
Mar, 3
Apr, 4
May, 5
Jun, 6
Jul, 7
Aug, 8
Sep, 9
Oct, 10
Nov, 11
Dec, 12
];

Sales2013:
Crosstable (MonthText, Sales) LOAD * inline [
Customer|Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec
Astrida|46|60|70|13|78|20|45|65|78|12|78|22
Betacab|65|56|22|79|12|56|45|24|32|78|55|15
```

```
Canutility|77|68|34|91|24|68|57|36|44|90|67|27
Divadip|57|36|44|90|67|27|57|68|47|90|80|94
] (delimiter is '|');
```

See also:

Aggr - chart function (page 459)

FractileExc

FractileExc() finds the value that corresponds to the exclusive fractile (quantile) of the aggregated data in the expression over a number of records as defined by a **group by** clause.

You can use Fractile (page 329) to calculate the inclusive fractile.

Syntax:

```
FractileExc(expr, fraction)
```

Return data type: numeric

The function returns the value corresponding to the rank as defined by `rank = fraction * (N+1)` where `N` is the number of values in `expr`. If `rank` is a non-integer number, an interpolation is made between the two closest values.

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to use when calculating the fractile.
fraction	A number between 0 and 1 corresponding to the fractile (quantile expressed as a fraction) to be calculated.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Resulting data

Example	Result

`Table1:`
`crosstable LOAD recno() as ID, *`
`inline [`
`Observation|Comparison`
`35|2`
`40|27`
`12|38`
`15|31`
`21|1`
`14|19`
`46|1`
`10|34`
`28|3`
`48|1`
`16|2`
`30|3`
`32|2`
`48|1`
`31|2`
`22|1`
`12|3`
`39|29`
`19|37`
`25|2 ] (delimiter is '|');`
 	
`Fractile1:`
`LOAD Type,`
`FractileExc(Value,0.75) as`
`MyFractile Resident Table1 Group By`
`Type;`
	In a table with the dimensions `Type` and `MyFractile`, the results of the FractileExc() calculations in the data load script are:
	
	`Type MyFractile`
	`Comparison 28.5`
	`Observation 38`

FractileExc - chart function

FractileExc() finds the value that corresponds to the exclusive fractile (quantile) of the aggregated data in the range given by the expression iterated over the chart dimensions.

You can use Fractile - chart function (page 331) to calculate the inclusive fractile.

Syntax:

```
FractileExc([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] expr,
fraction)
```

Return data type: numeric

The function returns the value corresponding to the rank as defined by `rank = fraction * (N+1)` where `N` is the number of values in `expr`. If `rank` is a non-integer number, an interpolation is made between the two closest values.

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to use when calculating the fractile.
fraction	A number between 0 and 1 corresponding to the fractile (quantile expressed as a fraction) to be calculated.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Examples and results:

Example table

Customer	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Astrida	46	60	70	13	78	20	45	65	78	12	78	22
Betacab	65	56	22	79	12	56	45	24	32	78	55	15
Canutility	77	68	34	91	24	68	57	36	44	90	67	27
Divadip	57	36	44	90	67	27	57	68	47	90	80	94

Function examples

Example	Result
`FractileExc (Sales, 0.75)`	For a table including the dimension `Customer` and the measure `FractileExc([Sales])`, if **Totals** are shown, the result is 75.25. This is the point in the distribution of values of `Sales` that 75% of the values fall beneath.

Example	Result
`FractileExc (TOTAL Sales, 0.75))`	75.25 for all values of `Customer`, because the TOTAL qualifier means that dimensions are disregarded.
`FractileExc (DISTINCT Sales, 0.75)`	73.50 for the total, because using the DISTINCT qualifier means only unique values in `Sales` for each `Customer` are evaluated.

Data used in examples:

```
Monthnames:
LOAD *, Dual(MonthText,MonthNumber) as Month INLINE [
MonthText, MonthNumber
Jan, 1
Feb, 2
Mar, 3
Apr, 4
May, 5
Jun, 6
Jul, 7
Aug, 8
Sep, 9
Oct, 10
Nov, 11
Dec, 12
];

Sales2013:
Crosstable (MonthText, Sales) LOAD * inline [
Customer|Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec
Astrida|46|60|70|13|78|20|45|65|78|12|78|22
Betacab|65|56|22|79|12|56|45|24|32|78|55|15
Canutility|77|68|34|91|24|68|57|36|44|90|67|27
Divadip|57|36|44|90|67|27|57|68|47|90|80|94
] (delimiter is '|');
```

See also:

Aggr - chart function (page 459)

Kurtosis

Kurtosis() returns the kurtosis of the data in the expression over a number of records as defined by a **group by** clause.

Syntax:

```
Kurtosis([distinct ] expr )
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
distinct	If the word **distinct** occurs before the expression, all duplicates will be disregarded.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Resulting data

Example	Result

`Table1:`
`crosstable LOAD recno()`
`as ID, * inline [`
`Observation|Comparison`
`35|2`
`40|27`
`12|38`
`15|31`
`21|1`
`14|19`
`46|1`
`10|34`
`28|3`
`48|1`
`16|2`
`30|3`
`32|2`
`48|1`
`31|2`
`22|1`
`12|3`
`39|29`
`19|37`
`25|2 ] (delimiter is`
`'|');`
 	
`Kurtosis1:`
`LOAD Type,`
`Kurtosis(Value) as`
`MyKurtosis1,`
`Kurtosis(DISTINCT Value)`
`as MyKurtosis2`
`Resident Table1 Group By`
`Type;`
	In a table with the dimensions `Type`, `MyKurtosis1`,and `MyKurtosis2`, the results of the Kurtosis() calculations in the data load script are:
	
	`Type MyKurtosis1 MyKurtosis2`
	`Comparison -1.1612957 -1.4982366`
	`Observation -1.1148768 -0.93540144`

Kurtosis - chart function

Kurtosis() finds the kurtosis of the range of data aggregated in the expression or field iterated over the chart dimensions.

Syntax:

```
Kurtosis([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Examples and results:

Example table

Type	Value																			
Comparison	2	27	38	31	1	19	1	34	3	1	2	3	2	1	2	1	3	29	37	2
Observation	35	40	12	15	21	14	46	10	28	48	16	30	32	48	31	22	12	39	19	25

Function examples

Example	Result
`Kurtosis (Value)`	For a table including the dimension `Type` and the measure `Kurtosis(Value)`, if **Totals** are shown for the table, and number formatting is set to 3 significant figures, the result is 1.252. For `Comparison` it is 1.161 and for `Observation` it is 1.115.
`Kurtosis (TOTAL Value))`	1.252 for all values of `Type`, because the TOTAL qualifier means that dimensions are disregarded.

Data used in examples:

```
Table1:
crosstable LOAD recno() as ID, * inline [
Observation|Comparison
35|2
40|27
12|38
15|31
21|1
14|19
46|1
10|34
28|3
48|1
16|2
30|3
32|2
48|1
31|2
22|1
12|3
39|29
19|37
25|2 ] (delimiter is '|');
```

See also:

Avg - chart function (page 324)

LINEST_B

LINEST_B() returns the aggregated b value (y-intercept) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
LINEST_B (y_value, x_value[, y0 [, x0 ]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.
y(0), x(0)	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)

LINEST_B - chart function

LINEST_B() returns the aggregated b value (y-intercept) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in the expressions given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

Syntax:

```
LINEST_B([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value, x_value
[, y0_const [ , x0_const]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.

Argument	Description
y0_const, x0_const	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. *Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.*
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)
Avg - chart function (page 324)

LINEST_DF

LINEST_DF() returns the aggregated degrees of freedom of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
LINEST_DF (y_value, x_value[, y0 [, x0 ]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.
y(0), x(0)	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)

LINEST_DF - chart function

LINEST_DF() returns the aggregated degrees of freedom of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in the expressions given by **x_value** and **y_value**, iterated over the chart dimensions.

Syntax:

```
LINEST_DF([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value, x_value [, y0_const [, x0_const]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.

Argument	Description
y0, x0	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. *Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.*
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)
Avg - chart function (page 324)

LINEST_F

This script function returns the aggregated F statistic ($r^2/(1-r^2)$) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
LINEST_F (y_value, x_value[, y0 [, x0 ]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.
y(0), x(0)	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)

LINEST_F - chart function

LINEST_F() returns the aggregated F statistic (r2/(1-r2)) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in the expressions given by **x_value** and the **y_value**, iterated over the chart dimensions.

Syntax:

```
LINEST_F([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value, x_value
[, y0_const [, x0_const]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.

Argument	Description
y0, x0	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. *Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.*
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)
Avg - chart function (page 324)

LINEST_M

LINEST_M() returns the aggregated m value (slope) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
LINEST_M (y_value, x_value[, y0 [, x0 ]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.
y(0), x(0)	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)

LINEST_M - chart function

LINEST_M() returns the aggregated m value (slope) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

Syntax:

```
LINEST_M([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value, x_value
[, y0_const [, x0_const]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.

Argument	Description
y0, x0	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. *Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.*
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)
Avg - chart function (page 324)

LINEST_R2

LINEST_R2() returns the aggregated r^2 value (coefficient of determination) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
LINEST_R2 (y_value, x_value[, y0 [, x0 ]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.
y(0), x(0)	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)

LINEST_R2 - chart function

LINEST_R2() returns the aggregated r2 value (coefficient of determination) of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

Syntax:

```
LINEST_R2([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value, x_value[, y0_const[, x0_const]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.

Argument	Description
y0, x0	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. *Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.*
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)
Avg - chart function (page 324)

LINEST_SEB

LINEST_SEB() returns the aggregated standard error of the b value of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
LINEST_SEB (y_value, x_value[, y0 [, x0 ]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.
y(0), x(0)	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)

LINEST_SEB - chart function

LINEST_SEB() returns the aggregated standard error of the b value of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

Syntax:

```
LINEST_SEB([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value, x_
value[, y0_const[, x0_const]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.

Argument	Description
y0, x0	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. *Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.*
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)
Avg - chart function (page 324)

LINEST_SEM

LINEST_SEM() returns the aggregated standard error of the m value of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
LINEST_SEM (y_value, x_value[, y0 [, x0 ]])
```

Return data type: numeric

Arguments:

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.
y(0), x(0)	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)

LINEST_SEM - chart function

LINEST_SEM() returns the aggregated standard error of the m value of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

Syntax:

```
LINEST_SEM([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value, x_
value[, y0_const[, x0_const]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.

Argument	Description
y0, x0	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. *Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.*
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)
Avg - chart function (page 324)

LINEST_SEY

LINEST_SEY() returns the aggregated standard error of the y estimate of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
LINEST_SEY (y_value, x_value[, y0 [, x0 ]])
```

Return data type: numeric

Arguments:

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.
y(0), x(0)	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)

LINEST_SEY - chart function

LINEST_SEY() returns the aggregated standard error of the y estimate of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

Syntax:

```
LINEST_SEY([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value, x_
value[, y0_const[, x0_const]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.

Argument	Description
y0, x0	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. *Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.*
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)
Avg - chart function (page 324)

LINEST_SSREG

LINEST_SSREG() returns the aggregated regression sum of squares of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
LINEST_SSREG (y_value, x_value[, y0 [, x0 ]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.
y(0), x(0)	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)

LINEST_SSREG - chart function

LINEST_SSREG() returns the aggregated regression sum of squares of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers given by the expressions **x_value** and **y_value**, iterated over the chart dimensions.

Syntax:

```
LINEST_SSREG([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value, x_
value[, y0_const[, x0_const]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.

Argument	Description
y0, x0	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. *Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.*
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)
Avg - chart function (page 324)

LINEST_SSRESID

LINEST_SSRESID() returns the aggregated residual sum of squares of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
LINEST_SSRESID (y_value, x_value[, y0 [, x0 ]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.
y(0), x(0)	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)

LINEST_SSRESID - chart function

LINEST_SSRESID() returns the aggregated residual sum of squares of a linear regression defined by the equation y=mx+b for a series of coordinates represented by paired numbers in the expressions given by **x_value** and **y_value**, iterated over the chart dimensions.

Syntax:

```
LINEST_SSRESID([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value,
x_value[, y0_const[, x0_const]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.

Argument	Description
y0, x0	An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate. *Unless both y0 and x0 are stated, the function requires at least two valid data-pairs to calculate. If y0 and x0 are stated, a single data pair will do.*
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

An optional value y0 may be stated forcing the regression line to pass through the y-axis at a given point. By stating both y0 and x0 it is possible to force the regression line to pass through a single fixed coordinate.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

See also:

Examples of how to use linest functions (page 378)
Avg - chart function (page 324)

Median

Median() returns the aggregated median of the values in the expression over a number of records as defined by a **group by** clause.

Syntax:

```
Median (expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.

Example: Script expression using Median

Example - script expression

Load script

Load the following inline data and script expression in the data load editor for this example.

```
Table 1:
Load RecNo() as RowNo, Letter, Number Inline
[Letter, Number
A,1
A,3
A,4
A,9
B,2
B,8
B,9];

Median:
LOAD Letter,
Median(Number) as MyMedian
Resident Table1 Group By Letter;
```

Create a visualization

Create a table visualization in a Qlik Sense sheet with **Letter** and **MyMedian** as dimensions.

Result

Letter	MyMedian
A	3.5
B	8

Explanation

The median is considered the "middle" number when the numbers have been sorted in order from smallest to greatest. If the data set has an even number of values, the function returns the average of the two middle values. In this example, the median is calculated for each set of values of **A** and **B**, which is 3.5 and 8, respectively.

Median - chart function

Median() returns the median value of the range of values aggregated in the expression iterated over the chart dimensions.

Syntax:

```
Median([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Example: Chart expression using Median

Example - chart expression

Load script

Load the following data as an inline load in the data load editor to create the chart expression example below.

```
Load RecNo() as RowNo, Letter, Number Inline
[Letter, Number
A,1
A,3
A,4
A,9
B,2
```

```
B,8
B,9];
```

Create a visualization

Create a table visualization in a Qlik Sense sheet with **Letter** as a dimension.

Chart expression

Add the following expression to the table, as a measure:

```
Median(Number)
```

Result

Letter	Median(Number)
Totals	4
A	3.5
B	8

Explanation

The median is considered the "middle" number when the numbers have been sorted in order from smallest to greatest. If the data set has an even number of values, the function returns the average of the two middle values. In this example, the median is calculated for each set of values of **A** and **B**, which is 3.5 and 8, respectively.

The median for **Totals** is calculated from all values, which equals 4.

See also:

Avg - chart function (page 324)

MutualInfo - chart function

MutualInfo calculates the mutual information (MI) between two fields or between aggregated values in **Aggr()**.

MutualInfo returns the aggregated mutual information for two datasets. This allows key driver analysis between a field and a potential driver. Mutual information measures the relationship between the datasets and is aggregated for (x,y) pair values iterated over the chart dimensions. Mutual information is measured between 0 and 1 and can be formatted as a percentile value. **MutualInfo** is defined by either selections or by a set expression.

MutualInfo allows different kinds of MI analysis:

- Pair-wise MI: Calculate the MI between a driver field and a target field.
- Driver breakdown by value: The MI is calculated between individual field values in the driver and target fields.
- Feature selection: Use **MutualInfo** in a grid chart to create a matrix where all fields are compared to each other based on MI.

MutualInfo does not necessarily indicate causality between fields sharing mutual information. Two fields may share mutual information, but may not be equal drivers for each other. For example, when comparing ice cream sales and outdoor temperature, **MutualInfo** will show mutual information between the two. It will not indicate if it is outdoor temperature driving ice cream sales, which is likely, or if it is ice cream sales that drives outdoor temperature, which is unlikely.

When calculating mutual information, associations affect the correspondence between and the frequency of values from fields that are from different tables.

Returned values for the same fields or selections may vary slightly. This is due to each **MutualInfo** call operating on a randomly selected sample and the inherent randomness of the **MutualInfo** algorithm.

MutualInfo can be applied to the **Aggr()** function.

Syntax:

```
MutualInfo({SetExpression}] [DISTINCT] [TOTAL] field1, field2 , datatype [,
breakdownbyvalue [, samplesize ]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
field1, field2	The expressions or fields containing the two sample sets for which the mutual information to be measured.
datatype	The data types contained in the target and driver, 1 or 'dd' for discrete:discrete 2 or 'cc' for continuous:continuous 3 or 'cd' for continuous:discrete 4 or 'dc' for discrete:continuous Data types are not case sensitive.

Argument	Description
breakdownbyvalue	A static value corresponding to a value in the driver. If supplied, the calculation will calculate the MI contribution for that value. You can use **ValueList()** or **ValueLoop ()**. If **Null()** is added, the calculation will calculate the overall MI for all values in the driver. Breaking down by value requires the driver contain discrete data.
samplesize	The number of values to sample from the target and driver. Sampling is random. **MutualInfo** requires a minimum sample size of 80. By default, **MutualInfo** only samples up to 10,000 data-pairs as **MutualInfo** can be resource intensive. You can specify greater numbers of data-pairs in the sample size. If **MutualInfo** times out, reduce the sample size.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Function examples

Example	Result
`mutualinfo(Age, Salary, 1)`	For a table including the dimension `Employee name` and the measure `mutualinfo(Age, Salary, 1)`, the result is 0.99820986. The result is only displayed for the totals cell.
`mutualinfo (TOTAL Age, Salary, 1, null (), 81)`	If you create a filter pane with the dimension Gender, and make selections from it, you see the result 0.99805677 when Female is selected and 0.99847373 if Male is selected. This is because the selection excludes all results that do not belong to the other value of Gender.

Example	Result
`mutualinfo (TOTAL Age, Gender, 1, ValueLoop (25,35))`	0.68196996. Selecting any value from Gender will change this to 0.
`mutualinfo({1} TOTAL Age, Salary, 1, null ())`	0.99820986. This is independent of selections. The set expression {1} disregards all selections and dimensions.

Data used in examples:

```
Salary:
LOAD * inline [
"Employee name"|Age|Gender|Salary
Aiden Charles|20|Male|25000
Ann Lindquist|69|Female|58000
Anna Johansen|37|Female|36000
Anna Karlsson|42|Female|23000
Antonio Garcia|20|Male|61000
Benjamin Smith|42|Male|27000
Bill Yang|49|Male|50000
Binh Protzmann|69|Male|21000
Bob Park|51|Male|54000
Brenda Davies|25|Male|32000
Celine Gagnon|48|Female|38000
Cezar Sandu|50|Male|46000
Charles Ingvar Jönsson|27|Male|58000
Charlotte Edberg|45|Female|56000
Cindy Lynn|69|Female|28000
Clark Wayne|63|Male|31000
Daroush Ferrara|31|Male|29000
David Cooper|37|Male|64000
David Leg|58|Male|57000
Eunice Goldblum|31|Female|32000
Freddy Halvorsen|25|Male|26000
Gauri Indu|36|Female|46000
George van Zaant|59|Male|47000
Glenn Brown|58|Male|40000
Harry Jones|38|Male|40000
Helen Brolin|52|Female|66000
Hiroshi Ito|24|Male|42000
Ian Underwood|40|Male|45000
Ingrid Hendrix|63|Female|27000
Ira Baumel|39|Female|39000
Jackie Kingsley|23|Female|28000
Jennica Williams|36|Female|48000
Jerry Tessel|31|Male|57000
Jim Bond|50|Male|58000
Joan Callins|60|Female|65000
Joan Cleaves|25|Female|61000
Joe Cheng|61|Male|41000
John Doe|36|Male|59000
John Lemon|43|Male|21000
Karen Helmkey|54|Female|25000
```

```
Karl Berger|38|Male|68000
Karl Straubaum|30|Male|40000
Kaya Alpan|32|Female|60000
Kenneth Finley|21|Male|25000
Leif Shine|63|Male|70000
Lennart Skoglund|63|Male|24000
Leona Korhonen|46|Female|50000
Lina André|50|Female|65000
Louis Presley|29|Male|36000
Luke Langston|50|Male|63000
Marcus Salvatori|31|Male|46000
Marie Simon|57|Female|23000
Mario Rossi|39|Male|62000
Markus Danzig|26|Male|48000
Michael Carlen|21|Male|45000
Michelle Tyson|44|Female|69000
Mike Ashkenaz|45|Male|68000
Miro Ito|40|Male|39000
Nina Mihn|62|Female|57000
Olivia Nguyen|35|Female|51000
Olivier Simenon|44|Male|31000
Östen Ärlig|68|Male|57000
Pamala Garcia|69|Female|29000
Paolo Romano|34|Male|45000
Pat Taylor|67|Female|69000
Paul Dupont|34|Male|38000
Peter Smith|56|Male|53000
Pierre Clouseau|21|Male|37000
Preben Jørgensen|35|Male|38000
Rey Jones|65|Female|20000
Ricardo Gucci|55|Male|65000
Richard Ranieri|30|Male|64000
Rob Carsson|46|Male|54000
Rolf Wesenlund|25|Male|51000
Ronaldo Costa|64|Male|39000
Sabrina Richards|57|Female|40000
Sato Hiromu|35|Male|21000
Sehoon Daw|57|Male|24000
Stefan Lind|67|Male|35000
Steve Cioazzi|58|Male|23000
Sunil Gupta|45|Male|40000
Sven Svensson|45|Male|55000
Tom Lindwall|46|Male|24000
Tomas Nilsson|27|Male|22000
Trinity Rizzo|52|Female|48000
Vanessa Lambert|54|Female|27000
] (delimiter is '|');
```

Skew

Skew() returns the skewness of expression over a number of records as defined by a **group by** clause.

Syntax:

```
Skew([ distinct] expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
DISTINCT	If the word **distinct** occurs before the expression, all duplicates will be disregarded.

Examples and results:

Add the example script to your app and run it. Then build a straight table with `Type` and *MySkew* as dimensions.

Resulting data

Example	Result

`Table1:`
`crosstable LOAD recno() as ID, * inline [`
`Observation|Comparison`
`35|2`
`40|27`
`12|38`
`15|31`
`21|1`
`14|19`
`46|1`
`10|34`
`28|3`
`48|1`
`16|2`
`30|3`
`32|2`
`48|1`
`31|2`
`22|1`
`12|3`
`39|29`
`19|37`
`25|2 ] (delimiter is '|');`
 	
`Skew1:`
`LOAD Type,`
`Skew(Value) as MySkew`
`Resident Table1 Group By Type;`
	The results of the Skew() calculation are:
	• `Type` is `MySkew`
	• `Comparison` is 0.86414768
	• `Observation` is 0.32625351

Skew - chart function

Skew() returns the aggregated skewness of the expression or field iterated over the chart dimensions.

Syntax:

```
Skew([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Examples and results:

Add the example script to your app and run it. Then build a straight table with `Type` as dimension and `Skew(Value)` as measure.

`Totals` should be enabled in the properties of the table.

Example	Result

`Table1:`
`crosstable LOAD recno()`
`as ID, * inline [`
`Observation|Comparison`
`35|2`
`40|27`
`12|38`
`15|31`
`21|1`
`14|19`
`46|1`
`10|34`
`28|3`
`48|1`
`16|2`
`30|3`
`32|2`
`48|1`
`31|2`
`22|1`
`12|3`
`39|29`
`19|37`
`25|2 ] (delimiter is`
`'|');`
	The results of the Skew(Value) calculation are:
	• `Total` is 0.23522195
	• `Comparison` is 0.86414768
	• `Observation` is 0.32625351

See also:

Avg - chart function (page 324)

Stdev

Stdev() returns the standard deviation of the values given by the expression over a number of records as defined by a **group by** clause.

Syntax:

```
Stdev([distinct] expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
distinct	If the word **distinct** occurs before the expression, all duplicates will be disregarded.

Examples and results:

Add the example script to your app and run it. Then build a straight table with `Type` and `MyStdev` as dimensions.

Resulting data

Example	Result

`Table1:`
`crosstable LOAD recno() as ID, * inline [`
`Observation|Comparison`
`35|2`
`40|27`
`12|38`
`15|31`
`21|1`
`14|19`
`46|1`
`10|34`
`28|3`
`48|1`
`16|2`
`30|3`
`32|2`
`48|1`
`31|2`
`22|1`
`12|3`
`39|29`
`19|37`
`25|2 ] (delimiter is '|');`
 	
`Stdev1:`
`LOAD Type,`
`Stdev(Value) as MyStdev`
`Resident Table1 Group By Type;`
	The results of the Stdev() calculation are:
	• `Type` is `MyStdev`
	• `Comparison` is 14.61245
	• `Observation` is 12.507997

Stdev - chart function

Stdev() finds the standard deviation of the range of data aggregated in the expression or field iterated over the chart dimensions.

Syntax:

```
Stdev([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Examples and results:

Add the example script to your app and run it. Then build a straight table with `Type` as dimension and `Stdev(Value)` as measure.

`Totals` should be enabled in the properties of the table.

Example	Result

`Stdev(Value)`
`Table1:`
`crosstable LOAD recno()`
`as ID, * inline [`
`Observation|Comparison`
`35|2`
`40|27`
`12|38`
`15|31`
`21|1`
`14|19`
`46|1`
`10|34`
`28|3`
`48|1`
`16|2`
`30|3`
`32|2`
`48|1`
`31|2`
`22|1`
`12|3`
`39|29`
`19|37`
`25|2 ] (delimiter is`
`'|');`
	The results of the Stdev(Value) calculation are:
	• `Total` is 15.47529
	• `Comparison` is 14.61245
	• `Observation` is 12.507997

See also:

Avg - chart function (page 324)
STEYX - chart function (page 376)

Sterr

Sterr() returns the aggregated standard error (stdev/sqrt(n)) for a series of values represented by the expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
Sterr ([distinct] expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
distinct	If the word **distinct** occurs before the expression, all duplicates will be disregarded.

Limitations:

Text values, NULL values and missing values are disregarded.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Resulting data

Example	Result

`Table1:`
`crosstable LOAD recno() as ID, * inline [`
`Observation|Comparison`
`35|2`
`40|27`
`12|38`
`15|31`
`21|1`
`14|19`
`46|1`
`10|34`
`28|3`
`48|1`
`16|2`
`30|3`
`32|2`
`48|1`
`31|2`
`22|1`
`12|3`
`39|29`
`19|37`
`25|2 ] (delimiter is '|');`
 	
`Sterr1:`
`LOAD Type,`
`Sterr(Value) as MySterr`
`Resident Table1 Group By Type;`
	In a table with the dimensions `Type` and `MySterr`, the results of the Sterr() calculation in the data load script are:
	
	`Type MySterr`
	`Comparison 3.2674431`
	`Observation 2.7968733`

Sterr - chart function

Sterr() finds the value of the standard error of the mean, (stdev/sqrt(n)), for the series of values aggregated in the expression iterated over the chart dimensions.

Syntax:

```
Sterr([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] expr)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values are disregarded.

Examples and results:

Add the example script to your app and run it. Then build a straight table with `Type` as dimension and `Sterr (Value)` as measure.

`Totals` should be enabled in the properties of the table.

Example	Result

`Table1:`
`crosstable LOAD recno() as ID, * inline [`
`Observation|Comparison`
`35|2`
`40|27`
`12|38`
`15|31`
`21|1`
`14|19`
`46|1`
`10|34`
`28|3`
`48|1`
`16|2`
`30|3`
`32|2`
`48|1`
`31|2`
`22|1`
`12|3`
`39|29`
`19|37`
`25|2 ] (delimiter is '|');`
	The results of the Sterr(Value) calculation are:
	• `Total` is 2.4468583
	• `Comparison` is 3.2674431
	• `Observation` is 2.7968733

See also:

Avg - chart function (page 324)
STEYX - chart function (page 376)

STEYX

STEYX() returns the aggregated standard error of the predicted y-value for each x-value in the regression for a series of coordinates represented by paired numbers in x-expression and y-expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
STEYX (y_value, x_value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of y-values to be measured.
x_value	The expression or field containing the range of x-values to be measured.

Limitations:

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Resulting data

Example	Result

`Trend:`
`Load *, 1 as Grp;`
`LOAD * inline [`
`Month|KnownY|KnownX`
`Jan|2|6`
`Feb|3|5`
`Mar|9|11`
`Apr|6|7`
`May|8|5`
`Jun|7|4`
`Jul|5|5`
`Aug|10|8`
`Sep|9|10`
`Oct|12|14`
`Nov|15|17`
`Dec|14|16`
`] (delimiter is '|');`
 	
`STEYX1:`
`LOAD Grp,`
`STEYX(KnownY, KnownX)`
`as MySTEYX`
`Resident Trend Group`
`By Grp;`
	In a table with the dimension `MySTEYX`, the result of the STEYX() calculation in the data load script is 2.0714764.

STEYX - chart function

STEYX() returns the aggregated standard error when predicting y-values for each x-value in a linear regression given by a series of coordinates represented by paired numbers in the expressions given by **y_value** and **x_value**.

Syntax:

```
STEYX([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] y_value, x_value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
y_value	The expression or field containing the range of known y-values to be measured.
x_value	The expression or field containing the range of known x-values to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

The parameter of the aggregation function must not contain other aggregation functions, unless these inner aggregations contain the **TOTAL** qualifier. For more advanced nested aggregations, use the advanced function **Aggr**, in combination with a specified dimension.

Text values, NULL values and missing values in any or both pieces of a data-pair result in the entire data-pair being disregarded.

Examples and results:

Add the example script to your app and run it. Then build a straight table with `KnownY` and `KnownX` as dimension and `Steyx(KnownY,KnownX)` as measure.

`Totals` should be enabled in the properties of the table.

Example	Result

`Trend:`
`LOAD * inline [`
`Month|KnownY|KnownX`
`Jan|2|6`
`Feb|3|5`
`Mar|9|11`
`Apr|6|7`
`May|8|5`
`Jun|7|4`
`Jul|5|5`
`Aug|10|8`
`Sep|9|10`
`Oct|12|14`
`Nov|15|17`
`Dec|14|16`
`] (delimiter is '|');`
	The result of the STEYX(KnownY,KnownX) calculation is 2.071 (If number formatting is set to 3 decimal places.)

See also:

Avg - chart function (page 324)
Sterr - chart function (page 373)

Examples of how to use linest functions

The linest functions are used to find values associated with linear regression analysis. This section describes how to build visualizations using sample data to find the values of the linest functions available in Qlik Sense. The linest functions can be used in the data load script and in chart expressions.

Refer to the individual linest chart function and script function topics for descriptions of syntax and arguments.

Data and script expressions used in the examples

Load the following inline data and script expressions in the data load editor for the linest() examples below.

```
T1:
LOAD *, 1 as Grp;
LOAD * inline [
X|Y
1|0
2|1
3|3
4|8
5|14
6|20
7|0
8|50
9|25
10|60
11|38
12|19
13|26
14|143
15|98
```

```
16|27
17|59
18|78
19|158
20|279 ] (delimiter is '|');

R1:
LOAD
Grp,
linest_B(Y,X) as Linest_B,
linest_DF(Y,X) as Linest_DF,
linest_F(Y,X) as Linest_F,
linest_M(Y,X) as Linest_M,
linest_R2(Y,X) as Linest_R2,
linest_SEB(Y,X,1,1) as Linest_SEB,
linest_SEM(Y,X) as Linest_SEM,
linest_SEY(Y,X) as Linest_SEY,
linest_SSREG(Y,X) as Linest_SSREG,
linest_SSRESID(Y,X) as Linest_SSRESID
resident T1 group by Grp;
```

Example 1: Script expressions using linest

Example: Script expressions

Create a visualization from the data load script calculations

Create a table visualization in a Qlik Sense sheet with the following fields as columns:

- Linest_B
- Linest_DF
- Linest_F
- Linest_M
- Linest_R2
- Linest_SEB
- Linest_SEM
- Linest_SEY
- Linest_SSREG
- Linest_SSRESID

Result

The table containing the results of the linest calculations made in the data load script should look like this:

Results table

Linest_B	Linest_DF	Linest_F	Linest_M	Linest_R2	Linest_SEB
-35.047	18	20.788	8.605	0.536	22.607

Results table

Linest_SEM	Linest_SEY	Linest_SSREG	Linest_SSRESID
1.887	48.666	49235.014	42631.186

Example 2: Chart expressions using linest

Example: Chart expressions

Create a table visualization in a Qlik Sense sheet with the following fields as dimensions:

```
ValueList('Linest_b', 'Linest_df','Linest_f', 'Linest_m','Linest_r2','Linest_SEB','Linest_
SEM','Linest_SEY','Linest_SSREG','Linest_SSRESID')
```

This expression uses the synthetic dimensions function to create labels for the dimensions with the names of the linest functions. You can change the label to **Linest functions** to save space.

Add the following expression to the table as a measure:

```
Pick(Match(ValueList('Linest_b', 'Linest_df','Linest_f', 'Linest_m','Linest_r2','Linest_
SEB','Linest_SEM','Linest_SEY','Linest_SSREG','Linest_SSRESID'),'Linest_b', 'Linest_
df','Linest_f', 'Linest_m','Linest_r2','Linest_SEB','Linest_SEM','Linest_SEY','Linest_
SSREG','Linest_SSRESID'),Linest_b(Y,X),Linest_df(Y,X),Linest_f(Y,X),Linest_m(Y,X),Linest_r2
(Y,X),Linest_SEB(Y,X,1,1),Linest_SEM(Y,X),Linest_SEY(Y,X),Linest_SSREG(Y,X),Linest_SSRESID
(Y,X) )
```

This expression displays the value of the result of each linest function against the corresponding name in the synthetic dimension. The result of `Linest_b(Y,X)` is displayed next to **linest_b**, and so on.

Result

Results table

Linest functions	Linest function results
Linest_b	-35.047
Linest_df	18
Linest_f	20.788
Linest_m	8.605
Linest_r2	0.536
Linest_SEB	22.607
Linest_SEM	1.887
Linest_SEY	48.666
Linest_SSREG	49235.014
Linest_SSRESID	42631.186

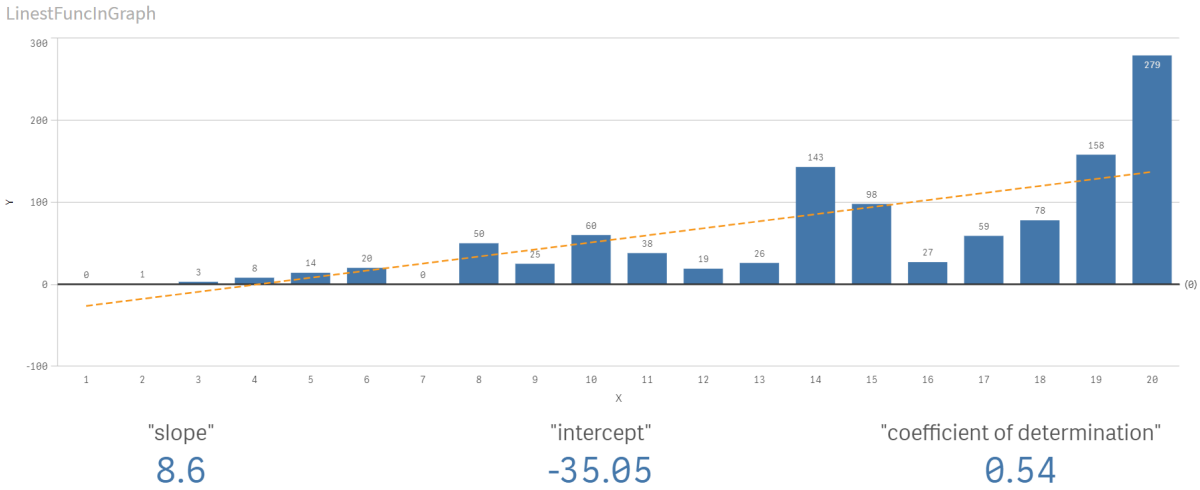
Example 3: Chart expressions using linest

Example: Chart expressions

1. Create a barchart visualization in a Qlik Sense sheet with **X** as a dimension and **Y** as a measure.
2. Add a linear trend line to the Y measure.

3. Add a KPI visualization to the sheet.
    1. Add *slope* as a label for the KPI.
    2. Add `sum(Linest_M)` as an expression for the KPI.
4. Add a second KPI visualization to the sheet.
    1. Add *intercept* as a label for the KPI.
    2. Add `Sum(Linest_B)` as an expression for the KPI.
5. Add a third KPI visualization to the sheet.
    1. Add *coefficient of determination* as a label for the KPI.
    2. Add `Sum(Linest_R2)` as an expression for the KPI.

Result



Explanation

The barchart shows the plotting of the X and Y data. Relevant linest() functions provide values for the linear regression equation that the trend line is based on, namely y = m * x + b. The equation uses the "least squares" method to calculate a straight line (trend line) by returning an array that describes a line that best fits the data.

The KPIs display the results of the linest() functions **sum(Linest_M)** for slope and **sum(Linest_B)** for the Y intercept, which are variables in the linear regression equation, and the corresponding aggregated R2 value for coefficient of determination.

Statistical test functions

Statistical test functions can be used in both the data load script and chart expressions, but the syntax differs.

Chi-2 test functions

Generally used in the study of qualitative variables. One can compare observed frequencies in a one-way frequency table with expected frequencies, or study the connection between two variables in a contingency table.

T-test functions

T-test functions are used for statistical examination of two population means. A two-sample t-test examines whether two samples are different and is commonly used when two normal distributions have unknown variances and when an experiment uses a small sample size.

Z-test functions

A statistical examination of two population means. A two sample z-test examines whether two samples are different and is commonly used when two normal distributions have known variances and when an experiment uses a large sample size.

Chi2-test functions

Generally used in the study of qualitative variables. One can compare observed frequencies in a one-way frequency table with expected frequencies, or study the connection between two variables in a contingency table.Chi-squared test functions are used to determine whether there is a statistically significant difference between the expected frequencies and the observed frequencies in one or more groups. Often a histogram is used, and the different bins are compared to an expected distribution.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Chi2Test_chi2

Chi2Test_chi2() returns the aggregated chi^2-test value for one or two series of values.

```
Chi2Test_chi2() returns the aggregated chi2-test value for one or two series
of values.(col, row, actual_value[, expected_value])
```

Chi2Test_df

Chi2Test_df() returns the aggregated chi^2-test df value (degrees of freedom) for one or two series of values.

```
Chi2Test_df() returns the aggregated chi2-test df value (degrees of freedom)
for one or two series of values.(col, row, actual_value[, expected_value])
```

Chi2Test_p

Chi2Test_p() returns the aggregated chi^2-test p value (significance) for one or two series of values.

```
Chi2Test_p - chart function(col, row, actual_value[, expected_value])
```

See also:

T-test functions (page 385)
Z-test functions (page 419)

Chi2Test_chi2

Chi2Test_chi2() returns the aggregated chi^2-test value for one or two series of values.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

All Qlik Sense chi^2 -test functions have the same arguments.

Syntax:

```
Chi2Test_chi2(col, row, actual_value[, expected_value])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
col, row	The specified column and row in the matrix of values being tested.
actual_value	The observed value of the data at the specified **col** and **row**.
expected_value	The expected value for the distribution at the specified **col** and **row**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
Chi2Test_chi2( Grp, Grade, Count )
Chi2Test_chi2( Gender, Description, Observed, Expected )
```

See also:

Examples of how to use chi2-test functions in charts (page 434)
Examples of how to use chi2-test functions in the data load script (page 437)

Chi2Test_df

Chi2Test_df() returns the aggregated chi^2-test df value (degrees of freedom) for one or two series of values.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

All Qlik Sense chi^2 -test functions have the same arguments.

Syntax:

```
Chi2Test_df(col, row, actual_value[, expected_value])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
col, row	The specified column and row in the matrix of values being tested.
actual_value	The observed value of the data at the specified **col** and **row**.
expected_value	The expected value for the distribution at the specified **col** and **row**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
Chi2Test_df( Grp, Grade, Count )
Chi2Test_df( Gender, Description, Observed, Expected )
```

See also:

Examples of how to use chi2-test functions in charts (page 434)
Examples of how to use chi2-test functions in the data load script (page 437)

Chi2Test_p - chart function

Chi2Test_p() returns the aggregated chi^2-test p value (significance) for one or two series of values. The test can be done either on the values in **actual_value**, testing for variations within the specified **col** and **row** matrix, or by comparing values in **actual_value** with corresponding values in **expected_value**, if specified.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

All Qlik Sense chi^2 -test functions have the same arguments.

Syntax:

```
Chi2Test_p(col, row, actual_value[, expected_value])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
col, row	The specified column and row in the matrix of values being tested.
actual_value	The observed value of the data at the specified **col** and **row**.
expected_value	The expected value for the distribution at the specified **col** and **row**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
Chi2Test_p( Grp, Grade, Count )
Chi2Test_p( Gender, Description, Observed, Expected )
```

See also:

Examples of how to use chi2-test functions in charts (page 434)
Examples of how to use chi2-test functions in the data load script (page 437)

T-test functions

T-test functions are used for statistical examination of two population means. A two-sample t-test examines whether two samples are different and is commonly used when two normal distributions have unknown variances and when an experiment uses a small sample size.

In the following sections, the t-test statistical test functions are grouped according to the sample student test that applies to each type of function.

Creating a typical t-test report (page 438)

Two independent samples t-tests

The following functions apply to two independent samples student's t-tests.

ttest_conf

TTest_conf returns the aggregated t-test confidence interval value for two independent samples.

```
TTest_conf returns the aggregated t-test confidence interval value for two
independent samples. ( grp, value [, sig[, eq_var]])
```

ttest_df

TTest_df() returns the aggregated student's t-test value (degrees of freedom) for two independent series of values.

```
TTest_df() returns the aggregated student's t-test value (degrees of freedom)
for two independent series of values. (grp, value [, eq_var)
```

ttest_dif

TTest_dif() is a numeric function that returns the aggregated student's t-test mean difference for two independent series of values.

```
TTest_dif() is a numeric function that returns the aggregated student's t-
test mean difference for two independent series of values. (grp, value)
```

ttest_lower

TTest_lower() returns the aggregated value for the lower end of the confidence interval for two independent series of values.

```
TTest_lower() returns the aggregated value for the lower end of the
confidence interval for two independent series of values. (grp, value [, sig
[, eq_var]])
```

ttest_sig

TTest_sig() returns the aggregated student's t-test 2-tailed level of significance for two independent series of values.

```
TTest_sig() returns the aggregated student's t-test 2-tailed level of
significance for two independent series of values. (grp, value [, eq_var])
```

ttest_sterr

TTest_sterr() returns the aggregated student's t-test standard error of the mean difference for two independent series of values.

```
TTest_sterr() returns the aggregated student's t-test standard error of the
mean difference for two independent series of values. (grp, value [, eq_var])
```

ttest_t

TTest_t() returns the aggregated t value for two independent series of values.

```
TTest_t() returns the aggregated t value for two independent series of
values. (grp, value [, eq_var])
```

ttest_upper

TTest_upper() returns the aggregated value for the upper end of the confidence interval for two independent series of values.

```
TTest_upper() returns the aggregated value for the upper end of the
confidence interval for two independent series of values. (grp, value [, sig
[, eq_var]])
```

Two independent weighted samples t-tests

The following functions apply to two independent samples student's t-tests where the input data series is given in weighted two-column format.

ttestw_conf

TTestw_conf() returns the aggregated t value for two independent series of values.

```
TTestw_conf() returns the aggregated t value for two independent series of
values. (weight, grp, value [, sig[, eq_var]])
```

ttestw_df

TTestw_df() returns the aggregated student's t-test df value (degrees of freedom) for two independent series of values.

```
TTestw_df() returns the aggregated student's t-test df value (degrees of
freedom) for two independent series of values. (weight, grp, value [, eq_
var])
```

ttestw_dif

TTestw_dif() returns the aggregated student's t-test mean difference for two independent series of values.

```
TTestw_dif() returns the aggregated student's t-test mean difference for two
independent series of values. ( weight, grp, value)
```

ttestw_lower

TTestw_lower() returns the aggregated value for the lower end of the confidence interval for two independent series of values.

```
TTestw_lower() returns the aggregated value for the lower end of the
confidence interval for two independent series of values. (weight, grp, value
[, sig[, eq_var]])
```

ttestw_sig

TTestw_sig() returns the aggregated student's t-test 2-tailed level of significance for two independent series of values.

```
TTestw_sig() returns the aggregated student's t-test 2-tailed level of
significance for two independent series of values. ( weight, grp, value [,
eq_var])
```

ttestw_sterr

TTestw_sterr() returns the aggregated student's t-test standard error of the mean difference for two independent series of values.

```
TTestw_sterr() returns the aggregated student's t-test standard error of the
mean difference for two independent series of values. (weight, grp, value [,
eq_var])
```

ttestw_t

TTestw_t() returns the aggregated t value for two independent series of values.

```
TTestw_t() returns the aggregated t value for two independent series of
values. (weight, grp, value [, eq_var])
```

ttestw_upper

TTestw_upper() returns the aggregated value for the upper end of the confidence interval for two independent series of values.

```
TTestw_upper() returns the aggregated value for the upper end of the
confidence interval for two independent series of values. (weight, grp, value
[, sig [, eq_var]])
```

One sample t-tests

The following functions apply to one-sample student's t-tests.

ttest1_conf

TTest1_conf() returns the aggregated confidence interval value for a series of values.

```
TTest1_conf() returns the aggregated confidence interval value for a series
of values. (value [, sig])
```

ttest1_df

TTest1_df() returns the aggregated student's t-test df value (degrees of freedom) for a series of values.

```
TTest1_df() returns the aggregated student's t-test df value (degrees of
freedom) for a series of values. (value)
```

ttest1_dif

TTest1_dif() returns the aggregated student's t-test mean difference for a series of values.

```
TTest1_dif() returns the aggregated student's t-test mean difference for a
series of values. (value)
```

ttest1_lower

TTest1_lower() returns the aggregated value for the lower end of the confidence interval for a series of values.

```
TTest1_lower() returns the aggregated value for the lower end of the
confidence interval for a series of values. (value [, sig])
```

ttest1_sig

TTest1_sig() returns the aggregated student's t-test 2-tailed level of significance for a series of values.

```
TTest1_sig() returns the aggregated student's t-test 2-tailed level of
significance for a series of values. (value)
```

ttest1_sterr

TTest1_sterr() returns the aggregated student's t-test standard error of the mean difference for a series of values.

```
TTest1_sterr() returns the aggregated student's t-test standard error of the
mean difference for a series of values. (value)
```

ttest1_t

TTest1_t() returns the aggregated t value for a series of values.

```
TTest1_t() returns the aggregated t value for a series of values. (value)
```

ttest1_upper

TTest1_upper() returns the aggregated value for the upper end of the confidence interval for a series of values.

```
TTest1_upper() returns the aggregated value for the upper end of the
confidence interval for a series of values. (value [, sig])
```

One weighted sample t-tests

The following functions apply to one-sample student's t-tests where the input data series is given in weighted two-column format.

ttest1w_conf

TTest1w_conf() is a **numeric** function that returns the aggregated confidence interval value for a series of values.

```
TTest1w_conf() is a numeric function that returns the aggregated confidence
interval value for a series of values. (weight, value [, sig])
```

ttest1w_df

TTest1w_df() returns the aggregated student's t-test df value (degrees of freedom) for a series of values.

```
TTest1w_df() returns the aggregated student's t-test df value (degrees of
freedom) for a series of values. (weight, value)
```

ttest1w_dif

TTest1w_dif() returns the aggregated student's t-test mean difference for a series of values.

```
TTest1w_dif() returns the aggregated student's t-test mean difference for a
series of values. (weight, value)
```

ttest1w_lower

TTest1w_lower() returns the aggregated value for the lower end of the confidence interval for a series of values.

```
TTest1w_lower() returns the aggregated value for the lower end of the
confidence interval for a series of values. (weight, value [, sig])
```

ttest1w_sig

TTest1w_sig() returns the aggregated student's t-test 2-tailed level of significance for a series of values.

```
TTest1w_sig() returns the aggregated student's t-test 2-tailed level of
significance for a series of values. (weight, value)
```

ttest1w_sterr

TTest1w_sterr() returns the aggregated student's t-test standard error of the mean difference for a series of values.

```
TTest1w_sterr() returns the aggregated student's t-test standard error of the
mean difference for a series of values. (weight, value)
```

ttest1w_t

TTest1w_t() returns the aggregated t value for a series of values.

```
TTest1w_t() returns the aggregated t value for a series of values. ( weight,
value)
```

ttest1w_upper

TTest1w_upper() returns the aggregated value for the upper end of the confidence interval for a series of values.

```
TTest1w_upper() returns the aggregated value for the upper end of the
confidence interval for a series of values. (weight, value [, sig])
```

TTest_conf

TTest_conf returns the aggregated t-test confidence interval value for two independent samples.

This function applies to independent samples student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest_conf ( grp, value [, sig [, eq_var]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.

Argument	Description
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest_conf( Group, Value )
TTest_conf( Group, Value, Sig, false )
```

See also:

Creating a typical t-test report (page 438)

TTest_df

TTest_df() returns the aggregated student's t-test value (degrees of freedom) for two independent series of values.

This function applies to independent samples student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest_df (grp, value [, eq_var])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest_df( Group, Value )
TTest_df( Group, Value, false )
```

See also:

Creating a typical t-test report (page 438)

TTest_dif

TTest_dif() is a numeric function that returns the aggregated student's t-test mean difference for two independent series of values.

This function applies to independent samples student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest_dif (grp, value [, eq_var] )
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest_dif( Group, Value )
TTest_dif( Group, Value, false )
```

See also:

Creating a typical t-test report (page 438)

TTest_lower

TTest_lower() returns the aggregated value for the lower end of the confidence interval for two independent series of values.

This function applies to independent samples student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest_lower (grp, value [, sig [, eq_var]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest_lower( Group, Value )
TTest_lower( Group, Value, Sig, false )
```

See also:

Creating a typical t-test report (page 438)

TTest_sig

TTest_sig() returns the aggregated student's t-test 2-tailed level of significance for two independent series of values.

This function applies to independent samples student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest_sig (grp, value [, eq_var])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest_sig( Group, Value )
TTest_sig( Group, Value, false )
```

See also:

Creating a typical t-test report (page 438)

TTest_sterr

TTest_sterr() returns the aggregated student's t-test standard error of the mean difference for two independent series of values.

This function applies to independent samples student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest_sterr (grp, value [, eq_var])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest_sterr( Group, Value )
TTest_sterr( Group, Value, false )
```

See also:

Creating a typical t-test report (page 438)

TTest_t

TTest_t() returns the aggregated t value for two independent series of values.

This function applies to independent samples student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest_t(grp, value[, eq_var])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
TTest_t( Group, Value, false )
```

See also:

Creating a typical t-test report (page 438)

TTest_upper

TTest_upper() returns the aggregated value for the upper end of the confidence interval for two independent series of values.

This function applies to independent samples student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest_upper (grp, value [, sig [, eq_var]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest_upper( Group, Value )
TTest_upper( Group, Value, sig, false )
```

See also:

Creating a typical t-test report (page 438)

TTestw_conf

TTestw_conf() returns the aggregated t value for two independent series of values.

This function applies to two independent samples student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTestw_conf (weight, grp, value [, sig [, eq_var]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTestw_conf( Weight, Group, Value )
TTestw_conf( Weight, Group, Value, sig, false )
```

See also:

Creating a typical t-test report (page 438)

TTestw_df

TTestw_df() returns the aggregated student's t-test df value (degrees of freedom) for two independent series of values.

This function applies to two independent samples student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTestw_df (weight, grp, value [, eq_var])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTestw_df( Weight, Group, Value )
TTestw_df( Weight, Group, Value, false )
```

See also:

Creating a typical t-test report (page 438)

TTestw_dif

TTestw_dif() returns the aggregated student's t-test mean difference for two independent series of values.

This function applies to two independent samples student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTestw_dif (weight, grp, value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTestw_dif( Weight, Group, Value )
TTestw_dif( Weight, Group, Value, false )
```

See also:

Creating a typical t-test report (page 438)

TTestw_lower

TTestw_lower() returns the aggregated value for the lower end of the confidence interval for two independent series of values.

This function applies to two independent samples student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTestw_lower (weight, grp, value [, sig [, eq_var]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTestw_lower( Weight, Group, Value )
TTestw_lower( Weight, Group, Value, sig, false )
```

See also:

Creating a typical t-test report (page 438)

TTestw_sig

TTestw_sig() returns the aggregated student's t-test 2-tailed level of significance for two independent series of values.

This function applies to two independent samples student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTestw_sig ( weight, grp, value [, eq_var])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTestw_sig( Weight, Group, Value )
TTestw_sig( Weight, Group, Value, false )
```

See also:

Creating a typical t-test report (page 438)

TTestw_sterr

TTestw_sterr() returns the aggregated student's t-test standard error of the mean difference for two independent series of values.

This function applies to two independent samples student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTestw_sterr (weight, grp, value [, eq_var])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTestw_sterr( Weight, Group, Value )
TTestw_sterr( Weight, Group, Value, false )
```

See also:

Creating a typical t-test report (page 438)

TTestw_t

TTestw_t() returns the aggregated t value for two independent series of values.

This function applies to two independent samples student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ttestw_t (weight, grp, value [, eq_var])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTestw_t( Weight, Group, Value )
TTestw_t( Weight, Group, Value, false )
```

See also:

Creating a typical t-test report (page 438)

TTestw_upper

TTestw_upper() returns the aggregated value for the upper end of the confidence interval for two independent series of values.

This function applies to two independent samples student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTestw_upper (weight, grp, value [, sig [, eq_var]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTestw_upper( Weight, Group, Value )
TTestw_upper( Weight, Group, Value, sig, false )
```

See also:

Creating a typical t-test report (page 438)

TTest1_conf

TTest1_conf() returns the aggregated confidence interval value for a series of values.

This function applies to one-sample student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1_conf (value [, sig ])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest1_conf( value )
TTest1_conf( value, 0.005 )
```

See also:

Creating a typical t-test report (page 438)

TTest1_df

TTest1_df() returns the aggregated student's t-test df value (degrees of freedom) for a series of values.

This function applies to one-sample student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1_df (value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
TTest1_df( Value )
```

See also:

Creating a typical t-test report (page 438)

TTest1_dif

TTest1_dif() returns the aggregated student's t-test mean difference for a series of values.

This function applies to one-sample student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1_dif (value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
TTest1_dif( Value )
```

See also:

Creating a typical t-test report (page 438)

TTest1_lower

TTest1_lower() returns the aggregated value for the lower end of the confidence interval for a series of values.

This function applies to one-sample student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1_lower (value [, sig])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest1_lower( value )
TTest1_lower( value, 0.005 )
```

See also:

Creating a typical t-test report (page 438)

TTest1_sig

TTest1_sig() returns the aggregated student's t-test 2-tailed level of significance for a series of values.

This function applies to one-sample student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1_sig (value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
TTest1_sig( Value )
```

See also:

Creating a typical t-test report (page 438)

TTest1_sterr

TTest1_sterr() returns the aggregated student's t-test standard error of the mean difference for a series of values.

This function applies to one-sample student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1_sterr (value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
TTest1_sterr( Value )
```

See also:

Creating a typical t-test report (page 438)

TTest1_t

TTest1_t() returns the aggregated t value for a series of values.

This function applies to one-sample student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1_t (value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
TTest1_t( Value )
```

See also:

Creating a typical t-test report (page 438)

TTest1_upper

TTest1_upper() returns the aggregated value for the upper end of the confidence interval for a series of values.

This function applies to one-sample student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1_upper (value [, sig])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest1_upper( value )
TTest1_upper( value, 0.005 )
```

See also:

Creating a typical t-test report (page 438)

TTest1w_conf

TTest1w_conf() is a **numeric** function that returns the aggregated confidence interval value for a series of values.

This function applies to one-sample student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1w_conf (weight, value [, sig ])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest1w_conf( Weight, Value )
TTest1w_conf( Weight, Value, 0.005 )
```

See also:

Creating a typical t-test report (page 438)

TTest1w_df

TTest1w_df() returns the aggregated student's t-test df value (degrees of freedom) for a series of values.

This function applies to one-sample student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1w_df (weight, value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
TTest1w_df( Weight, Value )
```

See also:

Creating a typical t-test report (page 438)

TTest1w_dif

TTest1w_dif() returns the aggregated student's t-test mean difference for a series of values.

This function applies to one-sample student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1w_dif (weight, value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
TTest1w_dif( Weight, Value )
```

See also:

Creating a typical t-test report (page 438)

TTest1w_lower

TTest1w_lower() returns the aggregated value for the lower end of the confidence interval for a series of values.

This function applies to one-sample student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1w_lower (weight, value [, sig ])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest1w_lower( Weight, Value )
TTest1w_lower( Weight, Value, 0.005 )
```

See also:

Creating a typical t-test report (page 438)

TTest1w_sig

TTest1w_sig() returns the aggregated student's t-test 2-tailed level of significance for a series of values.

This function applies to one-sample student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1w_sig (weight, value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
TTest1w_sig( Weight, Value )
```

See also:

Creating a typical t-test report (page 438)

TTest1w_sterr

TTest1w_sterr() returns the aggregated student's t-test standard error of the mean difference for a series of values.

This function applies to one-sample student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1w_sterr (weight, value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
TTest1w_sterr( Weight, Value )
```

See also:

Creating a typical t-test report (page 438)

TTest1w_t

TTest1w_t() returns the aggregated t value for a series of values.

This function applies to one-sample student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1w_t ( weight, value)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
TTest1w_t( Weight, Value )
```

See also:

Creating a typical t-test report (page 438)

TTest1w_upper

TTest1w_upper() returns the aggregated value for the upper end of the confidence interval for a series of values.

This function applies to one-sample student's t-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
TTest1w_upper (weight, value [, sig])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The samples to be evaluated. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
weight	Each value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
TTest1w_upper( Weight, Value )
TTest1w_upper( Weight, Value, 0.005 )
```

See also:

Creating a typical t-test report (page 438)

Z-test functions

A statistical examination of two population means. A two sample z-test examines whether two samples are different and is commonly used when two normal distributions have known variances and when an experiment uses a large sample size.

The z-test statistical test functions are grouped according the type of input data series that applies to the function.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Examples of how to use z-test functions (page 441)

One column format functions

The following functions apply to z-tests with simple input data series.

ztest_conf

ZTest_conf() returns the aggregated z value for a series of values.

```
ZTest_conf() returns the aggregated z value for a series of values. (value [,
sigma [, sig ])
```

ztest_dif

ZTest_dif() returns the aggregated z-test mean difference for a series of values.

```
ZTest_dif() returns the aggregated z-test mean difference for a series of
values. (value [, sigma])
```

ztest_sig

ZTest_sig() returns the aggregated z-test 2-tailed level of significance for a series of values.

```
ZTest_sig() returns the aggregated z-test 2-tailed level of significance for
a series of values. (value [, sigma])
```

ztest_sterr

ZTest_sterr() returns the aggregated z-test standard error of the mean difference for a series of values.

```
ZTest_sterr() returns the aggregated z-test standard error of the mean
difference for a series of values. (value [, sigma])
```

ztest_z

ZTest_z() returns the aggregated z value for a series of values.

```
ZTest_z() returns the aggregated z value for a series of values. (value [,
sigma])
```

ztest_lower

ZTest_lower() returns the aggregated value for the lower end of the confidence interval for two independent series of values.

```
ZTest_lower() returns the aggregated value for the lower end of the
confidence interval for two independent series of values. (grp, value [, sig
[, eq_var]])
```

ztest_upper

ZTest_upper() returns the aggregated value for the upper end of the confidence interval for two independent series of values.

```
ZTest_upper() returns the aggregated value for the upper end of the
confidence interval for two independent series of values. (grp, value [, sig
[, eq_var]])
```

Weighted two-column format functions

The following functions apply to z-tests where the input data series is given in weighted two-column format.

ztestw_conf

ZTestw_conf() returns the aggregated z confidence interval value for a series of values.

```
ZTestw_conf() returns the aggregated z confidence interval value for a series
of values. (weight, value [, sigma [, sig]])
```

ztestw_dif

ZTestw_dif() returns the aggregated z-test mean difference for a series of values.

```
ZTestw_dif() returns the aggregated z-test mean difference for a series of
values.  (weight, value [, sigma])
```

ztestw_lower

ZTestw_lower() returns the aggregated value for the lower end of the confidence interval for two independent series of values.

```
ZTestw_lower() returns the aggregated value for the lower end of the
confidence interval for two independent series of values. (weight, value [,
sigma])
```

ztestw_sig

ZTestw_sig() returns the aggregated z-test 2-tailed level of significance for a series of values.

```
ZTestw_sig() returns the aggregated z-test 2-tailed level of significance for
a series of values. (weight, value [, sigma])
```

ztestw_sterr

ZTestw_sterr() returns the aggregated z-test standard error of the mean difference for a series of values.

```
ZTestw_sterr() returns the aggregated z-test standard error of the mean
difference for a series of values. (weight, value [, sigma])
```

ztestw_upper

ZTestw_upper() returns the aggregated value for the upper end of the confidence interval for two independent series of values.

```
ZTestw_upper() returns the aggregated value for the upper end of the
confidence interval for two independent series of values. (weight, value [,
sigma])
```

ztestw_z

ZTestw_z() returns the aggregated z value for a series of values.

```
ZTestw_z() returns the aggregated z value for a series of values. (weight,
value [, sigma])
```

ZTest_z

ZTest_z() returns the aggregated z value for a series of values.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTest_z(value[, sigma])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. A population mean of 0 is assumed. If you want the test to be performed around another mean, subtract that mean from the sample values.
sigma	If known, the standard deviation can be stated in **sigma**. If **sigma** is omitted the actual sample standard deviation will be used.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
ZTest_z( Value-TestValue )
```

See also:

Examples of how to use z-test functions (page 441)

ZTest_sig

ZTest_sig() returns the aggregated z-test 2-tailed level of significance for a series of values.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTest_sig(value[, sigma])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. A population mean of 0 is assumed. If you want the test to be performed around another mean, subtract that mean from the sample values.
sigma	If known, the standard deviation can be stated in **sigma**. If **sigma** is omitted the actual sample standard deviation will be used.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
ZTest_sig(Value-TestValue)
```

See also:

Examples of how to use z-test functions (page 441)

ZTest_dif

ZTest_dif() returns the aggregated z-test mean difference for a series of values.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTest_dif(value[, sigma])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. A population mean of 0 is assumed. If you want the test to be performed around another mean, subtract that mean from the sample values.
sigma	If known, the standard deviation can be stated in **sigma**. If **sigma** is omitted the actual sample standard deviation will be used.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
ZTest_dif(Value-TestValue)
```

See also:

Examples of how to use z-test functions (page 441)

ZTest_sterr

ZTest_sterr() returns the aggregated z-test standard error of the mean difference for a series of values.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTest_sterr(value[, sigma])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. A population mean of 0 is assumed. If you want the test to be performed around another mean, subtract that mean from the sample values.
sigma	If known, the standard deviation can be stated in **sigma**. If **sigma** is omitted the actual sample standard deviation will be used.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
ZTest_sterr(Value-TestValue)
```

See also:

Examples of how to use z-test functions (page 441)

ZTest_conf

ZTest_conf() returns the aggregated z value for a series of values.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTest_conf(value[, sigma[, sig]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. A population mean of 0 is assumed. If you want the test to be performed around another mean, subtract that mean from the sample values.
sigma	If known, the standard deviation can be stated in **sigma**. If **sigma** is omitted the actual sample standard deviation will be used.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
ZTest_conf(Value-TestValue)
```

See also:

Examples of how to use z-test functions (page 441)

ZTest_lower

ZTest_lower() returns the aggregated value for the lower end of the confidence interval for two independent series of values.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTest_lower (grp, value [, sig [, eq_var]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
ZTest_lower( Group, Value )
ZTest_lower( Group, Value, sig, false )
```

See also:

Examples of how to use z-test functions (page 441)

ZTest_upper

ZTest_upper() returns the aggregated value for the upper end of the confidence interval for two independent series of values.

This function applies to independent samples student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTest_upper (grp, value [, sig [, eq_var]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
ZTest_upper( Group, Value )
ZTest_upper( Group, Value, sig, false )
```

See also:

Examples of how to use z-test functions (page 441)

ZTestw_z

ZTestw_z() returns the aggregated z value for a series of values.

This function applies to z-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTestw_z (weight, value [, sigma])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The values should be returned by **value**. A sample mean of 0 is assumed. If you want the test to be performed around another mean, subtract that value from the sample values.
weight	Each sample value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
sigma	If known, the standard deviation can be stated in **sigma**. If **sigma** is omitted the actual sample standard deviation will be used.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
ZTestw_z( Weight, Value-TestValue)
```

See also:

Examples of how to use z-test functions (page 441)

ZTestw_sig

ZTestw_sig() returns the aggregated z-test 2-tailed level of significance for a series of values.

This function applies to z-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTestw_sig (weight, value [, sigma])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The values should be returned by **value**. A sample mean of 0 is assumed. If you want the test to be performed around another mean, subtract that value from the sample values.
weight	Each sample value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
sigma	If known, the standard deviation can be stated in **sigma**. If **sigma** is omitted the actual sample standard deviation will be used.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
ZTestw_sig( Weight, Value-TestValue)
```

See also:

Examples of how to use z-test functions (page 441)

ZTestw_dif

ZTestw_dif() returns the aggregated z-test mean difference for a series of values.

This function applies to z-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTestw_dif ( weight, value [, sigma])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The values should be returned by **value**. A sample mean of 0 is assumed. If you want the test to be performed around another mean, subtract that value from the sample values.
weight	Each sample value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
sigma	If known, the standard deviation can be stated in **sigma**. If **sigma** is omitted the actual sample standard deviation will be used.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
ZTestw_dif( Weight, Value-TestValue)
```

See also:

Examples of how to use z-test functions (page 441)

ZTestw_sterr

ZTestw_sterr() returns the aggregated z-test standard error of the mean difference for a series of values.

This function applies to z-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTestw_sterr (weight, value [, sigma])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The values should be returned by **value**. A sample mean of 0 is assumed. If you want the test to be performed around another mean, subtract that value from the sample values.
weight	Each sample value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
sigma	If known, the standard deviation can be stated in **sigma**. If **sigma** is omitted the actual sample standard deviation will be used.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
ZTestw_sterr( Weight, Value-TestValue)
```

See also:

Examples of how to use z-test functions (page 441)

ZTestw_conf

ZTestw_conf() returns the aggregated z confidence interval value for a series of values.

This function applies to z-tests where the input data series is given in weighted two-column format.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTest_conf(weight, value[, sigma[, sig]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. A population mean of 0 is assumed. If you want the test to be performed around another mean, subtract that mean from the sample values.
weight	Each sample value in **value** can be counted one or more times according to a corresponding weight value in **weight**.
sigma	If known, the standard deviation can be stated in **sigma**. If **sigma** is omitted the actual sample standard deviation will be used.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Example:

```
ZTestw_conf( Weight, Value-TestValue)
```

See also:

Examples of how to use z-test functions (page 441)

ZTestw_lower

ZTestw_lower() returns the aggregated value for the lower end of the confidence interval for two independent series of values.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTestw_lower (grp, value [, sig [, eq_var]])
```

Return data type: numeric

Arguments:

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
ZTestw_lower( Group, Value )
ZTestw_lower( Group, Value, sig, false )
```

See also:

Examples of how to use z-test functions (page 441)

ZTestw_upper

ZTestw_upper() returns the aggregated value for the upper end of the confidence interval for two independent series of values.

This function applies to independent samples student's t-tests.

If the function is used in the data load script, the values are iterated over a number of records as defined by a group by clause.

If the function is used in a chart expression, the values are iterated over the chart dimensions.

Syntax:

```
ZTestw_upper (grp, value [, sig [, eq_var]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in **group**. If a field name for the sample values is not provided in the load script, the field will automatically be named **Value**.
grp	The field containing the names of each of the two sample groups. If a field name for the group is not provided in the load script, the field will automatically be given the name **Type**.
sig	The two-tailed level of significance can be specified in **sig**. If omitted, **sig** is set to 0.025, resulting in a 95% confidence interval.
eq_var	If **eq_var** is specified as False (0), separate variances of the two samples will be assumed. If **eq_var** is specified as True (1), equal variances between the samples will be assumed.

Limitations:

Text values, NULL values and missing values in the expression value will result in the function returning NULL.

Examples:

```
ZTestw_upper( Group, Value )
ZTestw_upper( Group, Value, sig, false )
```

See also:

Examples of how to use z-test functions (page 441)

Statistical test function examples

This section includes examples of statistical test functions as applied to charts and the data load script.

Examples of how to use chi2-test functions in charts

The chi2-test functions are used to find values associated with chi squared statistical analysis.

This section describes how to build visualizations using sample data to find the values of the chi-squared distribution test functions available in Qlik Sense. Please refer to the individual chi2-test chart function topics for descriptions of syntax and arguments.

Loading the data for the samples

There are three sets of sample data describing three different statistical samples to be loaded into the script.

Do the following:

1. Create a new app.
2. In the data load, enter the following:
```
// Sample_1 data is pre-aggregated... Note: make sure you set your DecimalSep='.' at the
top of the script.
Sample_1:
LOAD * inline [
Grp,Grade,Count
I,A,15
I,B,7
I,C,9
I,D,20
I,E,26
I,F,19
II,A,10
II,B,11
II,C,7
II,D,15
II,E,21
II,F,16
];
// Sample_2 data is pre-aggregated: If raw data is used, it must be aggregated using
count()...
Sample_2:
LOAD * inline [
Sex,Opinion,OpCount
1,2,58
1,1,11
1,0,10
2,2,35
2,1,25
2,0,23 ] (delimiter is ',');
// Sample_3a data is transformed using the crosstable statement...
Sample_3a:
crosstable(Gender, Actual) LOAD
Description,
[Men (Actual)] as Men,
[Women (Actual)] as Women;
LOAD * inline [
Men (Actual),Women (Actual),Description
58,35,Agree
11,25,Neutral
10,23,Disagree ] (delimiter is ',');
// Sample_3b data is transformed using the crosstable statement...
Sample_3b:
crosstable(Gender, Expected) LOAD
Description,
[Men (Expected)] as Men,
[Women (Expected)] as Women;
LOAD * inline [
Men (Expected),Women (Expected),Description
45.35,47.65,Agree
17.56,18.44,Neutral
16.09,16.91,Disagree ] (delimiter is ',');
// Sample_3a and Sample_3b will result in a (fairly harmless) Synthetic Key...
```
3. Click to load data.

Creating the chi2-test chart function visualizations

Example: Sample 1

Do the following:

1. In the data load editor, click to go to the app view and then click the sheet you created before. The sheet view is opened.
2. Click **Edit sheet** to edit the sheet.
3. From **Charts** add a table, and from **Fields** add Grp, Grade, and Count as dimensions. This table shows the sample data.
4. Add another table with the following expression as a dimension:
   `ValueList('p','df','Chi2')`
   This uses the synthetic dimensions function to create labels for the dimensions with the names of the three chi2-test functions.
5. Add the following expression to the table as a measure:
   `IF(ValueList('p','df','Chi2')='p',Chi2Test_p(Grp,Grade,Count),`
   `IF(ValueList('p','df','Chi2')='df',Chi2Test_df(Grp,Grade,Count),`
   `Chi2Test_Chi2(Grp,Grade,Count)))`
   This has the effect of putting the resulting value of each chi2-test function in the table next to its associated synthetic dimension.
6. Set the **Number formatting** of the measure to **Number** and **3Significant figures**.

In the expression for the measure, you could use the following expression instead: `Pick(Match(ValueList('p','df','Chi2'),'p','df','Chi2'),Chi2Test_p(Grp,Grade,Count),Chi2Test_df(Grp,Grade,Count),Chi2Test_Chi2(Grp,Grade,Count))`

Result:

The resulting table for the chi2-test functions for the Sample 1 data will contain the following values:

Results table

p	df	Chi2
0.820	5	2.21

Example: Sample 2

Do the following:

1. In the sheet you were editing in the example Sample 1, from **Charts** add a table, and from **Fields** add Sex, Opinion, and OpCount as dimensions.
2. Make a copy of the results table from Sample 1 using the **Copy** and **Paste** commands. Edit the expression in the measure and replace the arguments in all three chi2-test functions with the names of the fields used in the Sample 2 data, for example: `Chi2Test_p(Sex,Opinion,OpCount)`.

Result:

The resulting table for the chi2-test functions for the Sample 2 data will contain the following values:

Results table

p	df	Chi2
0.000309	2	16.2

Example: Sample 3

Do the following:

1. Create two more tables in the same way as in the examples for Sample 1 and Sample 2 data. In the dimensions table, use the following fields as dimensions: Gender, Description, Actual, and Expected.
2. In the results table, use the names of the fields used in the Sample 3 data, for example: `Chi2Test_p (Gender,Description,Actual,Expected)`.

Result:

The resulting table for the chi2-test functions for the Sample 3 data will contain the following values:

Results table

p	df	Chi2
0.000308	2	16.2

Examples of how to use chi2-test functions in the data load script

The chi2-test functions are used to find values associated with chi squared statistical analysis. This section describes how to use the chi-squared distribution test functions available in Qlik Sense in the data load script. Please refer to the individual chi2-test script function topics for descriptions of syntax and arguments.

This example uses a table containing the number of students achieving a grade (A-F) for two groups of students (I and II).

Data table

Group	A	B	C	D	E	F
I	15	7	9	20	26	19
II	10	11	7	15	21	16

Loading the sample data

Do the following:

1. Create a new app.
2. In the data load editor, enter the following:
```
// Sample_1 data is pre-aggregated... Note: make sure you set your DecimalSep='.' at the top of the script.
Sample_1:
```

```
LOAD * inline [
Grp,Grade,Count
I,A,15
I,B,7
I,C,9
I,D,20
I,E,26
I,F,19
II,A,10
II,B,11
II,C,7
II,D,15
II,E,21
II,F,16
];
```

3. Click to load data.

You have now loaded the sample data.

Loading the chi2-test function values

Now we will load the chi2-test values based on the sample data in a new table, grouped by Grp.

Do the following:

1. In the data load editor, add the following at the end of the script:
```
// Sample_1 data is pre-aggregated... Note: make sure you set your DecimalSep='.' at the top of the script.
Chi2_table:
LOAD Grp,
Chi2Test_chi2(Grp, Grade, Count) as chi2,
Chi2Test_df(Grp, Grade, Count) as df,
Chi2Test_p(Grp, Grade, Count) as p
resident Sample_1 group by Grp;
```
2. Click to load data.

You have now loaded the chi2-test values in a table named Chi2_table.

Results

You can view the resulting chi2-test values in the data model viewer under **Preview**, they should look like this:

Results

Grp	chi2	df	p
I	16.00	5	0.007
II	9.40	5	0.094

Creating a typical t-test report

A typical student t-test report can include tables with **Group Statistics** and **Independent Samples Test** results.

In the following sections we will build these tables using Qlik Senset-test functions applied to two independent groups of samples, Observation and Comparison. The corresponding tables for these samples would look like this:

Group statistics

Type	N	Mean	Standard Deviation	Standard Error Mean
Comparison	20	11.95	14.61245	3.2674431
Observation	20	27.15	12.507997	2.7968933

Independent Sample Test

Independent Sample Test

Type	t	df	Sig. (2-tailed)	Mean Difference	Standard Error Difference	95% Confidence Interval of the Difference (Lower)	95% Confidence Interval of the Difference (Upper)
Equal Variance not Assumed	3.534	37.116717335823	0.001	15.2	4.30101	6.48625	23.9137
Equal Variance Assumed	3.534	38	0.001	15.2	4.30101	6.49306	23.9069

Loading the sample data

Do the following:

1. Create a new app with a new sheet and open that sheet.
2. Enter the following in the data load editor:
```
Table1:
crosstable LOAD recno() as ID, * inline [
Observation|Comparison
35|2
40|27
12|38
15|31
21|1
14|19
46|1
10|34
28|3
48|1
16|2
30|3
32|2
```

```
48|1
31|2
22|1
12|3
39|29
19|37
25|2 ] (delimiter is '|');
```

In this load script, **recno()** is included because **crosstable** requires three arguments. So, **recno()** simply provides an extra argument, in this case an ID for each row. Without it, **Comparison** sample values would not be loaded.

3. Click to load data.

Creating the Group Statistics table

Do the following:

1. In the data load editor, click to go to app view, and then click the sheet you created before. This opens the sheet view.
2. Click **Edit sheet** to edit the sheet.
3. From **Charts**, add a table, and from **Fields**, add the following expressions as measures:

Example expressions

Label	Expression
N	Count(Value)
Mean	Avg(Value)
Standard Deviation	Stdev(Value)
Standard Error Mean	Sterr(Value)

4. Add Type as a dimension to the table.
5. Click **Sorting** and move Type to the top of the sorting list.

   Result:

   A Group Statistics table for these samples would look like this:

Group statistics

Type	N	Mean	Standard Deviation	Standard Error Mean
Comparison	20	11.95	14.61245	3.2674431
Observation	20	27.15	12.507997	2.7968933

Creating the Two Independent Sample Student's T-test table

Do the following:

1. Click **Edit sheet** to edit the sheet.
2. Add the following expression as a dimension to the table. `=ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1))`

3. From **Charts** add a table with the following expressions as measures:

Example expressions

Label	Expression
conf	if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_conf(Type, Value),TTest_conf(Type, Value, 0))
t	if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_t(Type, Value),TTest_t(Type, Value, 0))
df	if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_df(Type, Value),TTest_df(Type, Value, 0))
Sig. (2-tailed)	if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_sig(Type, Value),TTest_sig(Type, Value, 0))
Mean Difference	TTest_dif(Type, Value)
Standard Error Difference	if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_sterr(Type, Value),TTest_sterr(Type, Value, 0))
95% Confidence Interval of the Difference (Lower)	if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_lower(Type, Value,(1-(95)/100)/2),TTest_lower(Type, Value,(1-(95)/100)/2, 0))
95% Confidence Interval of the Difference (Upper)	if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_upper(Type, Value,(1-(95)/100)/2),TTest_upper(Type, Value,(1-(95)/100)/2, 0))

Result:

Independent sample test

Type	t	df	Sig. (2-tailed)	Mean Difference	Standard Error Difference	95% Confidence Interval of the Difference (Lower)	95% Confidence Interval of the Difference (Upper)
Equal Variance not Assumed	3.534	37.116717335823	0.001	15.2	4.30101	6.48625	23.9137
Equal Variance Assumed	3.534	38	0.001	15.2	4.30101	6.49306	23.9069

Examples of how to use z-test functions

The z-test functions are used to find values associated with z-test statistical analysis for large data samples, usually greater than 30, and where the variance is known.

This section describes how to build visualizations using sample data to find the values of the z-test functions available in Qlik Sense. Please refer to the individual z-test chart function topics for descriptions of syntax and arguments.

Loading the sample data

The sample data used here is the same as that used in the t-test function examples. The sample data size would normally be considered too small for z-test analysis, but is sufficient for the purposes of illustrating the use of the different z-test functions in Qlik Sense.

Do the following:

1. Create a new app with a new sheet and open that sheet.

   If you created an app for the t-test functions, you could use that and create a new sheet for these functions.

2. In the data load editor, enter the following:
```
Table1:
crosstable LOAD recno() as ID, * inline [
Observation|Comparison
35|2
40|27
12|38
15|31
21|1
14|19
46|1
10|34
28|3
48|1
16|2
30|3
32|2
48|1
31|2
22|1
12|3
39|29
19|37
25|2 ] (delimiter is '|');
```
   In this load script, **recno()** is included because **crosstable** requires three arguments. So, **recno()** simply provides an extra argument, in this case an ID for each row. Without it, **Comparison** sample values would not be loaded.

3. Click to load data.

Creating z-test chart function visualizations

Do the following:

1. In the data load editor, click to go to app view, and then click the sheet you created when loading the data.
   The sheet view is opened.

2. Click **Edit sheet** to edit the sheet.
3. From **Charts** add a table, and from **Fields** add Type as a dimension.
4. Add the following expressions to the table as measures.

Example expressions

Label	Expression
ZTest Conf	ZTest_conf(Value)
ZTest Dif	ZTest_dif(Value)
ZTest Sig	ZTest_sig(Value)
ZTest Sterr	ZTest_sterr(Value)
ZTest Z	ZTest_z(Value)

*You might wish to adjust the number formatting of the measures in order to see meaningful values. The table will be easier to read if you set number formatting on most of the measures to **Number>Simple**, instead of **Auto**. But for ZTest Sig, for example, use the number formatting: **Custom**, and then adjust the format pattern to **# ##**.*

Result:

The resulting table for the z-test functions for the sample data will contain the following values:

Results table

Type	ZTest Conf	ZTest Dif	ZTest Sig	ZTest Sterr	ZTest Z
Comparison	6.40	11.95	0.000123	3.27	3.66
Value	5.48	27.15	0.001	2.80	9.71

Creating z-testw chart function visualizations

The z-testw functions are for use when the input data series occurs in weighted two-column format. The expressions require a value for the argument weight. The examples here use the value 2 throughout, but you could use an expression, which would define a value for weight for each observation.

Examples and results:

Using the same sample data and number formatting as for the z-test functions, the resulting table for the z-testw functions will contain the following values:

Results table

Type	ZTestw Conf	ZTestw Dif	ZTestw Sig	ZTestw Sterr	ZTestw Z
Comparison	3.53	2.95	5.27e-005	1.80	3.88
Value	2.97	34.25	0	4.52	20.49

String aggregation functions

This section describes string-related aggregation functions.

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

String aggregation functions in the data load script

Concat

Concat() is used to combine string values. The script function returns the aggregated string concatenation of all values of the expression iterated over a number of records as defined by a **group by** clause.

```
Concat ([ distinct ] expression [, delimiter [, sort-weight]])
```

FirstValue

FirstValue() returns the value that was loaded first from the records defined by the expression, sorted by a **group by** clause.

This function is only available as a script function.

```
FirstValue  (expression)
```

LastValue

LastValue() returns the value that was loaded last from the records defined by the expression, sorted by a **group by** clause.

This function is only available as a script function.

```
LastValue  (expression)
```

MaxString

MaxString() finds string values in the expression and returns the last text value sorted alphabetically over a number of records, as defined by a **group by** clause.

```
MaxString (expression )
```

MinString

MinString() finds string values in the expression and returns the first text value sorted alphabetically over a number of records, as defined by a **group by** clause.

```
MinString (expression )
```

String aggregation functions in charts

The following chart functions are available for aggregating strings in charts.

Concat

Concat() is used to combine string values. The function returns the aggregated string concatenation of all the values of the expression evaluated over each dimension.

```
Concat - chart function({[SetExpression] [DISTINCT] [TOTAL [<fld{, fld}>]]
string[, delimiter[, sort_weight]]])
```

MaxString

MaxString() finds string values in the expression or field and returns the last text value in alphabetical sort order.

```
MaxString - chart function({[SetExpression] [TOTAL [<fld{, fld}>]]} expr)
```

MinString

MinString() finds string values in the expression or field and returns the first text value in alphabetical sort order.

```
MinString - chart function({[SetExpression] [TOTAL [<fld {, fld}>]]} expr)
```

Concat

Concat() is used to combine string values. The script function returns the aggregated string concatenation of all values of the expression iterated over a number of records as defined by a **group by** clause.

Syntax:

```
Concat ([ distinct ] string [, delimiter [, sort-weight]])
```

Return data type: string

Arguments:

The expression or field containing the string to be processed.

Arguments

Argument	Description
string	The expression or field containing the string to be processed.
delimiter	Each value may be separated by the string found in delimiter.
sort-weight	The order of concatenation may be determined by the value of the dimension **sort-weight**, if present, with the string corresponding to the lowest value appearing first in the concatenation.
distinct	If the word **distinct** occurs before the expression, all duplicates are disregarded.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Examples and results

Example	Result	Results once added to a sheet

`TeamData:`
`LOAD * inline [`
`SalesGroup|Team|Date|Amount`
`East|Gamma|01/05/2013|20000`
`East|Gamma|02/05/2013|20000`
`West|Zeta|01/06/2013|19000`
`East|Alpha|01/07/2013|25000`
`East|Delta|01/08/2013|14000`
`West|Epsilon|01/09/2013|17000`
`West|Eta|01/10/2013|14000`
`East|Beta|01/11/2013|20000`
`West|Theta|01/12/2013|23000`
`] (delimiter is '|');`
 	
`Concat1:`
`LOAD SalesGroup,Concat(Team) as TeamConcat1`
`Resident TeamData Group By SalesGroup;`
	SalesGroup
	
	East
	
	West
		TeamConcat1
		
		AlphaBetaDeltaGammaGamma
		
		EpsilonEtaThetaZeta

Given that the **TeamData** table is loaded as in the previous example: `LOAD SalesGroup,Concat(distinct Team,'-') as TeamConcat2 Resident TeamData Group By SalesGroup;`	SalesGroup East West	TeamConcat2 Alpha-Beta-Delta-Gamma Epsilon-Eta-Theta-Zeta
Given that the **TeamData** table is loaded as in the previous example. Because the argument for **sort-weight** is added, the results are ordered by the value of the dimension Amount: `LOAD SalesGroup,Concat(distinct Team,'-',Amount) as TeamConcat2 Resident TeamData Group By SalesGroup;`	SalesGroup East West	TeamConcat2 Delta-Beta-Gamma-Alpha Eta-Epsilon-Zeta-Theta

Concat - chart function

Concat() is used to combine string values. The function returns the aggregated string concatenation of all the values of the expression evaluated over each dimension.

Syntax:

```
Concat({[SetExpression] [DISTINCT] [TOTAL [<fld{, fld}>]]} string[, delimiter [, sort_weight]])
```

Return data type: string

Arguments:

Arguments

Argument	Description
string	The expression or field containing the string to be processed.
delimiter	Each value may be separated by the string found in delimiter.
sort-weight	The order of concatenation may be determined by the value of the dimension **sort-weight**, if present, with the string corresponding to the lowest value appearing first in the concatenation.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the word **DISTINCT** occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Results table

SalesGroup	Amount	Concat(Team)	Concat(TOTAL <SalesGroup> Team)
East	25000	Alpha	AlphaBetaDeltaGammaGamma
East	20000	BetaGammaGamma	AlphaBetaDeltaGammaGamma
East	14000	Delta	AlphaBetaDeltaGammaGamma
West	17000	Epsilon	EpsilonEtaThetaZeta
West	14000	Eta	EpsilonEtaThetaZeta
West	23000	Theta	EpsilonEtaThetaZeta
West	19000	Zeta	EpsilonEtaThetaZeta

Function examples

Example	Result
`Concat(Team)`	The table is constructed from the dimensions SalesGroup and Amount, and variations on the measure Concat(Team). Ignoring the Totals result, note that even though there is data for eight values of Team spread across two values of SalesGroup, the only result of the measure Concat(Team) that concatenates more than one Team string value in the table is the row containing the dimension Amount 20000, which gives the result BetaGammaGamma. This is because there are three values for the Amount 20000 in the input data. All other results remain unconcatenated when the measure is spanned across the dimensions because there is only one value of Team for each combination of SalesGroup and Amount.
`Concat (DISTINCT Team,', ')`	Beta, Gamma. because the DISTINCT qualifier means the duplicate Gamma result is disregarded. Also, the delimiter argument is defined as a comma followed by a space.
`Concat (TOTAL <SalesGroup> Team)`	All the string values for all values of Team are concatenated if the TOTAL qualifier is used. With the field selection <SalesGroup> specified, this divides the results into the two values of the dimension SalesGroup. For the SalesGroupEast, the results are AlphaBetaDeltaGammaGamma. For the SalesGroupWest, the results are EpsilonEtaThetaZeta.
`Concat (TOTAL <SalesGroup> Team,';', Amount)`	By adding the argument for **sort-weight**: Amount, the results are ordered by the value of the dimension Amount. The results becomes DeltaBetaGammaGammaAlpha and EtaEpsilonZEtaTheta.

Data used in example:

```
TeamData:
LOAD * inline [
SalesGroup|Team|Date|Amount
East|Gamma|01/05/2013|20000
East|Gamma|02/05/2013|20000
West|Zeta|01/06/2013|19000
East|Alpha|01/07/2013|25000
East|Delta|01/08/2013|14000
West|Epsilon|01/09/2013|17000
West|Eta|01/10/2013|14000
East|Beta|01/11/2013|20000
West|Theta|01/12/2013|23000
] (delimiter is '|');
```

FirstValue

FirstValue() returns the value that was loaded first from the records defined by the expression, sorted by a **group by** clause.

This function is only available as a script function.

Syntax:

```
FirstValue ( expr)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.

Limitations:

If no text value is found, NULL is returned.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Resulting data

Example	Result	Results on a sheet

`TeamData:`
`LOAD * inline [`
`SalesGroup|Team|Date|Amount`
`East|Gamma|01/05/2013|20000`
`East|Gamma|02/05/2013|20000`
`West|Zeta|01/06/2013|19000`
`East|Alpha|01/07/2013|25000`
`East|Delta|01/08/2013|14000`
`West|Epsilon|01/09/2013|17000`
`West|Eta|01/10/2013|14000`
`East|Beta|01/11/2013|20000`
`West|Theta|01/12/2013|23000`
`] (delimiter is '|');`
 	
`FirstValue1:`
`LOAD SalesGroup,FirstValue(Team) as FirstTeamLoaded`
`Resident TeamData Group By SalesGroup;`
	SalesGroup
	
	East
	
	West
		FirstTeamLoaded
		
		Gamma
		
		Zeta

LastValue

LastValue() returns the value that was loaded last from the records defined by the expression, sorted by a **group by** clause.

This function is only available as a script function.

Syntax:

```
LastValue ( expr )
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.

Limitations:

If no text value is found, NULL is returned.

Examples and results:

Add the example script to your app and run it. Then add, at least, the fields listed in the results column to a sheet in our app to see the result.

To get the same look as in the result column below, in the properties panel, under Sorting, switch from Auto to Custom, then deselect numerical and alphabetical sorting.

Example	Result	Result with custom sorting

`TeamData:`
`LOAD * inline [`
`SalesGroup|Team|Date|Amount`
`East|Gamma|01/05/2013|20000`
`East|Gamma|02/05/2013|20000`
`West|Zeta|01/06/2013|19000`
`East|Alpha|01/07/2013|25000`
`East|Delta|01/08/2013|14000`
`West|Epsilon|01/09/2013|17000`
`West|Eta|01/10/2013|14000`
`East|Beta|01/11/2013|20000`
`West|Theta|01/12/2013|23000`
`] (delimiter is '|');`
 	
`LastValue1:`
`LOAD SalesGroup,LastValue(Team) as LastTeamLoaded Resident TeamData Group By SalesGroup;`
	SalesGroup
	
	East
	
	West
		LastTeamLoaded
		
		Beta
		
		Theta

MaxString

MaxString() finds string values in the expression and returns the last text value sorted alphabetically over a number of records, as defined by a **group by** clause.

Syntax:

```
MaxString ( expr )
```

Return data type: dual

Arguments:

Argument	Description
expr	The expression or field containing the data to be measured.

Limitations:

If no text value is found, NULL is returned.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Example	Result	

`TeamData:`
`LOAD * inline [`
`SalesGroup|Team|Date|Amount`
`East|Gamma|01/05/2013|20000`
`East|Gamma|02/05/2013|20000`
`West|Zeta|01/06/2013|19000`
`East|Alpha|01/07/2013|25000`
`East|Delta|01/08/2013|14000`
`West|Epsilon|01/09/2013|17000`
`West|Eta|01/10/2013|14000`
`East|Beta|01/11/2013|20000`
`West|Theta|01/12/2013|23000`
`] (delimiter is '|');`
 	
`Concat1:`
`LOAD SalesGroup,MaxString(Team) as MaxString1 Resident TeamData Group By SalesGroup;`
	SalesGroup
	
	East
	
	West
		MaxString1
		
		Gamma
		
		Zeta

Given that the **TeamData** table is loaded as in the previous example, and your data load script has the SET statement:
`SET DateFormat='DD/MM/YYYY';':`
 	
`LOAD SalesGroup,MaxString(Date) as MaxString2 Resident TeamData Group By SalesGroup;`
	SalesGroup
	
	East
	
	West
		MaxString2
		
		01/11/2013
		
		01/12/2013

MaxString - chart function

MaxString() finds string values in the expression or field and returns the last text value in alphabetical sort order.

Syntax:

```
MaxString({[SetExpression] [TOTAL [<fld{, fld}>]]} expr)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Limitations:

If the expression contains no values with a string representation NULL is returned.

Examples and results:

Results table

SalesGroup	Amount	MaxString(Team)	MaxString(Date)
East	14000	Delta	2013/08/01
East	20000	Gamma	2013/11/01
East	25000	Alpha	2013/07/01
West	14000	Eta	2013/10/01
West	17000	Epsilon	2013/09/01
West	19000	Zeta	2013/06/01
West	23000	Theta	2013/12/01

Function examples

Example	Result
`MaxString (Team)`	There are three values of 20000 for the dimension Amount: two of Gamma (on different dates), and one of Beta. The result of the measure MaxString (Team) is therefore Gamma, because this is the highest value in the sorted strings.

Example	Result
`MaxString (Date)`	2013/11/01 is the greatest Date value of the three associated with the dimension Amount. This assumes your script has the SET statement `SET DateFormat='YYYY-MM-DD';'`

Data used in example:

```
TeamData:
LOAD * inline [
SalesGroup|Team|Date|Amount
East|Gamma|01/05/2013|20000
East|Gamma|02/05/2013|20000
West|Zeta|01/06/2013|19000
East|Alpha|01/07/2013|25000
East|Delta|01/08/2013|14000
West|Epsilon|01/09/2013|17000
West|Eta|01/10/2013|14000
East|Beta|01/11/2013|20000
West|Theta|01/12/2013|23000
] (delimiter is '|');
```

MinString

MinString() finds string values in the expression and returns the first text value sorted alphabetically over a number of records, as defined by a **group by** clause.

Syntax:

```
MinString ( expr )
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.

Limitations:

If no text value is found, NULL is returned.

Examples and results:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

Resulting data

Example	Result	

`TeamData:`
`LOAD * inline [`
`SalesGroup|Team|Date|Amount`
`East|Gamma|01/05/2013|20000`
`East|Gamma|02/05/2013|20000`
`West|Zeta|01/06/2013|19000`
`East|Alpha|01/07/2013|25000`
`East|Delta|01/08/2013|14000`
`West|Epsilon|01/09/2013|17000`
`West|Eta|01/10/2013|14000`
`East|Beta|01/11/2013|20000`
`West|Theta|01/12/2013|23000`
`] (delimiter is '|');`
 	
`Concat1:`
`LOAD SalesGroup,MinString(Team) as MinString1 Resident TeamData`
`Group By SalesGroup;`
	SalesGroup
	
	East
	
	West
		MinString1
		
		Alpha
		
		Epsilon

Given that the **TeamData** table is loaded as in the previous example, and your data load script has the SET statement:
`SET DateFormat='DD/MM/YYYY';':`
 	
`LOAD SalesGroup,MinString(Date) as MinString2 Resident TeamData`
`Group By SalesGroup;`
	SalesGroup
	
	East
	
	West
		MinString2
		
		01/05/2013
		
		01/06/2013

MinString - chart function

MinString() finds string values in the expression or field and returns the first text value in alphabetical sort order.

Syntax:

```
MinString({[SetExpression] [TOTAL [<fld {, fld}>]]} expr)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.

Argument	Description
TOTAL	If the word **TOTAL** occurs before the function arguments, the calculation is made over all possible values given the current selections, and not just those that pertain to the current dimensional value, that is, it disregards the chart dimensions. By using **TOTAL [<fld {.fld}>]**, where the **TOTAL** qualifier is followed by a list of one or more field names as a subset of the chart dimension variables, you create a subset of the total possible values.

Examples and results:

Sample data

SalesGroup	Amount	MinString(Team)	MinString(Date)
East	14000	Delta	2013/08/01
East	20000	Beta	2013/05/01
East	25000	Alpha	2013/07/01
West	14000	Eta	2013/10/01
West	17000	Epsilon	2013/09/01
West	19000	Zeta	2013/06/01
West	23000	Theta	2013/12/01

Function examples

Examples	Results
`MinString (Team)`	There are three values of 20000 for the dimension Amount: two of Gamma (on different dates), and one of Beta. The result of the measure MinString (Team) is therefore Beta, because this is the first value in the sorted strings.
`MinString (Date)`	2013/11/01 is the earliest Date value of the three associated with the dimension Amount. This assumes your script has the SET statement `SET DateFormat='YYYY-MM-DD';'`

Data used in example:

```
TeamData:
LOAD * inline [
SalesGroup|Team|Date|Amount
East|Gamma|01/05/2013|20000
East|Gamma|02/05/2013|20000
West|Zeta|01/06/2013|19000
East|Alpha|01/07/2013|25000
East|Delta|01/08/2013|14000
West|Epsilon|01/09/2013|17000
West|Eta|01/10/2013|14000
East|Beta|01/11/2013|20000
West|Theta|01/12/2013|23000
] (delimiter is '|');
```

Synthetic dimension functions

A synthetic dimension is created in the app from values generated from the synthetic dimension functions and not directly from fields in the data model. When values generated by a synthetic dimension function are used in a chart as a calculated dimension, this creates a synthetic dimension. Synthetic dimensions allow you to create, for example, charts with dimensions with values arising from your data, that is, dynamic dimensions.

> *Synthetic dimensions are not affected by selections.*

The following synthetic dimension functions can be used in charts.

ValueList

ValueList() returns a set of listed values, which, when used in a calculated dimension, will form a synthetic dimension.

```
ValueList - chart function (v1 {, Expression})
```

ValueLoop

ValueLoop() returns a set of iterated values which, when used in a calculated dimension, will form a synthetic dimension.

```
ValueLoop - chart function(from [, to [, step ]])
```

ValueList - chart function

ValueList() returns a set of listed values, which, when used in a calculated dimension, will form a synthetic dimension.

> *In charts with a synthetic dimension created with the **ValueList** function it is possible to reference the dimension value corresponding to a specific expression cell by restating the **ValueList** function with the same parameters in the chart expression. The function may of course be used anywhere in the layout, but apart from when used for synthetic dimensions it will only be meaningful inside an aggregation function.*

> *Synthetic dimensions are not affected by selections.*

Syntax:

```
ValueList(v1 {,...})
```

Return data type: dual

Arguments:

Arguments

Argument	Description
v1	Static value (usually a string, but can be a number).
{,...}	Optional list of static values.

Examples and results:

Function examples

Example	Result
`ValueList ('Number of Orders', 'Average Order Size', 'Total Amount')`	When used to create a dimension in a table, for example, this results in the three string values as row labels in the table. These can then be referenced in an expression.
`=IF( ValueList ('Number of Orders', 'Average Order Size', 'Total Amount') = 'Number of Orders', count (SaleID), IF( ValueList ('Number of Orders', 'Average Order Size', 'Total Amount') = 'Average Order Size', avg (Amount), sum (Amount) ))`	This expression takes the values from the created dimension and references them in a nested IF statement as input to three aggregation functions:

ValueList()

Created dimension	Year	Added expression
		522.00
Number of Orders	2012	5.00
Number of Orders	2013	7.00
Average Order Size	2012	13.20
Average Order Size	2013	15.43
Total Amount	2012	66.00
Total Amount	2013	108.00

Data used in examples:

```
SalesPeople:
LOAD * INLINE [
SaleID|SalesPerson|Amount|Year
1|1|12|2013
2|1|23|2013
3|1|17|2013
4|2|9|2013
5|2|14|2013
6|2|29|2013
```

```
7|2|4|2013
8|1|15|2012
9|1|16|2012
10|2|11|2012
11|2|17|2012
12|2|7|2012
] (delimiter is '|');
```

ValueLoop - chart function

ValueLoop() returns a set of iterated values which, when used in a calculated dimension, will form a synthetic dimension.
The values generated will start with the **from** value and end with the **to** value including intermediate values in increments of step.

> *In charts with a synthetic dimension created with the **ValueLoop** function it is possible to reference the dimension value corresponding to a specific expression cell by restating the **ValueLoop** function with the same parameters in the chart expression. The function may of course be used anywhere in the layout, but apart from when used for synthetic dimensions it will only be meaningful inside an aggregation function.*

> *Synthetic dimensions are not affected by selections.*

Syntax:

```
ValueLoop(from [, to [, step ]])
```

Return data type: dual

Arguments:

Arguments

Arguments	Description
from	Start value in the set of values to be generated.
to	End value in the set of values to be generated.
step	Size of increment between values.

Examples and results:

Function examples

Example	Result
`ValueLoop (1, 10)`	This creates a dimension in a table, for example, that can be used for purposes such as numbered labeling. The example here results in values numbered 1 to 10. These values can then be referenced in an expression.
`ValueLoop (2, 10,2)`	This example results in values numbered 2, 4, 6, 8, and 10 because the argument step has a value of 2.

Nested aggregations

You may come across situations where you need to apply an aggregation to the result of another aggregation. This is referred to as nesting aggregations.

You cannot nest aggregations in most chart expressions. You can, however, nest aggregations if you use the **TOTAL** qualifier in the inner aggregation function.

No more than 100 levels of nesting is allowed.

Nested aggregations with the TOTAL qualifier

Example:

You want to calculate the sum of the field **Sales**, but only include transactions with an **OrderDate** equal to the last year. The last year can be obtained via the aggregation function `Max(TOTAL Year(OrderDate))`.

The following aggregation would return the desired result:

```
Sum(If(Year(OrderDate)=Max(TOTAL Year(OrderDate)), Sales))
```

Qlik Sense requires the inclusion of the **TOTAL** qualifier this type of nesting. It is necessary for the desired comparison. This type of nesting need is quite common and is a good practice.

See also:

Aggr - chart function (page 459)

5.3 Aggr - chart function

Aggr() returns an array of values for the expression calculated over the stated dimension or dimensions. For example, the maximum value of sales, per customer, per region.

The **Aggr** function is used for nested aggregations, in which its first parameter (the inner aggregation) is calculated once per dimensional value. The dimensions are specified in the second parameter (and subsequent parameters).

In addition, the **Aggr** function should be enclosed in an outer aggregation function, using the array of results from the **Aggr** function as input to the aggregation in which it is nested.

Syntax:

```
Aggr({SetExpression}[DISTINCT] [NODISTINCT ] expr, StructuredParameter{,
StructuredParameter})
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	An expression consisting of an aggregation function. By default, the aggregation function will aggregate over the set of possible records defined by the selection.
StructuredParameter	StructuredParameter consists of a dimension and optionally, sorting criteria in the format: `(Dimension(Sort-type, Ordering))` The dimension is a single field and cannot be an expression. The dimension is used to determine the array of values the Aggr expression is calculated for. If sorting criteria are included, the array of values created by the Aggr function, calculated for the dimension, is sorted. This is important when the sort order affects the result of the expression the Aggr function is enclosed in. For details of how to use sorting criteria, see Adding sorting criteria to the dimension in the structured parameter.
SetExpression	By default, the aggregation function will aggregate over the set of possible records defined by the selection. An alternative set of records can be defined by a set analysis expression.
DISTINCT	If the expression argument is preceded by the **distinct** qualifier or if no qualifier is used at all, each distinct combination of dimension values will generate only one return value. This is the normal way aggregations are made – each distinct combination of dimension values will render one line in the chart.
NODISTINCT	If the expression argument is preceded by the **nodistinct** qualifier, each combination of dimension values may generate more than one return value, depending on underlying data structure. If there is only one dimension, the **aggr** function will return an array with the same number of elements as there are rows in the source data.

Basic aggregation functions, such as **Sum**, **Min**, and **Avg**, return a single numerical value, whereas the Aggr() function can be compared to creating a temporary staged result set (a virtual table), over which another aggregation can be made. For example, by computing an average sales value by summing the sales by customer in an **Aggr()** statement, and then calculating the average of the summed results: **Avg(TOTAL Aggr (Sum(Sales),Customer))**.

Use the Aggr() function in calculated dimensions if you want to create nested chart aggregations on multiple levels.

Limitations:

Each dimension in an Aggr() function must be a single field, and cannot be an expression (calculated dimension).

Adding sorting criteria to the dimension in the structured parameter

In its basic form, the argument StructuredParameter in the Aggr function syntax is a single dimension. The expression: Aggr(Sum(Sales, Month)) finds the total value of sales for each month. However, when enclosed in another aggregation function, there can be unexpected results unless sorting criteria are used. This is because some dimensions can be sorted numerically or alphabetically, and so on.

In the StructuredParameter argument in the Aggr function, you can specify sorting criteria on the dimension in your expression. This way, you impose a sort order on the virtual table that is produced by the Aggr function.

The argument StructuredParameter has the following syntax:

```
(FieldName, (Sort-type, Ordering))
```

Structured parameters can be nested:

```
(FieldName, (FieldName2, (Sort-type, Ordering)))
```

Sort-type can be: NUMERIC, TEXT, FREQUENCY, or LOAD_ORDER.

The Ordering types associated with each Sort-type are as follows:

Allowed ordering types

Sort-type	Allowed Ordering types
NUMERIC	ASCENDING, DESCENDING, or REVERSE
TEXT	ASCENDING, A2Z, DESCENDING, REVERSE, or Z2A
FREQUENCY	DESCENDING, REVERSE or ASCENDING
LOAD_ORDER	ASCENDING, ORIGINAL, DESCENDING, or REVERSE

The ordering types REVERSE and DESCENDING are equivalent.

For Sort-type TEXT, the ordering types ASCENDING and A2Z are equivalent, and DESCENDING, REVERSE, and Z2A are equivalent.

For Sort-type LOAD_ORDER, the ordering types ASCENDING and ORIGINAL are equivalent.

Examples: Chart expressions using Aggr

Examples - chart expressions

Chart expression example 1

Load script

Load the following data as an inline load in the data load editor to create the chart expression example below.

```
ProductData:

LOAD * inline [
Customer|Product|UnitSales|UnitPrice
Astrida|AA|4|16
Astrida|AA|10|15
Astrida|BB|9|9
Betacab|BB|5|10
Betacab|CC|2|20
Betacab|DD|25|25
Canutility|AA|8|15
Canutility|CC|0|19
] (delimiter is '|');
```

Chart expression

Create a KPI visualization in a Qlik Sense sheet. Add the following expression to the KPI, as a measure:

```
Avg(Aggr(Sum(UnitSales*UnitPrice), Customer))
```

Result

376.7

Explanation

The expression `Aggr(Sum(UnitSales*UnitPrice), Customer)` finds the total value of sales by **Customer**, and returns an array of values: 295, 715, and 120 for the three **Customer** values.

Effectively, we have built a temporary list of values without having to create an explicit table or column containing those values.

These values are used as input to the **Avg()** function to find the average value of sales, 376.7.

Chart expression example 2

Load script

Load the following data as an inline load in the data load editor to create the chart expression example below.

```
ProductData:

LOAD * inline [
Customer|Product|UnitSales|UnitPrice
Astrida|AA|4|16
Astrida|AA|10|15
```

```
Astrida|BB|10|15
Astrida|BB|9|9
Betacab|BB|5|10
Betacab|BB|7|12
Betacab|CC|2|22
Betacab|CC|4|20
Betacab|DD|25|25
Canutility|AA|8|15
Canutility|AA|5|11
Canutility|CC|0|19
] (delimiter is '|');
```

Chart expression

Create a table visualization in a Qlik Sense sheet with **Customer**, **Product**, **UnitPrice**, and **UnitSales** as dimensions. Add the following expression to the table, as a measure:

```
Aggr(NODISTINCT Max(UnitPrice), Customer, Product)
```

Result

Customer	Product	UnitPrice	UnitSales	Aggr(NODISTINCT Max(UnitPrice), Customer, Product)
Astrida	AA	15	10	16
Astrida	AA	16	4	16
Astrida	BB	9	9	15
Astrida	BB	15	10	15
Betacab	BB	10	5	12
Betacab	BB	12	7	12
Betacab	CC	20	4	22
Betacab	CC	22	2	22
Betacab	DD	25	25	25
Canutility	AA	11	5	15
Canutility	AA	15	8	15
Canutility	CC	19	0	19

Explanation

An array of values: 16, 16, 15, 15, 12, 12, 22, 22, 25, 15, 15, and 19. The **nodistinct** qualifier means that the array contains one element for each row in the source data: each is the maximum **UnitPrice** for each **Customer** and **Product**.

Chart expression example 3

Load script

Load the following data as an inline load in the data load editor to create the chart expression example below.

```
Set vNumberOfOrders = 1000;

OrderLines:
Load
       RowNo() as OrderLineID,
       OrderID,
       OrderDate,
       Round((Year(OrderDate)-2005)*1000*Rand()*Rand()*Rand1) as Sales
       While Rand()<=0.5 or IterNo()=1;
Load * Where OrderDate<=Today();
Load
       Rand() as Rand1,
       Date(MakeDate(2013)+Floor((365*4+1)*Rand())) as OrderDate,
       RecNo() as OrderID
       Autogenerate vNumberOfOrders;

Calendar:
Load distinct
       Year(OrderDate) as Year,
       Month(OrderDate) as Month,
       OrderDate
       Resident OrderLines;
```

Chart expressions

Create a table visualization in a Qlik Sense sheet with **Year** and **Month** as dimensions. Add the following expressions to the table as measures:

- `Sum(Sales)`
- `Sum(Aggr( Rangesum(Above(Sum(Sales),0,12)), (Year, (Numeric, Ascending)), (Month, (Numeric, Ascending)) ))` labeled as Structured Aggr() in the table.

Result

Year	Month	Sum(Sales)	Structured Aggr()
2013	Jan	53495	53495
2013	Feb	48580	102075
2013	Mar	25651	127726
2013	Apr	36585	164311
2013	May	61211	225522
2013	Jun	23689	249211

Year	Month	Sum(Sales)	Structured Aggr()
2013	Jul	42311	291522
2013	Aug	41913	333435
2013	Sep	28886	362361
2013	Oct	25977	388298
2013	Nov	44455	432753
2013	Dec	64144	496897
2014	Jan	67775	67775

Explanation

This example displays the aggregated values over a twelve month period for each year in chronological ascending order, hence the structured parameters (Numeric, Ascending) part of the **Aggr()** expression. Two specific dimensions are required as structured parameters: **Year** and **Month**, sorted (1) **Year** (numeric) and (2) **Month** (numeric). These two dimensions must be used in the table or chart visualization. This is necessary for the dimension list of the **Aggr()** function to correspond with the dimensions of the object used in the visualization.

You can compare the difference between these measures in a table or in separate line charts:

- `Sum(Aggr( Rangesum(Above(Sum(Sales),0,12)), (Year), (Month) ))`
- `Sum(Aggr( Rangesum(Above(Sum(Sales),0,12)), (Year, (Numeric, Ascending)), (Month, (Numeric, Ascending)) ))`

It should be clear to see that only the latter expression performs the desired accumulation of aggregated values.

See also:

Basic aggregation functions (page 266)

5.4 Color functions

These functions can be used in expressions associated with setting and evaluating the color properties of chart objects, as well as in data load scripts.

*Qlik Sense supports the color functions **Color()**, **qliktechblue**, and **qliktechgray** for backwards compatibility reasons, but use of them is not recommended.*

ARGB

ARGB() is used in expressions to set or evaluate the color properties of a chart object, where the color is defined by a red component **r**, a green component **g**, and a blue component **b**, with an alpha factor (opacity) of **alpha**.

```
ARGB(alpha, r, g, b)
```

HSL

HSL() is used in expressions to set or evaluate the color properties of a chart object, where the color is defined by values of **hue**, **saturation**, and **luminosity** between 0 and 1.

```
HSL (hue, saturation, luminosity)
```

RGB

RGB() returns an integer corresponding to the color code of the color defined by the three parameters: the red component r, the green component g, and the blue component b. These components must have integer values between 0 and 255. The function can be used in expressions to set or evaluate the color properties of a chart object.

```
RGB (r, g, b)
```

Colormix1

Colormix1() is used in expressions to return an ARGB color representation from a two color gradient, based on a value between 0 and 1.

```
Colormix1 (Value , ColorZero , ColorOne)
```

Value is a real number between 0 and 1.

- If Value = 0 ColorZero is returned.
- If Value = 1 ColorOne is returned.
- If 0 < Value< 1 the appropriate intermediate shading is returned.

ColorZero is a valid RGB color representation for the color to be associated with the low end of the interval.

ColorOne is a valid RGB color representation for the color to be associated with the high end of the interval.

Example:

```
Colormix1(0.5, red(), blue())
```

returns:

```
ARGB(255,64,0,64) (purple)
```

Colormix2

Colormix2() is used in expressions to return an ARGB color representation from a two color gradient, based on a value between -1 and 1, with the possibility to specify an intermediate color for the center (0) position.

```
Colormix2 (Value ,ColorMinusOne , ColorOne[ , ColorZero])
```

Value is a real number between -1 and 1.

- If Value = -1 the first color is returned.
- If Value = 1 the second color is returned.
- If -1 < Value< 1 the appropriate color mix is returned.

ColorMinusOne is a valid RGB color representation for the color to be associated with the low end of the interval.

ColorOne is a valid RGB color representation for the color to be associated with the high end of the interval.

ColorZero is an optional valid RGB color representation for the color to be associated with the center of the interval.

SysColor
SysColor() returns the ARGB color representation for the Windows system color nr, where nr corresponds to the parameter to the Windows API function **GetSysColor(nr)**.

```
SysColor (nr)
```

ColorMapHue
ColorMapHue() returns an ARGB value of a color from a colormap that varies the hue component of the HSV color model. The colormap starts with red, passes through yellow, green, cyan, blue, magenta, and returns to red. x must be specified as a value between 0 and 1.

```
ColorMapHue (x)
```

ColorMapJet
ColorMapJet() returns an ARGB value of a color from a colormap that starts with blue, passes through cyan, yellow and orange, and returns to red. x must be specified as a value between 0 and 1.

```
ColorMapJet (x)
```

Pre-defined color functions

The following functions can be used in expressions for pre-defined colors. Each function returns an RGB color representation.

Optionally a parameter for alpha factor can be given, in which case an ARGB color representation is returned. An alpha factor of 0 corresponds to full transparency, and an alpha factor of 255 corresponds to full opacity. If a value for alpha is not entered, it is assumed to be 255.

Pre-defined color functions

Color function	RGB value
black ([alpha])	(0,0,0)
blue([alpha])	(0,0,128)
brown([alpha])	(128,128,0)
cyan([alpha])	(0,128,128)
darkgray([alpha])	(128,128,128)

green([alpha])	(0,128,0)
lightblue([alpha])	(0,0,255)
lightcyan([alpha])	(0,255,255)
lightgray([alpha])	(192,192,192)
lightgreen([alpha])	(0,255,0)
lightmagenta([alpha])	(255,0,255)
lightred([alpha])	(255,0,0)
magenta([alpha])	(128,0,128)
red([alpha])	(128,0,0)
white([alpha])	(255,255,255)
yellow([alpha])	(255,255,0)

Examples and results:

Examples and results

Examples	Results
`Blue()`	RGB(0,0,128)
`Blue(128)`	ARGB(128,0,0,128)

ARGB

ARGB() is used in expressions to set or evaluate the color properties of a chart object, where the color is defined by a red component **r**, a green component **g**, and a blue component **b**, with an alpha factor (opacity) of **alpha**.

Syntax:

```
ARGB(alpha, r, g, b)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
alpha	Transparency value in the range 0 - 255. 0 corresponds to full transparency and 255 corresponds to full opacity.
r, g, b	Red, green, and blue component values. A color component of 0 corresponds to no contribution and one of 255 to full contribution.

All arguments must be expressions that resolve to integers in the range 0 to 255.

If interpreting the numeric component and formatting it in hexadecimal notation, the values of the color components are easier to see. For example, light green has the number 4 278 255 360, which in hexadecimal notation is FF00FF00. The first two positions 'FF' (255) denote the **alpha** channel. The next two positions '00' denote the amount of **red**, the next two positions 'FF' denote the amount of **green**, and the final two positions '00' denote the amount of **blue**.

RGB

RGB() returns an integer corresponding to the color code of the color defined by the three parameters: the red component r, the green component g, and the blue component b. These components must have integer values between 0 and 255. The function can be used in expressions to set or evaluate the color properties of a chart object.

Syntax:

```
RGB (r, g, b)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
r, g, b	Red, green, and blue component values. A color component of 0 corresponds to no contribution and one of 255 to full contribution.

All arguments must be expressions that resolve to integers in the range 0 to 255.

If interpreting the numeric component and formatting it in hexadecimal notation, the values of the color components are easier to see. For example, light green has the number 4 278 255 360, which in hexadecimal notation is FF00FF00. The first two positions 'FF' (255) denote the **alpha** channel. In the functions **RGB** and **HSL**, this is always 'FF' (opaque). The next two positions '00' denote the amount of **red**, the next two positions 'FF' denote the amount of **green**, and the final two positions '00' denote the amount of **blue**.

Example: Chart expression
This example applies a custom color to a chart:

Data used in this example:

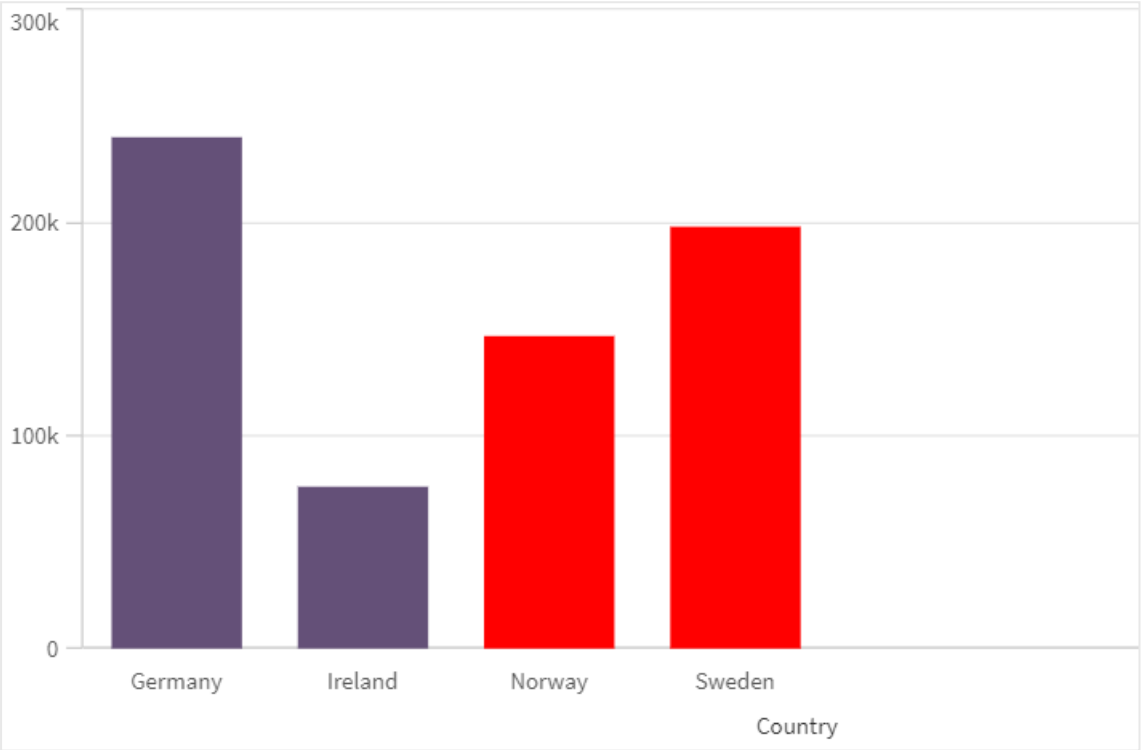
```
ProductSales:
Load * Inline
[Country,Sales,Budget
Sweden,100000,50000
Germany, 125000, 175000
Norway, 74850, 68500
```

```
Ireland, 45000, 48000
Sweden,98000,50000
Germany, 115000, 175000
Norway, 71850, 68500
Ireland, 31000, 48000
] (delimiter is ',');
```

Enter the following expression in the **Colors and legend** properties panel:

```
If (Sum(Sales)>Sum(Budget),RGB(255,0,0),RGB(100,80,120))
```

Result:



Example: Load script

The following example displays the equivalent RGB values for values in hex format:

```
Load
Text(R & G & B) as Text,
RGB(R,G,B)      as Color;
Load
Num#(R,'(HEX)') as R,
Num#(G,'(HEX)') as G,
Num#(B,'(HEX)') as B
Inline
[R,G,B
01,02,03
AA,BB,CC];
```

Result:

Text	Color
010203	RGB(1,2,3)
AABBCC	RGB(170,187,204)

HSL

HSL() is used in expressions to set or evaluate the color properties of a chart object, where the color is defined by values of **hue**, **saturation**, and **luminosity** between 0 and 1.

Syntax:

```
HSL (hue, saturation, luminosity)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
hue, saturation, luminosity	hue, saturation, and luminosity component values ranging between 0 and 1.

All arguments must be expressions that resolve to integers in the range 0 to 1.

If interpreting the numeric component and formatting it in hexadecimal notation, the RGB values of the color components are easier to see. For example, light green has the number 4 278 255 360, which in hexadecimal notation is FF00FF00 and RGB (0,255,0). This is equivalent to HSL (80/240, 240/240, 120/240) - a HSL value of (0.33, 1, 0.5).

5.5 Conditional functions

The conditional functions all evaluate a condition and then return different answers depending on the condition value. The functions can be used in the data load script and in chart expressions.

Conditional functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

alt

The **alt** function returns the first of the parameters that has a valid number representation. If no such match is found, the last parameter will be returned. Any number of parameters can be used.

```
alt (expr1[ , expr2 , expr3 , ...] , else)
```

class

The **class** function assigns the first parameter to a class interval. The result is a dual value with a<=x<b as the textual value, where a and b are the upper and lower limits of the bin, and the lower bound as numeric value.

```
class (expression, interval [ , label [ , offset ]])
```

coalesce

The **coalesce** function returns the first of the parameters that has a valid non-NULL representation. Any number of parameters can be used.

```
coalesce(expr1[ , expr2 , expr3 , ...])
```

if

The **if** function returns a value depending on whether the condition provided with the function evaluates as True or False.

```
if (condition , then , else)
```

match

The **match** function compares the first parameter with all the following ones and returns the numeric location of the expressions that match. The comparison is case sensitive.

```
match ( str, expr1 [ , expr2,...exprN ])
```

mixmatch

The **mixmatch** function compares the first parameter with all the following ones and returns the numeric location of the expressions that match. The comparison is case insensitive.

```
mixmatch ( str, expr1 [ , expr2,...exprN ])
```

pick

The pick function returns the *n*:th expression in the list.

```
pick (n, expr1[ , expr2,...exprN])
```

wildmatch

The **wildmatch** function compares the first parameter with all the following ones and returns the number of the expression that matches. It permits the use of wildcard characters (***** and **?**) in the comparison strings. ***** matches any sequence of characters. **?** matches any single character. The comparison is case insensitive.

```
wildmatch ( str, expr1 [ , expr2,...exprN ])
```

alt

The **alt** function returns the first of the parameters that has a valid number representation. If no such match is found, the last parameter will be returned. Any number of parameters can be used.

Syntax:

```
alt(expr1[ , expr2 , expr3 , ...] , else)
```

Arguments:

Arguments

Argument	Description
expr1	The first expression to check for a valid number representation.
expr2	The second expression to check for a valid number representation.
expr3	The third expression to check for a valid number representation.
else	Value to return if none of the previous parameters has a valid number representation.

The alt function is often used with number or date interpretation functions. This way, Qlik Sense can test different date formats in a prioritized order. It can also be used to handle NULL values in numerical expressions.

Examples:

Examples

Example	Result
`alt( date#( dat , 'YYYY/MM/DD' ), date#( dat , 'MM/DD/YYYY' ), date#( dat , 'MM/DD/YY' ), 'No valid date' )`	This expression will test if the field date contains a date according to any of the three specified date formats. If so, it will return a dual value containing the original string and a valid number representation of a date. If no match is found, the text 'No valid date' will be returned (without any valid number representation).
`alt(Sales,0) + alt(Margin,0)`	This expression adds the fields Sales and Margin, replacing any missing value (NULL) with a 0.

class

The **class** function assigns the first parameter to a class interval. The result is a dual value with a<=x<b as the textual value, where a and b are the upper and lower limits of the bin, and the lower bound as numeric value.

Syntax:

```
class(expression, interval [ , label [ , offset ]])
```

Arguments:

Arguments

Argument	Description
interval	A number that specifies the bin width.

Argument	Description
label	An arbitrary string that can replace the 'x' in the result text.
offset	A number that can be used as offset from the default starting point of the classification. The default starting point is normally 0.

Examples:

Examples

Example	Result
`class( var,10 )` with `var = 23`	returns '20<=x<30'
`class( var,5,'value' )` with `var = 23`	returns '20<= value <25'
`class( var,10,'x',5 )` with `var = 23`	returns '15<=x<25'

Example - Load script using class

Example: load script

Load script

In this example, we load a table containing name and age of people. We want to add a field that classifies each person according to an age group with a ten year interval. The original source table looks like the following.

Results

Name	Age
John	25
Karen	42
Yoshi	53

To add the age group classification field, you can add a preceding load statement using the **class** function.

Create a new tab in the data load editor, and then load the following data as an inline load. Create the table below in Qlik Sense to see the results.

```
LOAD *,
class(Age, 10, 'age') As Agegroup;

LOAD * INLINE
[ Age, Name
25, John
42, Karen
53, Yoshi];
```

Results

Results

Name	Age	Agegroup
John	25	20 <= age < 30
Karen	42	40 <= age < 50
Yoshi	53	50 <= age < 60

coalesce

The **coalesce** function returns the first of the parameters that has a valid non-NULL representation. Any number of parameters can be used.

Syntax:

```
coalesce(expr1[ , expr2 , expr3 , ...])
```

Arguments:

Arguments

Argument	Description
expr1	The first expression to check for a valid non-NULL representation.
expr2	The second expression to check for a valid non-NULL representation.
expr3	The third expression to check for a valid non-NULL representation.

Examples:

Examples

Example	Result
	This expression changes all the NULL values of a field to 'N/A'.
`Coalesce(ProductDescription, ProductName, ProductCode, 'no description available')`	This expression will select between three different product description fields, for when some fields may not have values for the product. The first of the fields, in the order given, with a non-null value will be returned. If none of the fields contain a value, the result will be 'no description available'.
`Coalesce(TextBetween(FileName, '"', '"'), FileName)`	This expression will trim potential enclosing quotes from the field *FileName*. If the *FileName* given is quoted, these are removed, and the enclosed, unquoted *FileName* is returned. If the *TextBetween* function doesn't find the delimiters it returns null, which the **Coalesce** rejects, returning instead the raw *FileName*.

if

The **if** function returns a value depending on whether the condition provided with the function evaluates as True or False.

Syntax:

```
if(condition , then [, else])
```

Arguments

Argument	Description
condition	Expression that is interpreted logically.
then	Expression that can be of any type. If the *condition* is True, then the if function returns the value of the *then* expression.
else	Expression that can be of any type. If the *condition* is False, then the if function returns the value of the *else* expression. This parameter is optional. If the *condition* is False, NULL is returned if you have not specified else.

Example

Example	Result
`if( Amount>= 0, 'OK', 'Alarm' )`	This expression tests if the amount is a positive number (0 or larger) and return 'OK' if it is. If the amount is less than 0, 'Alarm' is returned.

Example - Load script using if

Example: Load script

Load script

If can be used in load script with other methods and objects, including variables. For example, if you set a variable *threshold* and want to include a field in the data model based on that threshold, you can do the following.

Create a new tab in the data load editor, and then load the following data as an inline load. Create the table below in Qlik Sense to see the results.

```
Transactions:
Load * Inline [
transaction_id, transaction_date, transaction_amount, transaction_quantity, customer_id, size,
color_code
3750, 20180830, 23.56, 2, 2038593, L, Red
3751, 20180907, 556.31, 6, 203521, m, orange
3752, 20180916, 5.75, 1, 5646471, S, blue
3753, 20180922, 125.00, 7, 3036491, l, Black
3754, 20180922, 484.21, 13, 049681, xs, Red
3756, 20180922, 59.18, 2, 2038593, M, Blue
```

```
3757, 20180923, 177.42, 21, 203521, XL, Black
];

set threshold = 100;

/* Create new table called Transaction_Buckets
Compare transaction_amount field from Transaction table to threshold of 100.
Output results into a new field called Compared to Threshold
 */

Transaction_Buckets:
Load
   transaction_id,
   If(transaction_amount > $(threshold),'Greater than $(threshold)','Less than $(threshold)')
as [Compared to Threshold]
Resident Transactions;
```

Results

Qlik Sense table showing the output from using the *if* function in the load script.

transaction_id	Compared to Threshold
3750	Less than 100
3751	Greater than 100
3752	Less than 100
3753	Greater than 100
3754	Greater than 100
3756	Less than 100
3757	Greater than 100

Examples - Chart expressions using if

Examples: Chart expressions

Chart expression 1

Load script

Create a new tab in the data load editor, and then load the following data as an inline load. After loading the data, create the chart expression examples below in a Qlik Sense table.

```
MyTable:
LOAD * inline [Date, Location, Incidents
1/3/2016, Beijing, 0
1/3/2016, Boston, 12
1/3/2016, Stockholm, 3
1/3/2016, Toronto, 0
```

```
1/4/2016, Beijing, 0
1/4/2016, Boston, 8];
```

Qlik Sense table showing examples of the *if* function in a chart expression.

Date	Location	Incidents	if(Incidents>=10, 'Critical', 'Ok')	if(Incidents>=10, 'Critical', If(Incidents>=1 and Incidents<10, 'Warning', 'Ok'))
1/3/2016	Beijing	0	Ok	Ok
1/3/2016	Boston	12	Critical	Critical
1/3/2016	Stockholm	3	Ok	Warning
1/3/2016	Toronto	0	Ok	Ok
1/4/2016	Beijing	0	Ok	Ok
1/4/2016	Boston	8	Ok	Warning

Chart expression 2

In a new app, add the following script in a new tab in the data load editor, and then load the data. You can then create the table with the chart expressions below.

```
SET FirstWeekDay=0;
Load
Date(MakeDate(2022)+RecNo()-1) as Date
Autogenerate 14;
```

Qlik Sense table showing an example of the *if* function in a chart expression.

Date	WeekDay(Date)	If(WeekDay (Date)>=5,'WeekEnd','Normal Day')
1/1/2022	Sat	WeekEnd
1/2/2022	Sun	WeekEnd
1/3/2022	Mon	Normal Day
1/4/2022	Tue	Normal Day
1/5/2022	Wed	Normal Day
1/6/2022	Thu	Normal Day
1/7/2022	Fri	Normal Day
1/8/2022	Sat	WeekEnd
1/9/2022	Sun	WeekEnd
1/10/2022	Mon	Normal Day
1/11/2022	Tue	Normal Day

Date	WeekDay(Date)	If(WeekDay (Date)>=5,'WeekEnd','Normal Day')
1/12/2022	Wed	Normal Day
1/13/2022	Thu	Normal Day
1/14/2022	Fri	Normal Day

match

The **match** function compares the first parameter with all the following ones and returns the numeric location of the expressions that match. The comparison is case sensitive.

Syntax:

```
match( str, expr1 [ , expr2,...exprN ])
```

*If you want to use case insensitive comparison, use the **mixmatch** function. If you want to use case insensitive comparison and wildcards, use the **wildmatch** function.*

Example: Load script using match

Example: Load script

Load script

You can use match to load a subset of data. For example, you can return a numeric value for an expression in the function. You can then limit the data loaded based on the numeric value. Match returns 0 if there is no match. All expressions that are not matched in this example will therefore return 0 and will be excluded from the data load by the WHERE statement.

Create a new tab in the data load editor, and then load the following data as an inline load. Create the table below in Qlik Sense to see the results.

```
Transactions:
Load * Inline [
transaction_id, transaction_date, transaction_amount, transaction_quantity, customer_id, size,
color_code
3750, 20180830, 23.56, 2, 2038593, L, Red
3751, 20180907, 556.31, 6, 203521, m, orange
3752, 20180916, 5.75, 1, 5646471, S, blue
3753, 20180922, 125.00, 7, 3036491, l, Black
3754, 20180922, 484.21, 13, 049681, xs, Red
3756, 20180922, 59.18, 2, 2038593, M, Blue
3757, 20180923, 177.42, 21, 203521, XL, Black
];

/*
Create new table called Transaction_Buckets
Create new fields called Customer, and Color code - Blue and Black
```

```
Load Transactions table.
Match returns 1 for 'Blue', 2 for 'Black'.
Does not return a value for 'blue' because match is case sensitive.
Only values that returned numeric value greater than 0
are loaded by WHERE statment into Transactions_Buckets table.
*/

Transaction_Buckets:
Load
 customer_id,
 customer_id as [Customer],
 color_code as [Color Code Blue and Black]
Resident Transactions
Where match(color_code,'Blue','Black') > 0;
```

Results

Qlik Sense table showing the output from using the match function in the load script

Color Code Blue and Black	Customer
Black	203521
Black	3036491
Blue	2038593

Examples - Chart expressions using match

Examples: Chart expressions

Chart expression 1

Load script

Create a new tab in the data load editor, and then load the following data as an inline load. After loading the data, create the chart expression examples below in a Qlik Sense table.

```
MyTable:
Load * inline [Cities, Count
Toronto, 123
Toronto, 234
Toronto, 231
Boston, 32
Boston, 23
Boston, 1341
Beijing, 234
Beijing, 45
Beijing, 235
Stockholm, 938
Stockholm, 39
Stockholm, 189
zurich, 2342
```

```
zurich, 9033
zurich, 0039];
```

The first expression in the table below returns 0 for Stockholm because 'Stockholm' is not included in the list of expressions in the **match** function. It also returns 0 for 'Zurich' because the **match** comparison is case-sensitive.

Qlik Sense table showing examples of the *match* function in a chart expression

Cities	**match(Cities,'Toronto','Boston','Beijing','Zurich')**	**match(Cities,'Toronto','Boston','Beijing','Stockholm','zurich')**
Beijing	3	3
Boston	2	2
Stockholm	0	4
Toronto	1	1
zurich	0	5

Chart expression 2

You can use match to perform a custom sort for an expression.

By default, columns sort numerically or alphabetically, depending on the data.

Qlik Sense table showing an example of the default sort order

Cities
Beijing
Boston
Stockholm
Toronto
zurich

To change the order, do the following:

1. Open the **Sorting** section for your chart in the **Properties** panel.
2. Turn off auto sorting for the column on which you want to do a custom sort.
3. Deselect **Sort numerically** and **Sort alphabetically**.
4. Select **Sort by expression**, and then enter an expression similar to the following:
   `=match( Cities, 'Toronto','Boston','Beijing','Stockholm','zurich')`
   The sort order on the Cities column changes.

Qlik Sense table showing an example of changing the sort order using the *match* function

Cities
Toronto
Boston
Beijing
Stockholm
zurich

You can also view the numeric value that is returned.

Qlik Sense table showing an example of the numeric values that are returned from the *match* function

Cities	Cities & ' - ' & match (Cities, 'Toronto','Boston', 'Beijing','Stockholm','zurich')
Toronto	Toronto - 1
Boston	Boston - 2
Beijing	Beijing - 3
Stockholm	Stockholm - 4
zurich	zurich - 5

mixmatch

The **mixmatch** function compares the first parameter with all the following ones and returns the numeric location of the expressions that match. The comparison is case insensitive.

Syntax:

```
mixmatch( str, expr1 [ , expr2,...exprN ])
```

If you instead want to use case sensitive comparison, use the **match** function. If you want to use case insensitive comparison and wildcards, use the **wildmatch** function.

Example - Load script using mixmatch

Example: Load script

Load script

You can use mixmatch to load a subset of data. For example, you can return a numeric value for an expression in the function. You can then limit the data loaded based on the numeric value. Mixmatch returns 0 if there is no match. All expressions that are not matched in this example will therefore return 0 and will be excluded from the data load by the WHERE statement.

Create a new tab in the data load editor, and then load the following data as an inline load. Create the table below in Qlik Sense to see the results.

```
Load * Inline [
transaction_id, transaction_date, transaction_amount, transaction_quantity, customer_id, size,
color_code
3750, 20180830, 23.56, 2, 2038593, L, Red
3751, 20180907, 556.31, 6, 203521, m, orange
3752, 20180916, 5.75, 1, 5646471, S, blue
3753, 20180922, 125.00, 7, 3036491, l, Black
3754, 20180922, 484.21, 13, 049681, xs, Red
3756, 20180922, 59.18, 2, 2038593, M, Blue
3757, 20180923, 177.42, 21, 203521, XL, Black
];

/*
Create new table called Transaction_Buckets
Create new fields called Customer, and Color code - Black, Blue, blue
Load Transactions table.
Mixmatch returns 1 for 'Black', 2 for 'Blue'.
Also returns 3 for 'blue' because mixmatch is not case sensitive.
Only values that returned numeric value greater than 0
are loaded by WHERE statement into Transactions_Buckets table.
*/

Transaction_Buckets:
Load
   customer_id,
   customer_id as [Customer],
   color_code as [Color Code - Black, Blue, blue]
Resident Transactions
Where mixmatch(color_code,'Black','Blue') > 0;
```

Results

Qlik Sense table showing the output from using the mixmatch function in the load script.

Color Code Black, Blue, blue	Customer
Black	203521
Black	3036491
Blue	2038593
blue	5646471

Examples - Chart expressions using mixmatch

Examples: Chart expressions

Create a new tab in the data load editor, and then load the following data as an inline load. After loading the data, create the chart expression examples below in a Qlik Sense table.

Chart expression 1

```
MyTable:
Load * inline [Cities, Count
Toronto, 123
Toronto, 234
Toronto, 231
Boston, 32
Boston, 23
Boston, 1341
Beijing, 234
Beijing, 45
Beijing, 235
Stockholm, 938
Stockholm, 39
Stockholm, 189
zurich, 2342
zurich, 9033
zurich, 0039];
```

The first expression in the table below returns 0 for Stockholm because 'Stockholm' is not included in the list of expressions in the **mixmatch** function. It returns 4 for 'Zurich' because the **mixmatch** comparison is not case-sensitive.

Qlik Sense table showing examples of the *mixmatch* function in a chart expression

Cities	**mixmatch(Cities,'Toronto','Boston','Beijing','Zurich')**	**mixmatch(Cities,'Toronto','Boston','Beijing','Stockholm','Zurich')**
Beijing	3	3
Boston	2	2
Stockholm	0	4
Toronto	1	1
zurich	4	5

Chart expression 2

You can use mixmatch to perform a custom sort for an expression.

By default, columns sort alphabetically or numerically, depending on the data.

Qlik Sense table showing an example of the default sort order

Cities
Beijing
Boston

Cities
Stockholm
Toronto
zurich

To change the order, do the following:

1. Open the **Sorting** section for your chart in the **Properties** panel.
2. Turn off auto sorting for the column on which you want to do a custom sort.
3. Deselect **Sort numerically** and **Sort alphabetically**.
4. Select **Sort by expression**, and then enter the following expression:
   `=mixmatch( Cities, 'Toronto','Boston','Beijing','Stockholm','Zurich')`
   The sort order on the Cities column changes.

Qlik Sense table showing an example of changing the sort order using the *mixmatch* function.

Cities
Toronto
Boston
Beijing
Stockholm
zurich

You can also view the numeric value that is returned.

Qlik Sense table showing an example of the numeric values that are returned from the *mixmatch* function.

Cities	Cities & ' - ' & mixmatch (Cities, 'Toronto','Boston', 'Beijing','Stockholm','Zurich')
Toronto	Toronto - 1
Boston	Boston - 2
Beijing	Beijing - 3
Stockholm	Stockholm - 4
zurich	zurich - 5

pick

The pick function returns the *n*:th expression in the list.

Syntax:

```
pick(n, expr1[ , expr2,...exprN])
```

Arguments:

Arguments

Argument	Description
n	*n* is an integer between 1 and N.

Example:

Example

Example	Result
`pick( N, 'A','B',4, 6 )`	returns 'B' if N = 2 returns 4 if N = 3

wildmatch

The **wildmatch** function compares the first parameter with all the following ones and returns the number of the expression that matches. It permits the use of wildcard characters (***** and **?**) in the comparison strings. ***** matches any sequence of characters. **?** matches any single character. The comparison is case insensitive.

Syntax:

```
wildmatch( str, expr1 [ , expr2,...exprN ])
```

If you want to use comparison without wildcards, use the **match** or **mixmatch** functions.

Example: Load script using wildmatch

Example: Load script

Load script

You can use wildmatch to load a subset of data. For example, you can return a numeric value for an expression in the function. You can then limit the data loaded based on the numeric value. Wildmatch returns 0 if there is no match. All expressions that are not matched in this example will therefore return 0 and will be excluded from the data load by the WHERE statement.

Create a new tab in the data load editor, and then load the following data as an inline load. Create the table below in Qlik Sense to see the results.

```
Transactions:
Load * Inline [
transaction_id, transaction_date, transaction_amount, transaction_quantity, customer_id, size,
color_code
3750, 20180830, 23.56, 2, 2038593, L, Red
3751, 20180907, 556.31, 6, 203521, m, orange
3752, 20180916, 5.75, 1, 5646471, S, blue
3753, 20180922, 125.00, 7, 3036491, l, Black
```

```
3754, 20180922, 484.21, 13, 049681, xs, Red
3756, 20180922, 59.18, 2, 2038593, M, Blue
3757, 20180923, 177.42, 21, 203521, XL, Black
];

/*
Create new table called Transaction_Buckets
Create new fields called Customer, and Color code - Black, Blue, blue, red
Load Transactions table.
Wildmatch returns 1 for 'Black', 'Blue', and 'blue', and 2 for 'Red'.
Only values that returned numeric value greater than 0
are loaded by WHERE statement into Transactions_Buckets table.
*/

Transaction_Buckets:
Load
   customer_id,
   customer_id as [Customer],
   color_code as [Color Code Black, Blue, blue, Red]
Resident Transactions
Where wildmatch(color_code,'Bl*','R??') > 0;
```

Results

Qlik Sense table showing the output from using the *wildmatch* function in the load script

Color Code Black, Blue, blue, Red	Customer
Black	203521
Black	3036491
Blue	2038593
blue	5646471
Red	049681
Red	2038593

Examples: Chart expressions using wildmatch

Example: Chart expression

Chart expression 1

Create a new tab in the data load editor, and then load the following data as an inline load. After loading the data, create the chart expression examples below in a Qlik Sense table.

```
MyTable:
Load * inline [Cities, Count
Toronto, 123
Toronto, 234
Toronto, 231
```

```
Boston, 32
Boston, 23
Boston, 1341
Beijing, 234
Beijing, 45
Beijing, 235
Stockholm, 938
Stockholm, 39
Stockholm, 189
zurich, 2342
zurich, 9033
zurich, 0039];
```

The first expression in the table below returns 0 for Stockholm because 'Stockholm' is not included in the list of expressions in the **wildmatch** function. It also returns 0 for 'Boston' because ? only matches on a single character.

Qlik Sense table showing examples of the *wildmatch* function in a chart expression

Cities	**wildmatch(Cities,'Tor*','?ton','Beijing','*urich')**	**wildmatch(Cities,'Tor*','???ton','Beijing','Stockholm','*urich')**
Beijing	3	3
Boston	0	2
Stockholm	0	4
Toronto	1	1
zurich	4	5

Chart expression 2

You can use wildmatch to perform a custom sort for an expression.

By default, columns sort numerically or alphabetically, depending on the data.

Qlik Sense table showing an example of the default sort order

Cities
Beijing
Boston
Stockholm
Toronto
zurich

To change the order, do the following:

1. Open the **Sorting** section for your chart in the **Properties** panel.
2. Turn off auto sorting for the column on which you want to do a custom sort.
3. Deselect **Sort numerically** and **Sort alphabetically**.
4. Select **Sort by expression**, and then enter an expression similar to the following:
   `=wildmatch( Cities, 'Tor*','???ton','Beijing','Stockholm','*urich')`
   The sort order on the Cities column changes.

Qlik Sense table showing an example of changing the sort order using the *wildmatch* function.

Cities
Toronto
Boston
Beijing
Stockholm
zurich

You can also view the numeric value that is returned.

Qlik Sense table showing an example of the numeric values that are returned from the *wildmatch* function

Cities	**Cities & ' - ' & wildmatch (Cities, 'Tor*','???ton','Beijing','Stockholm','*urich')**
Toronto	Toronto - 1
Boston	Boston - 2
Beijing	Beijing - 3
Stockholm	Stockholm - 4
zurich	zurich - 5

5.6 Counter functions

This section describes functions related to record counters during **LOAD** statement evaluation in the data load script. The only function that can be used in chart expressions is **RowNo()**.

Some counter functions do not have any parameters, but the trailing parentheses are however still required.

Counter functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

autonumber

This script function returns a unique integer value for each distinct evaluated value of *expression* encountered during the script execution. This function can be used e.g. for creating a compact memory representation of a complex key.

```
autonumber (expression[ , AutoID])
```

autonumberhash128

This script function calculates a 128-bit hash of the combined input expression values and the returns a unique integer value for each distinct hash value encountered during the script execution. This function can be used for example for creating a compact memory representation of a complex key.

```
autonumberhash128 (expression {, expression})
```

autonumberhash256

This script function calculates a 256-bit hash of the combined input expression values and returns a unique integer value for each distinct hash value encountered during the script execution. This function can be used e.g. for creating a compact memory representation of a complex key.

```
autonumberhash256 (expression {, expression})
```

IterNo

This script function returns an integer indicating for which time one single record is evaluated in a **LOAD** statement with a **while** clause. The first iteration has number 1. The **IterNo** function is only meaningful if used together with a **while** clause.

```
IterNo ( )
```

RecNo

This script functions returns an integer for the number of the currently read row of the current table. The first record is number 1.

```
RecNo ( )
```

RowNo - script function

This function returns an integer for the position of the current row in the resulting Qlik Sense internal table. The first row is number 1.

```
RowNo ( )
```

RowNo - chart function

RowNo() returns the number of the current row within the current column segment in a table. For bitmap charts, **RowNo()** returns the number of the current row within the chart's straight table equivalent.

```
RowNo - chart function([TOTAL])
```

autonumber

This script function returns a unique integer value for each distinct evaluated value of *expression* encountered during the script execution. This function can be used e.g. for creating a compact memory representation of a complex key.

> *You can only connect **autonumber** keys that have been generated in the same data load, as the integer is generated according to the order the table is read. If you need to use keys that are persistent between data loads, independent of source data sorting, you should use the **hash128**, **hash160** or **hash256** functions.*

Syntax:

```
autonumber(expression[ , AutoID])
```

Arguments:

Argument	Description
AutoID	In order to create multiple counter instances if the **autonumber** function is used on different keys within the script, the optional parameter *AutoID* can be used for naming each counter.

Example: Creating a composite key

In this example we create a composite key using the **autonumber** function to conserve memory. The example is brief for demonstration purpose, but would be meaningful with a table containing a large number of rows.

Example data

Region	Year	Month	Sales
North	2014	May	245
North	2014	May	347
North	2014	June	127
South	2014	June	645
South	2013	May	367
South	2013	May	221

The source data is loaded using inline data. Then we add a preceding load which creates a composite key from the Region, Year and Month fields.

```
RegionSales:
LOAD *,
AutoNumber(Region&Year&Month) as RYMkey;

LOAD * INLINE
[ Region, Year, Month, Sales
North, 2014,   May,    245
North, 2014,   May,    347
North, 2014,   June,   127
South, 2014,   June,   645
South, 2013,   May, 367
South, 2013,   May,    221
];
```

The resulting table looks like this:

Results table

Region	Year	Month	Sales	RYMkey
North	2014	May	245	1
North	2014	May	347	1
North	2014	June	127	2
South	2014	June	645	3
South	2013	May	367	4
South	2013	May	221	4

In this example you can refer to the RYMkey, for example 1, instead of the string 'North2014May' if you need to link to another table.

Now we load a source table of costs in a similar way. The Region, Year and Month fields are excluded in the preceding load to avoid creating a synthetic key, we are already creating a composite key with the **autonumber** function, linking the tables.

```
RegionCosts:
LOAD Costs,
AutoNumber(Region&Year&Month) as RYMkey;

LOAD * INLINE
[ Region, Year, Month, Costs
South, 2013,   May,    167
North, 2014,   May,    56
North, 2014,   June,   199
South, 2014,   June,   64
South, 2013,   May, 172
South, 2013,   May,    126
];
```

Now we can add a table visualization to a sheet, and add the Region, Year and Month fields, as well as Sum measures for the sales and the costs. The table will look like this:

Results table

Region	Year	Month	Sum([Sales])	Sum([Costs])
Totals	-	-	1952	784
North	2014	June	127	199
North	2014	May	592	56
South	2014	June	645	64
South	2013	May	588	465

autonumberhash128

This script function calculates a 128-bit hash of the combined input expression values and the returns a unique integer value for each distinct hash value encountered during the script execution. This function can be used for example for creating a compact memory representation of a complex key.

> *You can only connect **autonumberhash128** keys that have been generated in the same data load, as the integer is generated according to the order the table is read. If you need to use keys that are persistent between data loads, independent of source data sorting, you should use the **hash128**, **hash160** or **hash256** functions.*

Syntax:

```
autonumberhash128(expression {, expression})
```

Example: Creating a composite key

In this example we create a composite key using the **autonumberhash128** function to conserve memory. The example is brief for demonstration purpose, but would be meaningful with a table containing a large number of rows.

Example data

Region	Year	Month	Sales
North	2014	May	245
North	2014	May	347
North	2014	June	127
South	2014	June	645
South	2013	May	367
South	2013	May	221

The source data is loaded using inline data. Then we add a preceding load which creates a composite key from the Region, Year and Month fields.

```
RegionSales:
LOAD *,
AutoNumberHash128(Region, Year, Month) as RYMkey;

LOAD * INLINE
[ Region, Year, Month, Sales
North, 2014,   May,    245
North, 2014,   May,    347
North, 2014,   June,   127
South, 2014,   June,   645
South, 2013,   May, 367
```

```
South, 2013,   May,    221
];
```

The resulting table looks like this:

Results table

Region	Year	Month	Sales	RYMkey
North	2014	May	245	1
North	2014	May	347	1
North	2014	June	127	2
South	2014	June	645	3
South	2013	May	367	4
South	2013	May	221	4

In this example you can refer to the RYMkey, for example 1, instead of the string 'North2014May' if you need to link to another table.

Now we load a source table of costs in a similar way. The Region, Year and Month fields are excluded in the preceding load to avoid creating a synthetic key, we are already creating a composite key with the **autonumberhash128** function, linking the tables.

```
RegionCosts:
LOAD Costs,
AutoNumberHash128(Region, Year, Month) as RYMkey;

LOAD * INLINE
[ Region, Year, Month, Costs
South, 2013,   May,    167
North, 2014,   May,    56
North, 2014,   June,   199
South, 2014,   June,   64
South, 2013,   May, 172
South, 2013,   May,    126
];
```

Now we can add a table visualization to a sheet, and add the Region, Year and Month fields, as well as Sum measures for the sales and the costs. The table will look like this:

Results table

Region	Year	Month	Sum([Sales])	Sum([Costs])
Totals	-	-	1952	784
North	2014	June	127	199
North	2014	May	592	56

Region	Year	Month	Sum([Sales])	Sum([Costs])
South	2014	June	645	64
South	2013	May	588	465

autonumberhash256

This script function calculates a 256-bit hash of the combined input expression values and returns a unique integer value for each distinct hash value encountered during the script execution. This function can be used e.g. for creating a compact memory representation of a complex key.

> *You can only connect **autonumberhash256** keys that have been generated in the same data load, as the integer is generated according to the order the table is read. If you need to use keys that are persistent between data loads, independent of source data sorting, you should use the **hash128**, **hash160** or **hash256** functions.*

Syntax:

```
autonumberhash256(expression {, expression})
```

Example: Creating a composite key

In this example we create a composite key using the **autonumberhash256** function to conserve memory. The example is brief for demonstration purpose, but would be meaningful with a table containing a large number of rows.

Example table

Region	Year	Month	Sales
North	2014	May	245
North	2014	May	347
North	2014	June	127
South	2014	June	645
South	2013	May	367
South	2013	May	221

The source data is loaded using inline data. Then we add a preceding load which creates a composite key from the Region, Year and Month fields.

```
RegionSales:
LOAD *,
AutoNumberHash256(Region, Year, Month) as RYMkey;

LOAD * INLINE
```

```
[ Region, Year, Month, Sales
North, 2014,   May,    245
North, 2014,   May,    347
North, 2014,   June,   127
South, 2014,   June,   645
South, 2013,   May, 367
South, 2013,   May,    221
];
```

The resulting table looks like this:

Results table

Region	Year	Month	Sales	RYMkey
North	2014	May	245	1
North	2014	May	347	1
North	2014	June	127	2
South	2014	June	645	3
South	2013	May	367	4
South	2013	May	221	4

In this example you can refer to the RYMkey, for example 1, instead of the string 'North2014May' if you need to link to another table.

Now we load a source table of costs in a similar way. The Region, Year and Month fields are excluded in the preceding load to avoid creating a synthetic key, we are already creating a composite key with the **autonumberhash256** function, linking the tables.

```
RegionCosts:
LOAD Costs,
AutoNumberHash256(Region, Year, Month) as RYMkey;

LOAD * INLINE
[ Region, Year, Month, Costs
South, 2013,   May,    167
North, 2014,   May,    56
North, 2014,   June,   199
South, 2014,   June,   64
South, 2013,   May, 172
South, 2013,   May,    126
];
```

Now we can add a table visualization to a sheet, and add the Region, Year and Month fields, as well as Sum measures for the sales and the costs. The table will look like this:

Results table

Region	Year	Month	Sum([Sales])	Sum([Costs])
Totals	-	-	1952	784
North	2014	June	127	199
North	2014	May	592	56
South	2014	June	645	64
South	2013	May	588	465

IterNo

This script function returns an integer indicating for which time one single record is evaluated in a **LOAD** statement with a **while** clause. The first iteration has number 1. The **IterNo** function is only meaningful if used together with a **while** clause.

Syntax:

```
IterNo( )
```

Examples and results:

Example:

```
LOAD
   IterNo() as Day,
   Date( StartDate + IterNo() - 1 ) as Date
   While StartDate + IterNo() - 1 <= EndDate;

LOAD * INLINE
[StartDate, EndDate
2014-01-22, 2014-01-26
];
```

This **LOAD** statement will generate one record per date within the range defined by **StartDate** and **EndDate**.

In this example, the resulting table will look like this:

Results table

Day	Date
1	2014-01-22
2	2014-01-23
3	2014-01-24
4	2014-01-25
5	2014-01-26

RecNo

This script functions returns an integer for the number of the currently read row of the current table. The first record is number 1.

Syntax:

```
RecNo( )
```

In contrast to **RowNo( )**, which counts rows in the resulting Qlik Sense table, **RecNo( )**, counts the records in the raw data table and is reset when a raw data table is concatenated to another.

Example: Data load script

Raw data table load:

```
Tab1:
LOAD * INLINE
[A, B
1, aa
2,cc
3,ee];

Tab2:
LOAD * INLINE
[C, D
5, xx
4,yy
6,zz];
```

Loading record and row numbers for selected rows:

```
QTab:
LOAD *,
RecNo( ),
RowNo( )
resident Tab1 where A<>2;

LOAD
C as A,
D as B,
RecNo( ),
RowNo( )
resident Tab2 where A<>5;

//We don't need the source tables anymore, so we drop them
Drop tables Tab1, Tab2;
```

The resulting Qlik Sense internal table:

Results table

A	B	RecNo( )	RowNo( )
1	aa	1	1
3	ee	3	2
4	yy	2	3
6	zz	3	4

RowNo

This function returns an integer for the position of the current row in the resulting Qlik Sense internal table. The first row is number 1.

Syntax:

```
RowNo( [TOTAL])
```

In contrast to **RecNo( )**, which counts the records in the raw data table, the **RowNo( )** function does not count records that are excluded by **where** clauses and is not reset when a raw data table is concatenated to another.

> *If you use preceding load, that is, a number of stacked **LOAD** statements reading from the same table, you can only use **RowNo( )** in the top **LOAD** statement. If you use **RowNo( )** in subsequent **LOAD** statements, 0 is returned.*

Example: Data load script

Raw data table load:

```
Tab1:
LOAD * INLINE
[A, B
1, aa
2,cc
3,ee];

Tab2:
LOAD * INLINE
[C, D
5, xx
4,yy
6,zz];
```

Loading record and row numbers for selected rows:

```
QTab:
LOAD *,
RecNo( ),
RowNo( )
```

```
resident Tab1 where A<>2;

LOAD
C as A,
D as B,
RecNo( ),
RowNo( )
resident Tab2 where A<>5;

//We don't need the source tables anymore, so we drop them
Drop tables Tab1, Tab2;
```

The resulting Qlik Sense internal table:

Results table

A	B	RecNo( )	RowNo( )
1	aa	1	1
3	ee	3	2
4	yy	2	3
6	zz	3	4

RowNo - chart function

RowNo() returns the number of the current row within the current column segment in a table. For bitmap charts, **RowNo()** returns the number of the current row within the chart's straight table equivalent.

If the table or table equivalent has multiple vertical dimensions, the current column segment will include only rows with the same values as the current row in all dimension columns, except for the column showing the last dimension in the inter-field sort order.

Column segments

	Region	Country	Population	Rank(Population)
Column segment #1	Americas	Mexico	128,932,753	2
	Americas	Canada	37,742,154	3
	Americas	United States of America	331,002,651	1
Column segment #2	Europe	Sweden	10,099,265	4
	Europe	United Kingdom	67,886,011	2
	Europe	France	65,273,511	3
	Europe	Germany	83,783,942	1

*Sorting on y-values in charts or sorting by expression columns in tables is not allowed when **RowNo ()** is used in any of the chart's expressions. These sort alternatives are therefore automatically disabled.*

Syntax:

```
RowNo([TOTAL])
```

Return data type: integer

Arguments:

Argument	Description
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

Example: Chart expression using RowNo

Example - chart expression

Load script

Load the following data as an inline load in the data load editor to create the chart expression examples below.

```
Temp:
LOAD * inline [
Customer|Product|OrderNumber|UnitSales|UnitPrice
Astrida|AA|1|4|16
Astrida|AA|7|10|15
Astrida|BB|4|9|9
Betacab|CC|6|5|10
Betacab|AA|5|2|20
Betacab|BB|1|25| 25
Canutility|AA|3|8|15
Canutility|CC|5|4|19
Divadip|CC|2|4|16
Divadip|DD|3|1|25
] (delimiter is '|');
```

Chart expression

Create a table visualization in a Qlik Sense sheet with **Customer** and **UnitSales** as dimensions. Add `RowNo( )` and `RowNo(TOTAL)` as measures labeled **Row in Segment** and **Row Number**, respectively. Add the following expression to the table as a measure:

```
If( RowNo( )=1, 0, UnitSales / Above( UnitSales ))
```

Result

Customer	UnitSales	Row in Segment	Row Number	If(RowNo( )=1, 0, UnitSales / Above(UnitSales))
Astrida	4	1	1	0
Astrida	9	2	2	2.25

Customer	UnitSales	Row in Segment	Row Number	If(RowNo( )=1, 0, UnitSales / Above(UnitSales))
Astrida	10	3	3	1.1111111111111
Betacab	2	1	4	0
Betacab	5	2	5	2.5
Betacab	25	3	6	5
Canutility	4	1	7	0
Canutility	8	2	8	2
Divadip	1	1	9	0
Divadip	4	2	10	4

Explanation

The **Row in Segment** column shows the results 1,2,3 for the column segment containing the values of UnitSales for customer Astrida. The row numbering then begins at 1 again for the next column segment, which is Betacab.

The **Row Number** column disregards the dimensions because of the `TOTAL` argument for `RowNo()` and counts the rows in the table.

This expression returns 0 for the first row in each column segment, so the column shows:

0, 2.25, 1.1111111, 0, 2.5, 5, 0, 2, 0, and 4.

See also:

Above - chart function (page 848)

5.7 Date and time functions

Qlik Sense date and time functions are used to transform and convert date and time values. All functions can be used in both the data load script and in chart expressions.

Functions are based on a date-time serial number that equals the number of days since December 30, 1899. The integer value represents the day and the fractional value represents the time of the day.

Qlik Sense uses the numerical value of the parameter, so a number is valid as a parameter also when it is not formatted as a date or a time. If the parameter does not correspond to numerical value, for example, because it is a string, then Qlik Sense attempts to interpret the string according to the date and time environment variables.

If the time format used in the parameter does not correspond to the one set in the environment variables, Qlik Sense will not be able to make a correct interpretation. To resolve this, either change the settings or use an interpretation function.

In the examples for each function, the default time and date formats hh:mm:ss and YYYY-MM-DD (ISO 8601) are assumed.

When processing a timestamp with a date or time function, Qlik Sense ignores any daylight savings time parameters unless the date or time function includes a geographical position.

For example, `ConvertToLocalTime( filetime('Time.qvd'), 'Paris')` would use daylight savings time parameters while `ConvertToLocalTime(filetime('Time.qvd'), 'GMT-01:00')` would not use daylight savings time parameters.

Date and time functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Integer expressions of time

second

This function returns an integer representing the second when the fraction of the **expression** is interpreted as a time according to the standard number interpretation.

```
second (expression)
```

minute

This function returns an integer representing the minute when the fraction of the **expression** is interpreted as a time according to the standard number interpretation.

```
minute (expression)
```

hour

This function returns an integer representing the hour when the fraction of the **expression** is interpreted as a time according to the standard number interpretation.

```
hour (expression)
```

day

This function returns an integer representing the day when the fraction of the **expression** is interpreted as a date according to the standard number interpretation.

```
day (expression)
```

week

This function returns an integer representing the week number according to ISO 8601. The week number is calculated from the date interpretation of the expression, according to the standard number interpretation.

```
week (expression)
```

month

This function returns a dual value: a month name as defined in the environment variable **MonthNames** and an integer between 1-12. The month is calculated from the date interpretation of the expression, according to the standard number interpretation.

```
month (expression)
```

year

This function returns an integer representing the year when the **expression** is interpreted as a date according to the standard number interpretation.

```
year (expression)
```

weekyear

This function returns the year to which the week number belongs according to ISO 8601. The week number ranges between 1 and approximately 52.

```
weekyear (expression)
```

weekday

This function returns a dual value with:

- A day name as defined in the environment variable **DayNames**.
- An integer between 0-6 corresponding to the nominal day of the week (0-6).

```
weekday (date)
```

Timestamp functions

now

This function returns a timestamp of the current time from the system clock. The default value is 1.

```
now ([ timer_mode])
```

today

This function returns the current date from the system clock.

```
today ([timer_mode])
```

LocalTime

This function returns a timestamp of the current time for a specified time zone.

```
localtime ([timezone [, ignoreDST ]])
```

Make functions

makedate

This function returns a date calculated from the year **YYYY**, the month **MM** and the day **DD**.

```
makedate (YYYY [ , MM [ , DD ] ])
```

makeweekdate

This function returns a date calculated from the year **YYYY**, the week **WW** and the day-of-week **D**.

```
makeweekdate (YYYY [ , WW [ , D ] ])
```

maketime

This function returns a time calculated from the hour **hh**, the minute **mm**, and the second **ss**.

```
maketime (hh [ , mm [ , ss [ .fff ] ] ])
```

Other date functions

AddMonths

This function returns the date occurring **n** months after **startdate** or, if **n** is negative, the date occurring **n** months before **startdate**.

```
addmonths (startdate, n , [ , mode])
```

AddYears

This function returns the date occurring **n** years after **startdate** or, if **n** is negative, the date occurring **n** years before **startdate**.

```
addyears (startdate, n)
```

yeartodate

This function finds if the input timestamp falls within the year of the date the script was last loaded, and returns True if it does, False if it does not.

```
yeartodate (date [ , yearoffset [ , firstmonth [ , todaydate] ] ])
```

Timezone functions

timezone

This function returns the name of the current time zone, as defined in Windows.

```
timezone ( )
```

GMT

This function returns the current Greenwich Mean Time, as derived from the regional settings.

```
GMT ( )
```

UTC

Returns the current Coordinated Universal Time.

```
UTC ( )
```

daylightsaving

Returns the current adjustment for daylight saving time, as defined in Windows.

```
daylightsaving ( )
```

converttolocaltime

Converts a UTC or GMT timestamp to local time as a dual value. The place can be any of a number of cities, places and time zones around the world.

```
converttolocaltime (timestamp [, place [, ignore_dst=false]])
```

Set time functions

setdateyear

This function takes as input a **timestamp** and a **year** and updates the **timestamp** with the **year** specified in input.

```
setdateyear  (timestamp, year)
```

setdateyearmonth

This function takes as input a **timestamp**, a **month** and a **year** and updates the **timestamp** with the **year** and the **month** specified in input.

```
setdateyearmonth  (timestamp, year, month)
```

In... functions

inyear

This function returns True if **timestamp** lies inside the year containing **base_date**.

```
inyear (date, basedate , shift [, first_month_of_year = 1])
```

inyeartodate

This function returns True if **timestamp** lies inside the part of year containing **base_date** up until and including the last millisecond of **base_date**.

```
inyeartodate  (date, basedate , shift [, first_month_of_year = 1])
```

inquarter

This function returns True if **timestamp** lies inside the quarter containing **base_date**.

```
inquarter  (date, basedate , shift [, first_month_of_year = 1])
```

inquartertodate

This function returns True if **timestamp** lies inside the part of the quarter containing **base_date** up until and including the last millisecond of **base_date**.

```
inquartertodate  (date, basedate , shift [, first_month_of_year = 1])
```

inmonth

This function returns True if **timestamp** lies inside the month containing **base_date**.

```
inmonth (date, basedate , shift)
```

inmonthtodate

Returns True if **date** lies inside the part of month containing **basedate** up until and including the last millisecond of **basedate**.

```
inmonthtodate (date, basedate , shift)
```

inmonths

This function finds if a timestamp falls within the same month, bi-month, quarter, four-month period, or half-year as a base date.It is also possible to find if the timestamp falls within a previous or following time period.

```
inmonths (n, date, basedate , shift [, first_month_of_year = 1])
```

inmonthstodate

This function finds if a timestamp falls within the part a period of the month, bi-month, quarter, four-month period, or half-year up to and including the last millisecond of `base_date`. It is also possible to find if the timestamp falls within a previous or following time period.

```
inmonthstodate (n, date, basedate , shift [, first_month_of_year = 1])
```

inweek

This function returns True if **timestamp** lies inside the week containing **base_date**.

```
inweek (date, basedate , shift [, weekstart])
```

inweektodate

This function returns True if **timestamp** lies inside the part of week containing **base_date** up until and including the last millisecond of **base_date**.

```
inweektodate (date, basedate , shift [, weekstart])
```

inlunarweek

This function finds if **timestamp** lies inside the lunar week containing **base_date**. Lunar weeks in Qlik Sense are defined by counting 1 January as the first day of the week.

```
inlunarweek (date, basedate , shift [, weekstart])
```

inlunarweektodate

This function finds if **timestamp** lies inside the part of the lunar week up to and including the last millisecond of **base_date**. Lunar weeks in Qlik Sense are defined by counting 1 January as the first day of the week.

```
inlunarweektodate (date, basedate , shift [, weekstart])
```

inday

This function returns True if **timestamp** lies inside the day containing **base_timestamp**.

```
inday (timestamp, basetimestamp , shift [, daystart])
```

indaytotime

This function returns True if **timestamp** lies inside the part of day containing **base_timestamp** up until and including the exact millisecond of **base_timestamp**.

```
indaytotime (timestamp, basetimestamp , shift [, daystart])
```

Start ... end functions

yearstart

This function returns a timestamp corresponding to the start of the first day of the year containing **date**. The default output format will be the **DateFormat** set in the script.

```
yearstart ( date [, shift = 0 [, first_month_of_year = 1]])
```

yearend

This function returns a value corresponding to a timestamp of the last millisecond of the last day of the year containing **date**. The default output format will be the **DateFormat** set in the script.

```
yearend ( date [, shift = 0 [, first_month_of_year = 1]])
```

yearname

This function returns a four-digit year as display value with an underlying numeric value corresponding to a timestamp of the first millisecond of the first day of the year containing **date**.

```
yearname (date [, shift = 0 [, first_month_of_year = 1]] )
```

quarterstart

This function returns a value corresponding to a timestamp of the first millisecond of the quarter containing **date**. The default output format will be the **DateFormat** set in the script.

```
quarterstart (date [, shift = 0 [, first_month_of_year = 1]])
```

quarterend

This function returns a value corresponding to a timestamp of the last millisecond of the quarter containing **date**. The default output format will be the **DateFormat** set in the script.

```
quarterend (date [, shift = 0 [, first_month_of_year = 1]])
```

quartername

This function returns a display value showing the months of the quarter (formatted according to the **MonthNames** script variable) and year with an underlying numeric value corresponding to a timestamp of the first millisecond of the first day of the quarter.

```
quartername (date [, shift = 0 [, first_month_of_year = 1]])
```

monthstart

This function returns a value corresponding to a timestamp of the first millisecond of the first day of the month containing **date**. The default output format will be the **DateFormat** set in the script.

```
monthstart (date [, shift = 0])
```

monthend

This function returns a value corresponding to a timestamp of the last millisecond of the last day of the month containing `date`. The default output format will be the `DateFormat` set in the script.

```
monthend (date [, shift = 0])
```

monthname

This function returns a display value showing the month (formatted according to the **MonthNames** script variable) and year with an underlying numeric value corresponding to a timestamp of the first millisecond of the first day of the month.

```
monthname (date [, shift = 0])
```

monthsstart

This function returns a value corresponding to the timestamp of the first millisecond of the month, bi-month, quarter, four-month period, or half-year containing a base date. It is also possible to find the timestamp for a previous or following time period.

```
monthsstart (n, date [, shift = 0 [, first_month_of_year = 1]])
```

monthsend

This function returns a value corresponding to a timestamp of the last millisecond of the month, bi-month, quarter, four-month period, or half-year containing a base date. It is also possible to find the timestamp for a previous or following time period.

```
monthsend (n, date [, shift = 0 [, first_month_of_year = 1]])
```

monthsname

This function returns a display value representing the range of the months of the period (formatted according to the **MonthNames** script variable) as well as the year. The underlying numeric value corresponds to a timestamp of the first millisecond of the month, bi-month, quarter, four-month period, or half-year containing a base date.

```
monthsname (n, date [, shift = 0 [, first_month_of_year = 1]])
```

weekstart

This function returns a value corresponding to a timestamp of the first millisecond of the first day (Monday) of the calendar week containing **date**. The default output format is the **DateFormat** set in the script.

```
weekstart (date [, shift = 0 [,weekoffset = 0]])
```

weekend

This function returns a value corresponding to a timestamp of the last millisecond of the last day (Sunday) of the calendar week containing **date** The default output format will be the **DateFormat** set in the script.

```
weekend (date [, shift = 0 [,weekoffset = 0]])
```

weekname

This function returns a value showing the year and week number with an underlying numeric value corresponding to a timestamp of the first millisecond of the first day of the week containing **date**.

```
weekname (date [, shift = 0 [,weekoffset = 0]])
```

lunarweekstart

This function returns a value corresponding to a timestamp of the first millisecond of the lunar week containing **date**. Lunar weeks in Qlik Sense are defined by counting 1 January as the first day of the week.

```
lunarweekstart (date [, shift = 0 [,weekoffset = 0]])
```

lunarweekend

This function returns a value corresponding to a timestamp of the last millisecond of the lunar week containing **date**. Lunar weeks in Qlik Sense are defined by counting 1 January as the first day of the week.

```
lunarweekend (date [, shift = 0 [,weekoffset = 0]])
```

lunarweekname

This function returns a display value showing the year and lunar week number corresponding to a timestamp of the first millisecond of the first day of the lunar week containing **date**. Lunar weeks in Qlik Sense are defined by counting 1 January as the first day of the week.

```
lunarweekname (date [, shift = 0 [,weekoffset = 0]])
```

daystart

This function returns a value corresponding to a timestamp with the first millisecond of the day contained in the **time** argument. The default output format will be the **TimestampFormat** set in the script.

```
daystart (timestamp [, shift = 0 [, dayoffset = 0]])
```

dayend

This function returns a value corresponding to a timestamp of the final millisecond of the day contained in **time**. The default output format will be the **TimestampFormat** set in the script.

```
dayend (timestamp [, shift = 0 [, dayoffset = 0]])
```

dayname

This function returns a value showing the date with an underlying numeric value corresponding to a timestamp of the first millisecond of the day containing **time**.

```
dayname (timestamp [, shift = 0 [, dayoffset = 0]])
```

Day numbering functions

age

The **age** function returns the age at the time of **timestamp** (in completed years) of somebody born on **date_of_birth**.

```
age (timestamp, date_of_birth)
```

networkdays

The **networkdays** function returns the number of working days (Monday-Friday) between and including **start_date** and **end_date** taking into account any optionally listed **holiday**.

```
networkdays  (start:date, end_date {, holiday})
```

firstworkdate

The **firstworkdate** function returns the latest starting date to achieve **no_of_workdays** (Monday-Friday) ending no later than **end_date** taking into account any optionally listed holidays. **end_date** and **holiday** should be valid dates or timestamps.

```
firstworkdate (end_date, no_of_workdays {, holiday} )
```

lastworkdate

The **lastworkdate** function returns the earliest ending date to achieve **no_of_workdays** (Monday-Friday) if starting at **start_date** taking into account any optionally listed **holiday**. **start_date** and **holiday** should be valid dates or timestamps.

```
lastworkdate (start_date, no_of_workdays {, holiday})
```

daynumberofyear

This function calculates the day number of the year in which a timestamp falls. The calculation is made from the first millisecond of the first day of the year, but the first month can be offset.

```
daynumberofyear (date[,firstmonth])
```

daynumberofquarter

This function calculates the day number of the quarter in which a timestamp falls. This function is used when creating a Master Calendar.

```
daynumberofquarter (date[,firstmonth])
```

addmonths

This function returns the date occurring **n** months after **startdate** or, if **n** is negative, the date occurring **n** months before **startdate**.

Syntax:

```
AddMonths(startdate, n , [ , mode])
```

Return data type: dual

The AddMonths function returns a dual value with both the string and the number value. The function takes the numeric value of the input expression and generates a string representing the number. The string is displayed, whereas the numeric value is used for all numerical calculations and sorting.

Arguments:

Arguments

Argument	Description
startdate	The start date as a time stamp, for example '2012-10-12'.
n	Number of months as a positive or negative integer.
mode	Specifies if the month is added relative to the beginning or to the end of the month. Default mode is 0 for additions relative to the beginning of the month. Set mode to 1 for additions relative to the end of the month. When mode is set to 1 and the input date is the 28th or above, the function checks how many days are left to reach the end of the month on the startdate. The same number of days to reach the end of the month are set on the date returned.

Examples and results:

Scripting examples

Example	Result
`addmonths ('2003-01-29',3)`	returns '2003-04-29'
`addmonths ('2003-01-29',3,0)`	returns '2003-04-29'
`addmonths ('2003-01-29',3,1)`	returns '2003-04-28'
`addmonths ('2003-01-29',1,0)`	returns '2003-02-28'
`addmonths ('2003-01-29',1,1)`	returns '2003-02-26'
`addmonths ('2003-02-28',1,0)`	returns '2003-03-28'
`addmonths ('2003-02-28',1,1)`	returns '2003-03-31'
`addmonths ('2003-01-29',-3)`	returns '2002-10-29'

addyears

This function returns the date occurring **n** years after **startdate** or, if **n** is negative, the date occurring **n** years before **startdate**.

Syntax:

```
AddYears(startdate, n)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
startdate	The start date as a time stamp, for example '2012-10-12'.
n	Number of years as a positive or negative integer.

Examples and results:

Scripting examples

Example	Result
`addyears ('2010-01-29',3)`	returns '2013-01-29'
`addyears ('2010-01-29',-1)`	returns '2009-01-29'

age

The **age** function returns the age at the time of **timestamp** (in completed years) of somebody born on **date_of_birth**.

Syntax:

```
age(timestamp, date_of_birth)
```

Can be an expression.

Return data type: numeric

Arguments:

Arguments

Argument	Description
timestamp	The timestamp,or expression resolving to a timestamp, up to which to calculate the completed number of years.
date_of_birth	Date of birth of the person whose age is being calculated. Can be an expression.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`age('25/01/2014', '29/10/2012')`	Returns 1.
`age('29/10/2014', '29/10/2012')`	Returns 2.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
Employees:
LOAD * INLINE [
Member|DateOfBirth
John|28/03/1989
Linda|10/12/1990
Steve|5/2/1992
Birg|31/3/1993
Raj|19/5/1994
Prita|15/9/1994
Su|11/12/1994
Goran|2/3/1995
Sunny|14/5/1996
Ajoa|13/6/1996
Daphne|7/7/1998
Biffy|4/8/2000
] (delimiter is |);
AgeTable:
Load *,
age('20/08/2015', DateOfBirth) As Age
```

```
Resident Employees;
Drop table Employees;
```

The resulting table shows the returned values of age for each of the records in the table.

Results table

Member	DateOfBirth	Age
John	28/03/1989	26
Linda	10/12/1990	24
Steve	5/2/1992	23
Birg	31/3/1993	22
Raj	19/5/1994	21
Prita	15/9/1994	20
Su	11/12/1994	20
Goran	2/3/1995	20
Sunny	14/5/1996	19
Ajoa	13/6/1996	19
Daphne	7/7/1998	17
Biffy	4/8/2000	15

converttolocaltime

Converts a UTC or GMT timestamp to local time as a dual value. The place can be any of a number of cities, places and time zones around the world.

Syntax:

```
ConvertToLocalTime(timestamp [, place [, ignore_dst=false]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
timestamp	The timestamp,or expression resolving to a timestamp, to convert.

Argument	Description
place	A place or timezone from the table of valid places and timezones below. Alternatively, you can useGMT or UTC to define the local time. The following values and time offset ranges are valid: • GMT • GMT-12:00 - GMT-01:00 • GMT+01:00 - GMT+14:00 • UTC • UTC-12:00 - UTC-01:00 • UTC+01:00 - UTC+14:00 *You can only use standard time offsets. It's not possible to use an arbitrary time offset, for example, GMT-04:27.*
ignore_dst	Set to True if you want to ignore DST (daylight saving time).

The resulting time is adjusted for daylight-saving time, unless **ignore_dst** is set to True.

Valid places and time zones

A-C	D-K	L-R	S-Z
Abu Dhabi	Darwin	La Paz	Samoa
Adelaide	Dhaka	Lima	Santiago
Alaska	Eastern Time (US & Canada)	Lisbon	Sapporo
Amsterdam	Edinburgh	Ljubljana	Sarajevo
Arizona	Ekaterinburg	London	Saskatchewan
Astana	Fiji	Madrid	Seoul
Athens	Georgetown	Magadan	Singapore
Atlantic Time (Canada)	Greenland	Mazatlan	Skopje
Auckland	Greenwich Mean Time : Dublin	Melbourne	Sofia
Azores	Guadalajara	Mexico City	Solomon Is.
Baghdad	Guam	Mid-Atlantic	Sri Jayawardenepura
Baku	Hanoi	Minsk	St. Petersburg
Bangkok	Harare	Monrovia	Stockholm

A-C	D-K	L-R	S-Z
Beijing	Hawaii	Monterrey	Sydney
Belgrade	Helsinki	Moscow	Taipei
Berlin	Hobart	Mountain Time (US & Canada)	Tallinn
Bern	Hong Kong	Mumbai	Tashkent
Bogota	Indiana (East)	Muscat	Tbilisi
Brasilia	International Date Line West	Nairobi	Tehran
Bratislava	Irkutsk	New Caledonia	Tokyo
Brisbane	Islamabad	New Delhi	Urumqi
Brussels	Istanbul	Newfoundland	Warsaw
Bucharest	Jakarta	Novosibirsk	Wellington
Budapest	Jerusalem	Nuku'alofa	West Central Africa
Buenos Aires	Kabul	Osaka	Vienna
Cairo	Kamchatka	Pacific Time (US & Canada)	Vilnius
Canberra	Karachi	Paris	Vladivostok
Cape Verde Is.	Kathmandu	Perth	Volgograd
Caracas	Kolkata	Port Moresby	Yakutsk
Casablanca	Krasnoyarsk	Prague	Yerevan
Central America	Kuala Lumpur	Pretoria	Zagreb
Central Time (US & Canada)	Kuwait	Quito	-
Chennai	Kyiv	Riga	-
Chihuahua	-	Riyadh	-
Chongqing	-	Rome	-
Copenhagen	-	-	-

Examples and results:

Scripting examples

Example	Result
`ConvertToLocalTime('2007-11-10 23:59:00','Paris')`	Returns '2007-11-11 00:59:00' and the corresponding internal timestamp representation.
`ConvertToLocalTime(UTC(), 'GMT-05:00')`	Returns the time for the North American east coast, for example, New York.
`ConvertToLocalTime(UTC(), 'GMT-05:00', True)`	Returns the time for the North American east coast, for example, New York, without daylight-saving time adjustment.

day

This function returns an integer representing the day when the fraction of the **expression** is interpreted as a date according to the standard number interpretation.

The function returns the day of the month for a particular date. It is commonly used to derive a day field as part of a calendar dimension.

Syntax:

```
day(expression)
```

Return data type: integer

Function examples

Example	Result
`day( 1971-10-12 )`	returns 12
`day( 35648 )`	returns 6, because 35648 = 1997-08-06

Example 1 – DateFormat dataset (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates named `Master_Calendar`. The `DateFormat` system variable is set to DD/MM/YYYY.
- A preceding load that creates an additional field, named `day_of_month`, using the `day()` function.
- An additional field, named `long_date`, using the `date()` function to express the full month name.

Load script

```
SET DateFormat='DD/MM/YYYY';

Master_Calendar:
Load
        date,
        date(date,'dd-MMMM-YYYY') as long_date,
        day(date) as day_of_month
Inline
[
date
03/11/2022
03/12/2022
03/13/2022
03/14/2022
03/15/2022
03/16/2022
03/17/2022
03/18/2022
03/19/2022
03/20/2022
03/21/2022
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `long_date`
- `day_of_month`

Results table

date	long_date	day_of_month
03/11/2022	11-March- 2022	11
03/12/2022	12-March- 2022	12
03/13/2022	13-March- 2022	13
03/14/2022	14-March- 2022	14
03/15/2022	15-March- 2022	15
03/16/2022	16-March- 2022	16
03/17/2022	17-March- 2022	17
03/18/2022	18-March- 2022	18
03/19/2022	19-March- 2022	19

date	long_date	day_of_month
03/20/2022	20-March- 2022	20
03/21/2022	21-March- 2022	21

The day of the month is correctly evaluated by the `day()` function in the script.

Example 2 – ANSI dates (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates named `Master_Calendar`. The DateFormat system variable DD/MM/YYYY is used. However, the dates that are included in the dataset are in ANSI standard date format.
- A preceding load that creates an additional field, named `day_of_month`, using the `date()` function.
- An additional field, named `long_date`, using the `date()` function to express the date with the full month name.

Load script

```
SET DateFormat='DD/MM/YYYY';
Master_Calendar:
Load
       date,
       date(date,'dd-MMMM-YYYY') as long_date,
       day(date) as day_of_month

Inline
[
date
2022-03-11
2022-03-12
2022-03-13
2022-03-14
2022-03-15
2022-03-16
2022-03-17
2022-03-18
2022-03-19
2022-03-20
2022-03-21
 ];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- date
- long_date
- day_of_month

Results table

date	long_date	day_of_month
03/11/2022	11-March- 2022	11
03/12/2022	12-March- 2022	12
03/13/2022	13-March- 2022	13
03/14/2022	14-March- 2022	14
03/15/2022	15-March- 2022	15
03/16/2022	16-March- 2022	16
03/17/2022	17-March- 2022	17
03/18/2022	18-March- 2022	18
03/19/2022	19-March- 2022	19
03/20/2022	20-March- 2022	20
03/21/2022	21-March- 2022	21

The day of the month is correctly evaluated by the `day()` function in the script.

Example 3 – Unformatted dates (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates named `Master_Calendar`. The `DateFormat` system variable DD/MM/YYYY is used.
- A preceding load that creates an additional field, named `day_of_month`, using the `day()` function.
- The original unformatted date, named `unformatted_date`.
- An additional field, named `long_date`, using the `date()` is used to convert the numerical date into a formatted date field.

Load script

```
SET DateFormat='DD/MM/YYYY';

Master_Calendar:
Load
```

```
      unformatted_date,
      date(unformatted_date,'dd-MMMM-YYYY') as long_date,
      day(date) as day_of_month

Inline
[
unformatted_date
44868
44898
44928
44958
44988
45018
45048
45078
45008
45038
45068
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `unformatted_date`
- `long_date`
- `day_of_month`

Results table

unformatted_date	long_date	day_of_month
44868	03-November- 2022	3
44898	03-December- 2022	3
44928	02-January- 2023	2
44958	01-February- 2023	1
44988	03-March- 2023	3
45008	23-March- 2023	23
45018	02-April- 2023	2
45038	22-April- 2023	22
45048	02-May- 2023	2
45068	22-May- 2023	22
45078	01-June- 2023	1

The day of the month is correctly evaluated by the `day()` function in the script.

Example 4 – Calculating expiry month (chart)

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of orders placed in March named `Orders`. The table contains three fields:
  - id
  - order_date
  - amount

Load script

```
Orders:
Load
        id,
        order_date,
        amount
Inline
[
id,order_date,amount
1,03/01/2022,231.24
2,03/02/2022,567.28
3,03/03/2022,364.28
4,03/04/2022,575.76
5,03/05/2022,638.68
6,03/06/2022,785.38
7,03/07/2022,967.46
8,03/08/2022,287.67
9,03/09/2022,764.45
10,03/10/2022,875.43
11,03/11/2022,957.35
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension: `order_date`.

To calculate the delivery date, create this measure: `=day(order_date+5)`.

Results table

order_date	=day(order_date+5)
03/11/2022	16

order_date	=day(order_date+5)
03/12/2022	17
03/13/2022	18
03/14/2022	19
03/15/2022	20
03/16/2022	21
03/17/2022	22
03/18/2022	23
03/19/2022	24
03/20/2022	25
03/21/2022	26

The `day()` function correctly determines that an order placed on the 11th of March would be delivered on the 16th based on a 5 day delivery period.

dayend

This function returns a value corresponding to a timestamp of the final millisecond of the day contained in **time**. The default output format will be the **TimestampFormat** set in the script.

Syntax:

```
DayEnd(time[, [period_no[, day_start]])
```

When to use it

The `dayend()` function is commonly used as part of an expression when the user would like the calculation to use the fraction of the day that has not yet occurred. For example, to calculate the total expenses still to be incurred during the day.

Return data type: dual

Arguments

Argument	Description
time	The timestamp to evaluate.
period_no	**period_no** is an integer, or expression that resolves to an integer, where the value 0 indicates the day that contains **time**. Negative values in **period_no** indicate preceding days and positive values indicate succeeding days.
day_start	To specify days not starting at midnight, indicate an offset as a fraction of a day in **day_start**. For example, 0.125 to denote 3:00 AM. In other words, to create the offset, divide the start time by 24 hours. For example, for a day to begin at 7:00 AM, use the fraction 7/24.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Function examples

Example	Result
dayend('01/25/2013 16:45:00')	Returns 01/25/2013 23:59:59. PM
dayend('01/25/2013 16:45:00', -1)	Returns 01/24/2013 23:59:59. PM
dayend('01/25/2013 16:45:00', 0, 0.5)	Returns 01/26/2013 11:59:59. PM

Example 1 - Basic script

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a list of dates is loaded into a table named "Calendar".
- The default `DateFormat` system variable `(MM/DD/YYYY)`.
- A preceding load to create an additional field, '`EOD_timestamp`', using the `dayend()` function.

Load script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Calendar:
    Load
      date,
      dayend(date) as EOD_timestamp
      ;
Load
date
Inline
[
date
03/11/2022 1:47:15 AM
```

```
03/12/2022 4:34:58 AM
03/13/2022 5:15:55 AM
03/14/2022 9:25:14 AM
03/15/2022 10:06:54 AM
03/16/2022 10:44:42 AM
03/17/2022 11:33:30 AM
03/18/2022 12:58:14 PM
03/19/2022 4:23:12 PM
03/20/2022 6:42:15 PM
03/21/2022 7:41:16 PM
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `EOD_timestamp`

Results table

date	EOD_timestamp
03/11/2022 1:47:15 AM	3/11/2022 11:59:59 PM
03/12/2022 4:34:58 AM	3/12/2022 11:59:59 PM
03/13/2022 5:15:55 AM	3/13/2022 11:59:59 PM
03/14/2022 9:25:14 AM	3/14/2022 11:59:59 PM
03/15/2022 10:06:54 AM	3/15/2022 11:59:59 PM
03/16/2022 10:44:42 AM	3/16/2022 11:59:59 PM
03/17/2022 11:33:30 AM	3/17/2022 11:59:59 PM
03/18/2022 12:58:14 PM	3/18/2022 11:59:59 PM
03/19/2022 4:23:12 PM	3/19/2022 11:59:59 PM
03/20/2022 6:42:15 PM	3/20/2022 11:59:59 PM
03/21/2022 7:41:16 PM	3/21/2022 11:59:59 PM

As you can see in the table above, the end of day timestamp is generated for each date in our dataset. The timestamp is in the format of the system variable `TimestampFormat M/D/YYYY h:mm:ss[.fff] TT`.

Example 2 – period_no

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

You will load a dataset containing service bookings into a table named 'Services'.

The dataset includes the following fields:

- service_id
- service_date
- amount

You will create two new fields in the table:

- deposit_due_date: The date when the deposit should be received. This is the end of the day three days before the service_date.
- final_payment_due_date: The date when the final payment should be received. This is the end of the day seven days after the service_date.

The two fields above are created in a preceding load using the dayend() function and they supply the first two parameters, time and period_no.

Load script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Services:
   Load
     *,
     dayend(service_date,-3) as deposit_due_date,
     dayend(service_date,7) as final_payment_due_date
     ;
Load
service_id,
service_date,
amount
Inline
[
service_id, service_date,amount
1,03/11/2022 9:25:14 AM,231.24
2,03/12/2022 10:06:54 AM,567.28
3,03/13/2022 10:44:42 AM,364.28
4,03/14/2022 11:33:30 AM,575.76
5,03/15/2022 12:58:14 PM,638.68
6,03/16/2022 4:23:12 PM,785.38
7,03/17/2022 6:42:15 PM,967.46
8,03/18/2022 7:41:16 PM,287.67
9,03/19/2022 8:14:15 PM,764.45
10,03/20/2022 9:23:51 PM,875.43
11,03/21/2022 10:04:41 PM,957.35
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- service_date
- deposit_due_date
- final_payment_due_date

Results table

service_date	deposit_due_date	final_payment_due_date
03/11/2022 9:25:14 AM	3/8/2022 11:59:59 PM	3/18/2022 11:59:59 PM
03/12/2022 10:06:54 AM	3/9/2022 11:59:59 PM	3/19/2022 11:59:59 PM
03/13/2022 10:44:42 AM	3/10/2022 11:59:59 PM	3/20/2022 11:59:59 PM
03/14/2022 11:33:30 AM	3/11/2022 11:59:59 PM	3/21/2022 11:59:59 PM
03/15/2022 12:58:14 PM	3/12/2022 11:59:59 PM	3/22/2022 11:59:59 PM
03/16/2022 4:23:12 PM	3/13/2022 11:59:59 PM	3/23/2022 11:59:59 PM
03/17/2022 6:42:15 PM	3/14/2022 11:59:59 PM	3/24/2022 11:59:59 PM
03/18/2022 7:41:16 PM	3/15/2022 11:59:59 PM	3/25/2022 11:59:59 PM
03/19/2022 8:14:15 PM	3/16/2022 11:59:59 PM	3/26/2022 11:59:59 PM
03/20/2022 9:23:51 PM	3/17/2022 11:59:59 PM	3/27/2022 11:59:59 PM
03/21/2022 10:04:41 PM	3/18/2022 11:59:59 PM	3/28/2022 11:59:59 PM

The values of the new fields are in the `TimestampFormat M/D/YYYY h:mm:ss[.fff] TT`. Because the function `dayend()` was used, the timestamp values are all the last millisecond of the day.

The deposit due date values are three days before the service date because the second argument passed in the `dayend()` function is negative.

The final payment due date values are seven days after the service date because the second argument passed in the `dayend()` function is positive.

Example 3 – day_start script

Load script and results

Overview

Open the Data load editor and add the load script below in a new tab.

The dataset and scenario used in this example is the same as in the previous example.

As in the previous example, you will create two new fields:

- `deposit_due_date`: The date when the deposit should be received. This is the end of the day three days before the `service_date`.
- `final_payment_due_date`: The date when the final payment should be received. This is the end of the day seven days after the `service_date`.

However, your company would like to operate under a policy where the working day begins at 5 PM and ends at 5 PM the following day. Your company can then monitor transactions that occur in those working hours.

To achieve these requirements, the two fields above are created in a preceding load using the `dayend()` function and use all three arguments, `time`, `period_no`, and `day_start`.

Load Script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Services:
   Load
     *,
     dayend(service_date,-3,17/24) as deposit_due_date,
     dayend(service_date,7,17/24) as final_payment_due_date
     ;
Load
service_id,
service_date,
amount
Inline
[
service_id, service_date,amount
1,03/11/2022 9:25:14 AM,231.24
2,03/12/2022 10:06:54 AM,567.28
3,03/13/2022 10:44:42 AM,364.28
4,03/14/2022 11:33:30 AM,575.76
5,03/15/2022 12:58:14 PM,638.68
6,03/16/2022 4:23:12 PM,785.38
7,03/17/2022 6:42:15 PM,967.46
8,03/18/2022 7:41:16 PM,287.67
9,03/19/2022 8:14:15 PM,764.45
10,03/20/2022 9:23:51 PM,875.43
11,03/21/2022 10:04:41 PM,957.35
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `service_date`
- `deposit_due_date`
- `final_payment_due_date`

Results table

service_date	deposit_due_date	final_payment_due_date
03/11/2022 9:25:14 AM	3/8/2022 4:59:59 PM	3/18/2022 4:59:59 PM
03/12/2022 10:06:54 AM	3/9/2022 4:59:59 PM	3/19/2022 4:59:59 PM

service_date	deposit_due_date	final_payment_due_date
03/13/2022 10:44:42 AM	3/10/2022 4:59:59 PM	3/20/2022 4:59:59 PM
03/14/2022 11:33:30 AM	3/11/2022 4:59:59 PM	3/21/2022 4:59:59 PM
03/15/2022 12:58:14 PM	3/12/2022 4:59:59 PM	3/22/2022 4:59:59 PM
03/16/2022 4:23:12 PM	3/13/2022 4:59:59 PM	3/23/2022 4:59:59 PM
03/17/2022 6:42:15 PM	3/14/2022 4:59:59 PM	3/24/2022 4:59:59 PM
03/18/2022 7:41:16 PM	3/15/2022 4:59:59 PM	3/25/2022 4:59:59 PM
03/19/2022 8:14:15 PM	3/16/2022 4:59:59 PM	3/26/2022 4:59:59 PM
03/20/2022 9:23:51 PM	3/17/2022 4:59:59 PM	3/27/2022 4:59:59 PM
03/21/2022 10:04:41 PM	3/18/2022 4:59:59 PM	3/28/2022 4:59:59 PM

While the dates remain the same as in Example 2, the dates now have a timestamp of the last millisecond before 5:00 PM because the value of the third argument, `day_start`, passed into the `dayend()` function is 17/24.

Example 4 – Chart example

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The dataset and scenario used in this example is the same as in the previous two examples. The company would like to operate under a policy where the working day begins at 5:00 PM and ends at 5:00 PM the following day.

As in the previous example, you will create two new fields:

- `deposit_due_date`: The date when the deposit should be received. This is the end of the day three days before the `service_date`.
- `final_payment_due_date`: The date when the final payment should be received. This is the end of the day seven days after the `service_date`.

Load Script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Services:
Load
service_id,
service_date,
amount
Inline
[
service_id, service_date,amount
```

```
1,03/11/2022 9:25:14 AM,231.24
2,03/12/2022 10:06:54 AM,567.28
3,03/13/2022 10:44:42 AM,364.28
4,03/14/2022 11:33:30 AM,575.76
5,03/15/2022 12:58:14 PM,638.68
6,03/16/2022 4:23:12 PM,785.38
7,03/17/2022 6:42:15 PM,967.46
8,03/18/2022 7:41:16 PM,287.67
9,03/19/2022 8:14:15 PM,764.45
10,03/20/2022 9:23:51 PM,875.43
11,03/21/2022 10:04:41 PM,957.35
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension:

`service_date`.

To create the `deposit_due_date` field, create this measure:

`=dayend(service_date,-3,17/24)`.

Then, to create the `final_payment_due_date` field, create this measure:

`=dayend(service_date,7,17/24)`.

Results table

service_date	=dayend(service_date,-3,17/24)	=dayend(service_date,7,17/24
03/11/2022	3/8/2022 16:59:59 PM	3/18/2022 16:59:59 PM
03/12/2022	3/9/2022 16:59:59 PM	3/19/2022 16:59:59 PM
03/13/2022	3/10/2022 16:59:59 PM	3/20/2022 16:59:59 PM
03/14/2022	3/11/2022 16:59:59 PM	3/21/2022 16:59:59 PM
03/15/2022	3/12/2022 16:59:59 PM	3/22/2022 16:59:59 PM
03/16/2022	3/13/2022 16:59:59 PM	3/23/2022 16:59:59 PM
03/17/2022	3/14/2022 16:59:59 PM	3/24/2022 16:59:59 PM
03/18/2022	3/15/2022 16:59:59 PM	3/25/2022 16:59:59 PM
03/19/2022	3/16/2022 16:59:59 PM	3/26/2022 16:59:59 PM
03/20/2022	3/17/2022 16:59:59 PM	3/27/2022 16:59:59 PM
03/21/2022	3/18/2022 16:59:59 PM	3/28/2022 16:59:59 PM

The values of the new fields are in the `TimestampFormat M/D/YYYY h:mm:ss[.fff] TT`. Because the function `dayend()` was used, the timestamp values are all the last millisecond of the day.

The payment due date values are three days before the service date because the second argument passed in the `dayend()` function is negative.

The final payment due date values are seven days after the service date because the second argument passed in the `dayend()` function is positive.

The dates have a timestamp of the last millisecond before 5:00 PM because the value of the third argument, `day_start`,that passed into the `dayend()` function is 17/24.

daylightsaving

Returns the current adjustment for daylight saving time, as defined in Windows.

Syntax:

```
DaylightSaving( )
```

Return data type: dual

Example:

```
daylightsaving( )
```

dayname

This function returns a value showing the date with an underlying numeric value corresponding to a timestamp of the first millisecond of the day containing **time**.

Syntax:

```
DayName(time[, period_no [, day_start]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
time	The timestamp to evaluate.
period_no	**period_no** is an integer, or expression that resolves to an integer, where the value 0 indicates the day that contains **time**. Negative values in **period_no** indicate preceding days and positive values indicate succeeding days.
day_start	To specify days not starting at midnight, indicate an offset as a fraction of a day in **day_start**. For example, 0.125 to denote 3:00 AM.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`dayname('25/01/2013 16:45:00')`	Returns 25/01/2013.
`dayname('25/01/2013 16:45:00', -1)`	Returns 24/01/2013.
`dayname('25/01/2013 16:45:00', 0, 0.5 )`	Returns 25/01/2013. Displaying the full timestamp shows the underlying numeric value corresponds to '25/01/2013 12:00:00.000.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

In this example, the day name is created from the timestamp that marks the beginning of the day after each invoice date in the table.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
DayName(InvDate, 1) AS DName
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the dayname() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	DName
28/03/2012	29/03/2012 00:00:00
10/12/2012	11/12/2012 00:00:00

InvDate	DName
5/2/2013	07/02/2013 00:00:00
31/3/2013	01/04/2013 00:00:00
19/5/2013	20/05/2013 00:00:00
15/9/2013	16/09/2013 00:00:00
11/12/2013	12/12/2013 00:00:00
2/3/2014	03/03/2014 00:00:00
14/5/2014	15/05/2014 00:00:00
13/6/2014	14/06/2014 00:00:00
7/7/2014	08/07/2014 00:00:00
4/8/2014	05/08/2014 00:00:00

daynumberofquarter

This function calculates the day number of the quarter in which a timestamp falls. This function is used when creating a Master Calendar.

Syntax:

```
DayNumberOfQuarter(timestamp[,start_month])
```

Return data type: integer

Arguments

Argument	Description
timestamp	The date to evaluate.
start_month	By specifying a **start_month** between 2 and 12 (1, if omitted), the beginning of the year may be moved forward to the first day of any month. For example, if you want to work with a fiscal year starting March 1, specify **start_month** = 3.

These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Function examples

Example	Result
`DayNumberOfQuarter('12/09/2014')`	Returns 74, the day number of the current quarter.
`DayNumberOfQuarter ('12/09/2014',3)`	Returns 12, the day number of the current quarter. In this case, the first quarter starts with March (because start_ month is specified as 3). This means that the current quarter is the third quarter, which started on September 1.

Example 1 – January start of year (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A simple dataset containing a list of dates, which is loaded into a table named `Calendar`. The default `DateFormat` system variable MM/DD/YYYY is used.
- A preceding load that creates an additional field, named `DayNrQtr`, using the `DayNumberOfQuarter()` function.

Aside from the date, no additional parameters are provided to the function.

Load script

```
SET DateFormat='MM/DD/YYYY';

Calendar:
Load
        date,
        DayNumberOfQuarter(date) as DayNrQtr
        ;
Load
date
Inline
[
date
01/01/2022
01/10/2022
01/31/2022
02/01/2022
02/10/2022
02/28/2022
03/01/2022
03/31/2022
04/01/2022
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `daynrqtr`

Results table

date	daynrqtr
01/01/2022	1
01/10/2022	10
01/31/2022	31
02/01/2022	32
02/10/2022	41
02/28/2022	59
03/01/2022	61
03/31/2022	91
04/01/2022	1

The first day of the year is January 1 because no second argument was passed into the `DayNumberOfQuarter()` function.

January 1st is the 1st day of the quarter whilst February 1st is the 32nd day of the quarter. The 31st of March is the 91st and final day of the quarter, whilst the 1st of April is the 1st day of the 2nd Quarter.

Example 2 – February start of year (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset from the first example.
- The default `DateFormat` system variable MM/DD/YYYY is used.
- A `start_month` argument beginning on February 1. This sets the financial year to February 1.

Load script

```
SET DateFormat='MM/DD/YYYY';

Calendar:
Load
       date,
       DayNumberOfQuarter(date,2) as DayNrQtr
       ;
Load
date
Inline
[
```

```
date
01/01/2022
01/10/2022
01/31/2022
02/01/2022
02/10/2022
02/28/2022
03/01/2022
03/31/2022
04/01/2022
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `daynrqtr`

Results table

date	daynrqtr
01/01/2022	62
01/10/2022	71
01/31/2022	92
02/01/2022	1
02/10/2022	10
02/28/2022	28
03/01/2022	30
03/31/2022	60
04/01/2022	61

The first day of the year is the 1st of February because the second argument passed into the `DayNumberOfQuarter()` function was 2.

The first quarter of the year operates between February and April whilst the fourth quarter operates between November and January. This is shown in the results table where February 1st is the 1st day of the quarter whilst January 31st is the 92nd and last day of the quarter.

Example 3 – January start of year (chart)

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset from the first example.
- The default `DateFormat` system variable MM/DD/YYYY is used.

However, in this example, the unchanged dataset is loaded into the application. The value of the day of the quarter is calculated via a measure in a chart object.

Load script

```
SET DateFormat='MM/DD/YYYY';

Calendar:
Load
date
Inline
[
date
01/01/2022
01/10/2022
01/31/2022
02/01/2022
02/10/2022
02/28/2022
03/01/2022
03/31/2022
04/01/2022
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension: `date`.

Create the following measure:

```
=daynumberofquarter(date)
```

Results table

date	=daynumberofquarter(date)
01/01/2022	1
01/10/2022	10
01/31/2022	31
02/01/2022	32
02/10/2022	41
02/28/2022	59
03/01/2022	61
03/31/2022	91
04/01/2022	1

The first day of the year is the 1st of January because no second argument passed into the `DayNumberOfQuarter()` function.

January 1st is the 1st day of the quarter whilst February 1st is the 32nd day of the quarter. The 31st of March is the 91st and final day of the quarter, whilst the 1st of April is the 1st day of the 2nd Quarter.

Example 4 – February start of year (chart)

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset from the first example.
- The default `DateFormat` system variable MM/DD/YYYY is used.
- The financial year runs from the 1st of February to the 31st of January.

However, in this example, the unchanged dataset is loaded into the application. The value of the day of the quarter is calculated via a measure in a chart object.

Load script

```
SET DateFormat='MM/DD/YYYY';

Calendar:
Load
date
Inline
[
date
01/01/2022
01/10/2022
01/31/2022
02/01/2022
02/10/2022
02/28/2022
03/01/2022
03/31/2022
04/01/2022
];
```

Chart object

Load the data and open a sheet. Create a new table and add this field as a dimension: `date`.

Create the following measure:

```
=daynumberofquarter(date,2)
```

Results

Results table

date	=daynumberofquarter(date,2)
01/01/2022	62
01/10/2022	71
01/31/2022	92
02/01/2022	1
02/10/2022	10
02/28/2022	28
03/01/2022	30
03/31/2022	60
04/01/2022	61

The first day of the year is the 1st of January because the second argument passed into the `DayNumberOfQuarter()` function was 2.

The first quarter of the year operates between February and April whilst the fourth quarter operates between November and January. This is evidenced in the results table where February 1st is the 1st day of the quarter whilst January 31st is the 92nd and last day of the quarter.

daynumberofyear

This function calculates the day number of the year in which a timestamp falls. The calculation is made from the first millisecond of the first day of the year, but the first month can be offset.

Syntax:

```
DayNumberOfYear(timestamp[,start_month])
```

Return data type: integer

Arguments

Argument	Description
timestamp	The date to evaluate.
start_month	By specifying a **start_month** between 2 and 12 (1, if omitted), the beginning of the year may be moved forward to the first day of any month. For example, if you want to work with a fiscal year starting March 1, specify **start_month** = 3.

These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Function examples

Example	Result
`DayNumberOfYear( '12/09/2014' )`	Returns 256, the day number counted from the first of the year.
`DayNumberOfYear( '12/09/2014',3 )`	Returns 196, the number of the day, as counted from 1 March.

Example 1 – January start of year (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A simple dataset containing a list of dates, which is loaded into a table named `Calendar`. The default `DateFormat` system variable MM/DD/YYYY is used.
- A preceding load that creates an additional field, named `daynryear`, using the `DayNumberOfYear()` function.

Aside from the date, no additional parameters are provided to the function.

Load script

```
SET DateFormat='MM/DD/YYYY';

Calendar:
Load
       date,
       DayNumberOfYear(date) as daynryear
       ;
Load
date
Inline
[
date
01/01/2022
01/10/2022
01/31/2022
02/01/2022
02/10/2022
06/30/2022
07/26/2022
10/31/2022
11/01/2022
12/31/2022
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `daynryear`

Results table

date	daynryear
01/01/2022	1
01/10/2022	10
01/31/2022	31
02/01/2022	32
02/10/2022	41
06/30/2022	182
07/26/2022	208
10/31/2022	305
11/01/2022	306
12/31/2022	366

The first day of the year is the 1st of January because no second argument was passed into the `DayNumberOfYear()` function.

January 1st is the 1st day of the quarter whilst February 1st is the 32nd day of the year. The 30th of June is the 182nd whilst the 31st of December is the 366th and final day of the year.

Example 2 – November start of year (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset from the first example.
- The default `DateFormat` system variable MM/DD/YYYY is used
- A `start_month` argument beginning on November 1. This sets the financial year to November 1.

Load script

```
SET DateFormat='MM/DD/YYYY';

Calendar:
Load
       date,
       DayNumberOfYear(date,11) as daynryear
       ;
Load
date
Inline
[
date
01/01/2022
01/10/2022
01/31/2022
02/01/2022
02/10/2022
06/30/2022
07/26/2022
10/31/2022
11/01/2022
12/31/2022
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `daynryear`

Results table

date	daynryear
01/01/2022	62
01/10/2022	71
01/31/2022	92
02/01/2022	93
02/10/2022	102
06/30/2022	243
07/26/2022	269
10/31/2022	366
11/01/2022	1
12/31/2022	61

The first day of the year is the 1st of November because the second argument passed into the `DayNumberOfYear()` function was 11.

January 1st is the 1st day of the quarter whilst February 1st is the 32nd day of the year. The 30th of June is the 182nd whilst the 31st of December is the 366th and final day of the year.

Example 3 – January start of year (chart)

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset from the first example.
- The default `DateFormat` system variable MM/DD/YYYY is used.

However, in this example, the unchanged dataset is loaded into the application. The value of the day of the quarter is calculated via a measure in a chart object.

Load script

```
SET DateFormat='MM/DD/YYYY';

Calendar:
Load
date
Inline
[
date
01/01/2022
01/10/2022
01/31/2022
02/01/2022
02/10/2022
06/30/2022
07/26/2022
10/31/2022
11/01/2022
12/31/2022
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension: `date`.

Create the following measure:

```
=daynumberofyear(date)
```

Results table

date	=daynumberofyear(date)
01/01/2022	1
01/10/2022	10
01/31/2022	31
02/01/2022	32
02/10/2022	41
06/30/2022	182
07/26/2022	208
10/31/2022	305
11/01/2022	306
12/31/2022	366

The first day of the year is the 1st of January because no second argument was passed into the `DayNumberOfYear()` function.

January 1st is the 1st day of the year whilst February 1st is the 32nd day of the year. The 30th of June is the 182nd whilst the 31st of December is the 366th and final day of the year.

Example 4 – November start of year (chart)

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset from the first example.
- The default `DateFormat` system variable MM/DD/YYYY is used.
- The financial year runs from the 1st of November to the 31st of October.

However, in this example, the unchanged dataset is loaded into the application. The value of the day of the year is calculated via a measure in a chart object.

Load script

```
SET DateFormat='MM/DD/YYYY';
Calendar:
Load
date
Inline
[
```

```
date
01/01/2022
01/10/2022
01/31/2022
02/01/2022
02/10/2022
06/30/2022
07/26/2022
10/31/2022
11/01/2022
12/31/2022
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension: `date`.

Create the following measure:

```
=daynumberofyear(date)
```

Results table

date	=daynumberofyear(date,11)
01/01/2022	62
01/10/2022	71
01/31/2022	92
02/01/2022	93
02/10/2022	102
06/30/2022	243
07/26/2022	269
10/31/2022	366
11/01/2022	1
12/31/2022	61

The first day of the year is the 1st of November because the second argument passed into the `DayNumberOfYear()` function was 11.

The financial year operates between November and October. This is shown in the results table where November 1st is the 1st day of the year whilst October 31st is the 366th and last day of the year.

daystart

This function returns a value corresponding to a timestamp with the first millisecond of the day contained in the **time** argument. The default output format will be the **TimestampFormat** set in the script.

Syntax:

```
DayStart(time[, [period_no[, day_start]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
time	The timestamp to evaluate.
period_no	**period_no** is an integer, or expression that resolves to an integer, where the value 0 indicates the day that contains **time**. Negative values in **period_no** indicate preceding days and positive values indicate succeeding days.
day_start	To specify days not starting at midnight, indicate an offset as a fraction of a day in **day_start**. For example, 0.125 to denote 3:00 AM.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`daystart('25/01/2013 16:45:00')`	Returns 25/01/2013 00:00:00.
`daystart('25/01/2013 16:45:00', -1)`	Returns 24/01/2013 00:00:00.
`daystart('25/01/2013 16:45:00', 0, 0.5 )`	Returns 25/01/2013 12:00:00.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example finds the timestamp that marks the beginning of the day after each invoice date in the table.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
```

```
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
DayStart(InvDate, 1) AS DStart
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the daystart() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	DStart
28/03/2012	29/03/2012 00:00:00
10/12/2012	11/12/2012 00:00:00
5/2/2013	07/02/2013 00:00:00
31/3/2013	01/04/2013 00:00:00
19/5/2013	20/05/2013 00:00:00
15/9/2013	16/09/2013 00:00:00
11/12/2013	12/12/2013 00:00:00
2/3/2014	03/03/2014 00:00:00
14/5/2014	15/05/2014 00:00:00
13/6/2014	14/06/2014 00:00:00
7/7/2014	08/07/2014 00:00:00
4/8/2014	05/08/2014 00:00:00

firstworkdate

The **firstworkdate** function returns the latest starting date to achieve **no_of_workdays** (Monday-Friday) ending no later than **end_date** taking into account any optionally listed holidays. **end_date** and **holiday** should be valid dates or timestamps.

Syntax:

```
firstworkdate(end_date, no_of_workdays {, holiday} )
```

Return data type: integer

Arguments:

Arguments

Argument	Description
end_date	The timestamp of end date to evaluate.
no_of_ workdays	The number of working days to achieve.
holiday	Holiday periods to exclude from working days. A holiday period is stated as a start date and an end date, separated by commas. **Example:** `'25/12/2013', '26/12/2013'` You can specify more than one holiday period, separated by commas. **Example:** `'25/12/2013', '26/12/2013', '31/12/2013', '01/01/2014'` Holiday periods to exclude from working days. A holiday is stated as a string constant date. You can specify multiple holiday dates, separated by commas.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`firstworkdate ('29/12/2014', 9)`	Returns '17/12/2014.
`firstworkdate ('29/12/2014', 9, '25/12/2014', '26/12/2014')`	Returns 15/12/2014 because a holiday period of two days is taken into account.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
ProjectTable:
LOAD *, recno() as InvID, INLINE [
EndDate
28/03/2015
10/12/2015
5/2/2016
31/3/2016
19/5/2016
15/9/2016
] ;
```

```
NrDays:
Load *,
FirstWorkDate(EndDate,120) As StartDate
Resident ProjectTable;
Drop table ProjectTable;
```

The resulting table shows the returned values of FirstWorkDate for each of the records in the table.

Results table

InvID	EndDate	StartDate
1	28/03/2015	13/10/2014
2	10/12/2015	26/06/2015
3	5/2/2016	24/08/2015
4	31/3/2016	16/10/2015
5	19/5/2016	04/12/2015
6	15/9/2016	01/04/2016

GMT

This function returns the current Greenwich Mean Time, as derived from the regional settings.

Syntax:

```
GMT( )
```

Return data type: dual

Example:

```
gmt( )
```

hour

This function returns an integer representing the hour when the fraction of the **expression** is interpreted as a time according to the standard number interpretation.

Syntax:

```
hour(expression)
```

Return data type: integer

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may

be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Function examples

Example	Result
`hour( '09:14:36' )`	The text string supplied is implicitly converted to a timestamp as it matches the timestamp format defined in the TimestampFormat variable. The expression returns 9.
`hour( '0.5555' )`	The expression returns 13 (Because 0.5555 = 13:19:55).

Example 1 – Variable (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing transactions by timestamp
- The default `TimeStamp` system variable `(M/D/YYYY h:mm:ss[.fff] TT)`

Create a field, '`hour`', calculating when purchases took place.

Load script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Transactions:
  Load
    *,
    hour(date) as hour
    ;
Load
*
Inline
[
id,date,amount
9497,'2022-01-05 19:04:57',47.25,
9498,'2022-01-03 14:21:53',51.75,
9499,'2022-01-03 05:40:49',73.53,
9500,'2022-01-04 18:49:38',15.35,
```

```
9501,'2022-01-01 22:10:22',31.43,
9502,'2022-01-05 19:34:46',13.24,
9503,'2022-01-04 22:58:34',74.34,
9504,'2022-01-06 11:29:38',50.00,
9505,'2022-01-02 08:35:54',36.34,
9506,'2022-01-06 08:49:09',74.23
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `hour`

Results table

date	hour
2022-01-01 22:10:22	22
2022-01-02 08:35:54	8
2022-01-03 05:40:49	5
2022-01-03 14:21:53	14
2022-01-04 18:49:38	18
2022-01-04 22:58:34	22
2022-01-05 19:04:57	19
2022-01-05 19:34:46	19
2022-01-06 08:49:09	8
2022-01-06 11:29:38	11

The values in the hour field are created by using the `hour()` function and passing the date as the expression in the preceding load statement.

Example 2 – Chart object (chart)

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset from the first example.
- The default `TimeStamp` system variable `(M/D/YYYY h:mm:ss[.fff] TT)`.

However, in this example, the dataset, unchanged, is loaded into the application. The 'hour' values are calculated via a measure in a chart object.

Load Script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Transactions:
Load
*
Inline
[
id,date,amount
9497,'2022-01-05 19:04:57',47.25,
9498,'2022-01-03 14:21:53',51.75,
9499,'2022-01-03 05:40:49',73.53,
9500,'2022-01-04 18:49:38',15.35,
9501,'2022-01-01 22:10:22',31.43,
9502,'2022-01-05 19:34:46',13.24,
9503,'2022-01-04 22:58:34',74.34,
9504,'2022-01-06 11:29:38',50.00,
9505,'2022-01-02 08:35:54',36.34,
9506,'2022-01-06 08:49:09',74.23
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension: `date`.

To calculate the 'hour', create the following measure:

```
=hour(date)
```

Results table

due_date	=hour(date)
2022-01-01 22:10:22	22
2022-01-02 08:35:54	8
2022-01-03 05:40:49	5
2022-01-03 14:21:53	14
2022-01-04 18:49:38	18
2022-01-04 22:58:34	22
2022-01-05 19:04:57	19
2022-01-05 19:34:46	19
2022-01-06 08:49:09	8
2022-01-06 11:29:38	11

The values for 'hour' are created by using the `hour()` function and passing the date as the expression in a measure for the chart object.

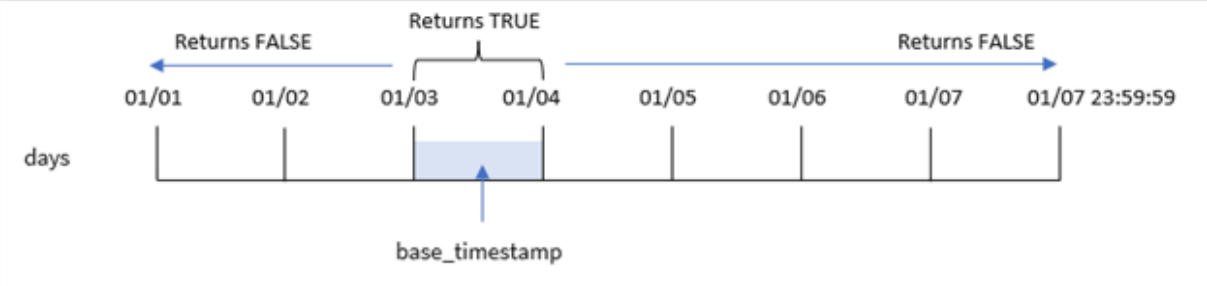
inday

This function returns True if **timestamp** lies inside the day containing **base_timestamp**.

Syntax:

```
InDay (timestamp, base_timestamp, period_no[, day_start])
```

Diagram of `inday` function



The `inday()` function uses the `base_timestamp` argument to identify which day the timestamp falls into. The start time of the day is, by default, midnight; but you can change the start time of the day by using the `day_start` argument of the `inday()` function. Once this day is defined, the function will return Boolean results when comparing the prescribed timestamp values to that day.

When to use it

The `inday()` function returns a Boolean result. Typically, this type of function will be used as a condition in an `if expression`. This returns an aggregation or calculation dependent on whether a date evaluated occurred in the day of the timestamp in question.

For example, the `inday()` function can be used to identify all equipment manufactured in a given day.

Return data type: Boolean

In Qlik Sense, the Boolean true value is represented by -1, and the false value is represented by 0.

Arguments

Argument	Description
`timestamp`	The date and time that you want to compare with `base_timestamp`.
`base_timestamp`	Date and time that is used to evaluate the timestamp.
`period_no`	The day can be offset by `period_no`. `period_no` is an integer, where the value 0 indicates the day which contains `base_timestamp`. Negative values in `period_no` indicate preceding days and positive values indicate succeeding days.
`day_start`	If you want to work with days not starting midnight, indicate an offset as a fraction of a day in `day_start`, For example, 0.125 to denote 3 AM.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Function examples

Example	Result
`inday` ('01/12/2006 12:23:00 PM', '01/12/2006 12:00:00 AM', 0)	Returns True
`inday` ('01/12/2006 12:23:00 PM', '01/13/2006 12:00:00 AM', 0)	Returns False
`inday` ('01/12/2006 12:23:00 PM', '01/12/2006 12:00:00 AM', -1)	Returns False
`inday` ('01/11/2006 12:23:00 PM', '01/12/2006 12:00:00 AM', -1)	Returns True
`inday` ('01/12/2006 12:23:00 PM', '01/12/2006 12:00:00 AM', 0, 0.5)	Returns False
`inday` ('01/12/2006 11:23:00 AM', '01/12/2006 12:00:00 AM', 0, 0.5)	Returns True

Example 1 – Load statement (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing transactions by timestamp which is loaded into a table called `Transactions`.
- A date field which is provided in the `TimeStamp` system variable `(M/D/YYYY h:mm:ss[.fff] TT)` format.
- A preceding load which contains the `inday()` function which is set as the `in_day` field.

Load script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Transactions:
    Load
        *,
```

```
        inday(date,'01/05/2022 12:00:00 AM', 0) as in_day
        ;
Load
*
Inline
[
id,date,amount
9497,'01/01/2022 7:34:46 PM',13.24
9498,'01/01/2022 10:10:22 PM',31.43
9499,'01/02/2022 8:35:54 AM',36.34
9500,'01/03/2022 2:21:53 PM',51.75
9501,'01/04/2022 6:49:38 PM',15.35
9502,'01/04/2022 10:58:34 PM',74.34
9503,'01/05/2022 5:40:49 AM',73.53
9504,'01/05/2022 11:29:38 AM',50.00
9505,'01/05/2022 7:04:57 PM',47.25
9506,'01/06/2022 8:49:09 AM',74.23
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `in_day`

Results table

date	in_day
01/01/2022 7:34:46 PM	0
01/01/2022 10:10:22 PM	0
01/02/2022 8:35:54 AM	0
01/03/2022 2:21:53 PM	0
01/04/2022 6:49:38 PM	0
01/04/2022 10:58:34 PM	0
01/05/2022 5:40:49 AM	-1
01/05/2022 11:29:38 AM	-1
01/05/2022 7:04:57 PM	-1
01/06/2022 8:49:09 AM	0

The `in_day` field is created in the preceding load statement by using the `inday()` function and passing the date field, a hard-coded timestamp for January 5 and a `period_no` of `0` as the function's arguments.

Example 2 – period_no

Load script and results

Overview

The load script uses the same dataset and scenario that were used in the first example.

However, in this example, the task is to calculate whether the transaction date occurred two days before January 5.

Load script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Transactions:
    Load
        *,
        inday(date,'01/05/2022 12:00:00 AM', -2) as in_day
        ;
Load
*
Inline
[
id,date,amount
9497,'01/01/2022 7:34:46 PM',13.24
9498,'01/01/2022 10:10:22 PM',31.43
9499,'01/02/2022 8:35:54 AM',36.34
9500,'01/03/2022 2:21:53 PM',51.75
9501,'01/04/2022 6:49:38 PM',15.35
9502,'01/04/2022 10:58:34 PM',74.34
9503,'01/05/2022 5:40:49 AM',73.53
9504,'01/05/2022 11:29:38 AM',50.00
9505,'01/05/2022 7:04:57 PM',47.25
9506,'01/06/2022 8:49:09 AM',74.23
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `in_day`

Results table

date	in_day
01/01/2022 7:34:46 PM	0
01/01/2022 10:10:22 PM	0
01/02/2022 8:35:54 AM	0
01/03/2022 2:21:53 PM	-1
01/04/2022 6:49:38 PM	0

date	in_day
01/04/2022 10:58:34 PM	0
01/05/2022 5:40:49 AM	0
01/05/2022 11:29:38 AM	0
01/05/2022 7:04:57 PM	0
01/06/2022 8:49:09 AM	0

In this instance, because a `period_no` of `-2` is used as the offset argument in the `inday()` function, the function determines whether each transaction date took place on January 3. This can be verified in the output table where one transaction returns a Boolean result of TRUE.

Example 3 – day_start

Load script and results

Overview

The load script uses the same dataset and scenario that were used in the previous examples.

However, in this example, the company policy is that the workday begins and ends at 7 AM.

Load script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Transactions:
    Load
        *,
        inday(date,'01/05/2022 12:00:00 AM', 0, 7/24) as in_day
        ;
Load
*
Inline
[
id,date,amount
9497,'01/01/2022 7:34:46 PM',13.24
9498,'01/01/2022 10:10:22 PM',31.43
9499,'01/02/2022 8:35:54 AM',36.34
9500,'01/03/2022 2:21:53 PM',51.75
9501,'01/04/2022 6:49:38 PM',15.35
9502,'01/04/2022 10:58:34 PM',74.34
9503,'01/05/2022 5:40:49 AM',73.53
9504,'01/05/2022 11:29:38 AM',50.00
9505,'01/05/2022 7:04:57 PM',47.25
9506,'01/06/2022 8:49:09 AM',74.23
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `in_day`

Results table

date	in_day
01/01/2022 7:34:46 PM	0
01/01/2022 10:10:22 PM	0
01/02/2022 8:35:54 AM	0
01/03/2022 2:21:53 PM	0
01/04/2022 6:49:38 PM	-1
01/04/2022 10:58:34 PM	-1
01/05/2022 5:40:49 AM	-1
01/05/2022 11:29:38 AM	0
01/05/2022 7:04:57 PM	0
01/06/2022 8:49:09 AM	0

Because the `start_day` argument of 7/24, which is 7 AM, is used in the `inday()` function, the function determines whether each transaction date took place on January 4 from 7 AM and January 5 before 7 AM.

This can be verified in the output table where transactions that take place after 7 AM on January 4 return a Boolean result of TRUE whilst transactions that take place after 7 AM on January 5 return a Boolean result of FALSE.

Example 4 – Chart object

Load script and chart expression

Overview

The load script uses the same dataset and scenario that were used in the previous examples.

However, in this example, the dataset is unchanged and loaded into the application. You will calculate to determine if a transaction takes place on January 5 by creating a measure in a chart object.

Load script

```
Transactions:
Load
*
Inline
```

```
[
id,date,amount
9497,'01/01/2022 7:34:46 PM',13.24
9498,'01/01/2022 10:10:22 PM',31.43
9499,'01/02/2022 8:35:54 AM',36.34
9500,'01/03/2022 2:21:53 PM',51.75
9501,'01/04/2022 6:49:38 PM',15.35
9502,'01/04/2022 10:58:34 PM',74.34
9503,'01/05/2022 5:40:49 AM',73.53
9504,'01/05/2022 11:29:38 AM',50.00
9505,'01/05/2022 7:04:57 PM',47.25
9506,'01/06/2022 8:49:09 AM',74.23
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension:

- `date`

To calculate whether a transaction takes place on January 5, create the following measure:

```
=inday(date,'01/05/2022 12:00:00 AM',0)
```

Results table

date	inday(date,'01/05/2022 12:00:00 AM',0)
01/01/2022 7:34:46 PM	0
01/01/2022 10:10:22 PM	0
01/02/2022 8:35:54 AM	0
01/03/2022 2:21:53 PM	0
01/04/2022 6:49:38 PM	0
01/04/2022 10:58:34 PM	0
01/05/2022 5:40:49 AM	-1
01/05/2022 11:29:38 AM	-1
01/05/2022 7:04:57 PM	-1
01/06/2022 8:49:09 AM	0

Example 5 – Scenario

Load script and results

Overview

In this example, it has been identified that due to equipment error, products that were manufactured on January 5 were defective. The end user would like a chart object that displays, by date, the status of which products that were manufactured were ‘defective’ or ‘faultless’ and the cost of the products manufactured on January 5.

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset which is loaded into a table called ‘Products’.
- The table contains the following fields:
    - product ID
    - manufacture time
    - cost price

Load script

```
Products:
Load
*
Inline
[
product_id,manufacture_date,cost_price
9497,'01/01/2022 7:34:46 PM',13.24
9498,'01/01/2022 10:10:22 PM',31.43
9499,'01/02/2022 8:35:54 AM',36.34
9500,'01/03/2022 2:21:53 PM',51.75
9501,'01/04/2022 6:49:38 PM',15.35
9502,'01/04/2022 10:58:34 PM',74.34
9503,'01/05/2022 5:40:49 AM',73.53
9504,'01/05/2022 11:29:38 AM',50.00
9505,'01/05/2022 7:04:57 PM',47.25
9506,'01/06/2022 8:49:09 AM',74.23
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension:

```
=dayname(manufacture_date)
```

Create the following measures:

- `=if(only(InDay(manufacture_date,makedate(2022,01,05),0)),'Defective','Faultless')`
- `=sum(cost_price)`

Set the measure's **Number Formatting** to **Money**.

Under **Appearance**, turn off **Totals**.

Results table

dayname (manufacture_date)	=if(only(InDay(manufacture_date,makedate (2022,01,05),0)),'Defective','Faultless')	=sum(cost_ price)
01/01/2022	Faultless	44.67
01/02/2022	Faultless	36.34
01/03/2022	Faultless	51.75
01/04/2022	Faultless	89.69
01/05/2022	Defective	170.78
01/06/2022	Faultless	74.23

The `inday()` function returns a Boolean value when evaluating the manufacturing dates of each of the products. For any product manufactured on January 5, the `inday()` function returns a Boolean value of TRUE and marks the products as 'Defective'. For any product returning a value of FALSE, and therefore not manufactured on that day, it marks the products as 'Faultless'.

indaytotime

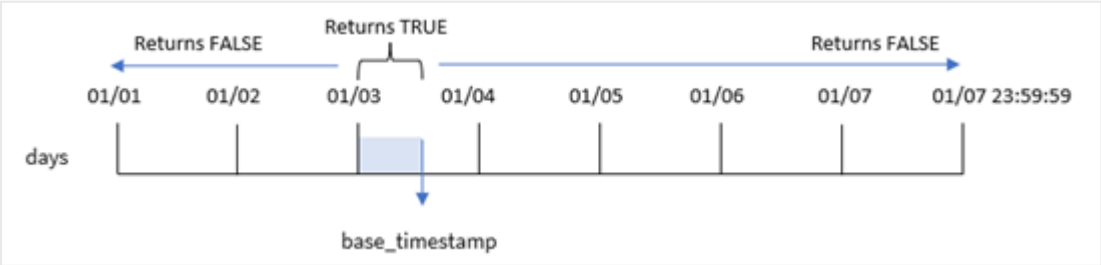
This function returns True if **timestamp** lies inside the part of day containing **base_timestamp** up until and including the exact millisecond of **base_timestamp**.

Syntax:

```
InDayToTime (timestamp, base_timestamp, period_no[, day_start])
```

The `indaytotime()` function returns a Boolean result depending on when a timestamp value occurs during the segment of the day. The start boundary of this segment is the start of the day, which is set as midnight by default; the start of the day can be modified by the `day_start` argument of the `indaytotime()` function. The end boundary of the day segment is determined by a `base_timestamp` argument of the function.

Diagram of `indaytotime` function.



When to use it

The `indaytotime()` function returns a Boolean result. Typically, this type of function will be used as a condition in an `if expression`. The `indaytotime()` function returns an aggregation or calculation depending on if a timestamp occurred in the segment of the day up to and including the time of the base timestamp.

For example, the `indaytotime()` function can be used to show the sum of ticket sales for shows that have taken place so far today.

Return data type: Boolean

In Qlik Sense, the Boolean true value is represented by -1, and the false value is represented by 0.

Arguments

Argument	Description
`timestamp`	The date and time that you want to compare with `base_timestamp`.
`base_timestamp`	Date and time that is used to evaluate the timestamp.
`period_no`	The day can be offset by `period_no`. `period_no` is an integer, where the value 0 indicates the day which contains `base_timestamp`. Negative values in `period_no` indicate preceding days and positive values indicate succeeding days.
`day_start`	(optional) If you want to work with days not starting midnight, indicate an offset as a fraction of a day in `day_start`. For example, use 0.125 to denote 3 AM

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Function examples

Example	Result
`indaytotime` ('01/12/2006 12:23:00 PM', '01/12/2006 11:59:00 PM', 0)	Returns True
`indaytotime` ('01/12/2006 12:23:00 PM', '01/12/2006 12:00:00 AM', 0)	Returns False
`indaytotime` '01/11/2006 12:23:00 PM', '01/12/2006 11:59:00 PM', -1)	Returns True

Example 1 – no additional arguments

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions for the period between January 4 and 5 is loaded into a table called 'Transactions'.
- A date field which is provided in the `TimeStamp` system variable `(M/D/YYYY h:mm:ss[.fff] TT)` format.
- A preceding load which contains the `indaytotime()` function which is set as the `'in_day_to_time'`, field that determines whether each of the transactions take place before 9:00 AM.

Load script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Transactions:
    Load
        *,
        indaytotime(date,'01/05/2022 9:00:00 AM',0) as in_day_to_time
        ;
Load
*
Inline
[
id,date,amount
8188,'01/04/2022 3:41:54 AM',25.66
8189,'01/04/2022 4:19:43 AM',87.21
8190,'01/04/2022 4:53:47 AM',53.80
8191,'01/04/2022 8:38:53 AM',69.98
8192,'01/04/2022 10:37:52 AM',57.42
8193,'01/04/2022 1:54:10 PM',45.89
8194,'01/04/2022 5:53:23 PM',82.77
8195,'01/04/2022 8:13:26 PM',36.23
8196,'01/04/2022 10:00:49 PM',76.11
8197,'01/05/2022 7:45:37 AM',82.06
8198,'01/05/2022 8:44:36 AM',17.17
8199,'01/05/2022 11:26:08 AM',40.39
8200,'01/05/2022 6:43:08 PM',37.23
8201,'01/05/2022 10:54:10 PM',88.27
8202,'01/05/2022 11:09:09 PM',95.93
];
```

Results

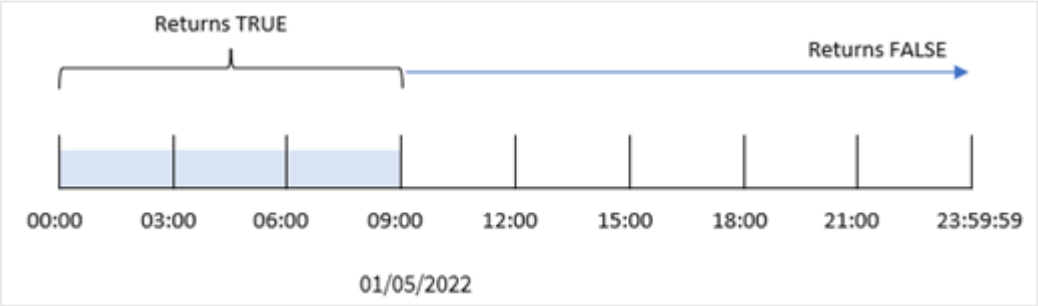
Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `in_day_to_time`

Results table

date	in_day_to_time
01/04/2022 3:41:54 AM	0
01/04/2022 4:19:43 AM	0
01/04/2022 04:53:47 AM	0
01/04/2022 8:38:53 AM	0
01/04/2022 10:37:52 AM	0
01/04/2022 1:54:10 PM	0
01/04/2022 5:53:23 PM	0
01/04/2022 8:13:26 PM	0
01/04/2022 10:00:49 PM	0
01/05/2022 7:45:37 AM	-1
01/05/2022 8:44:36 AM	-1
01/05/2022 11:26:08 AM	0
01/05/2022 6:43:08 PM	0
01/05/2022 10:54:10 PM	0
01/05/2022 11:09:09 PM	0

Example 1 diagram of `indaytotime` function with 9:00 AM limit..



The `in_day_to_time field` is created in the preceding load statement by using the `indaytotime()` function and passing the date field, a hard-coded timestamp for 9:00 AM January 5 and an offset of 0 as the function's arguments. Any transactions that occur between midnight and 9:00 AM on January 5 return TRUE.

Example 2 – period_no

Load script and results

Overview

The load script uses the same dataset and scenario that were used in the first example.

However, in this example, you will calculate whether the transaction date occurred one day before 9:00 AM on January 5.

Load script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Transactions:
    Load
        *,
        indaytotime(date,'01/05/2022 9:00:00 AM', -1) as in_day_to_time
        ;
Load
*
Inline
[
id,date,amount
8188,'01/04/2022 3:41:54 AM',25.66
8189,'01/04/2022 4:19:43 AM',87.21
8190,'01/04/2022 4:53:47 AM',53.80
8191,'01/04/2022 8:38:53 AM',69.98
8192,'01/04/2022 10:37:52 AM',57.42
8193,'01/04/2022 1:54:10 PM',45.89
8194,'01/04/2022 5:53:23 PM',82.77
8195,'01/04/2022 8:13:26 PM',36.23
8196,'01/04/2022 10:00:49 PM',76.11
8197,'01/05/2022 7:45:37 AM',82.06
8198,'01/05/2022 8:44:36 AM',17.17
8199,'01/05/2022 11:26:08 AM',40.39
8200,'01/05/2022 6:43:08 PM',37.23
8201,'01/05/2022 10:54:10 PM',88.27
8202,'01/05/2022 11:09:09 PM',95.93
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

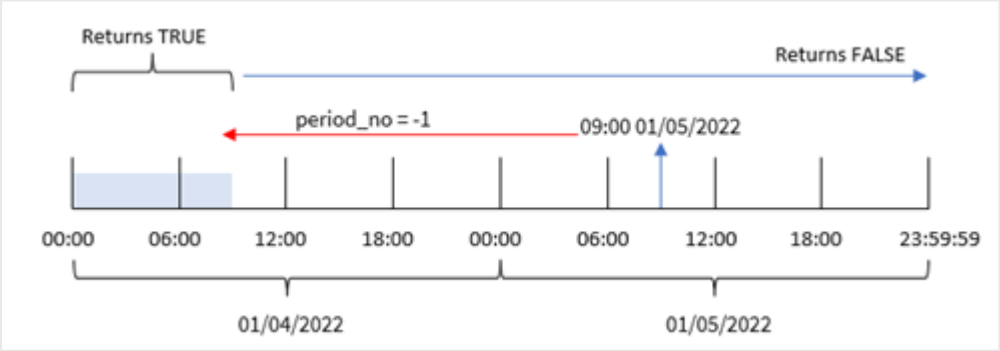
- `date`
- `in_day_to_time`

Results table

date	in_day_to_time
01/04/2022 3:41:54 AM	-1
01/04/2022 4:19:43 AM	-1

date	in_day_to_time
01/04/2022 04:53:47 AM	-1
01/04/2022 8:38:53 AM	-1
01/04/2022 10:37:52 AM	0
01/04/2022 1:54:10 PM	0
01/04/2022 5:53:23 PM	0
01/04/2022 8:13:26 PM	0
01/04/2022 10:00:49 PM	0
01/05/2022 7:45:37 AM	0
01/05/2022 8:44:36 AM	0
01/05/2022 11:26:08 AM	0
01/05/2022 6:43:08 PM	0
01/05/2022 10:54:10 PM	0
01/05/2022 11:09:09 PM	0

Example 2 diagram of `indaytotime` function with transactions from January 4.



In this example, because an offset of -1 was used as the offset argument in the `indaytotime()` function, the function determines whether each transaction date took place before 9:00 AM on January 4. This can be verified in the output table where a transaction returns a Boolean result of TRUE.

Example 3 – day_start

Load script and results

Overview

The same dataset and scenario as the first example are used.

However, in this example, the company policy is that the workday begins and ends at 8AM.

Load script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Transactions:
    Load
        *,
        indaytotime(date,'01/05/2022 9:00:00 AM', 0,8/24) as in_day_to_time
        ;
Load
*
Inline
[
id,date,amount
8188,'01/04/2022 3:41:54 AM',25.66
8189,'01/04/2022 4:19:43 AM',87.21
8190,'01/04/2022 4:53:47 AM',53.80
8191,'01/04/2022 8:38:53 AM',69.98
8192,'01/04/2022 10:37:52 AM',57.42
8193,'01/04/2022 1:54:10 PM',45.89
8194,'01/04/2022 5:53:23 PM',82.77
8195,'01/04/2022 8:13:26 PM',36.23
8196,'01/04/2022 10:00:49 PM',76.11
8197,'01/05/2022 7:45:37 AM',82.06
8198,'01/05/2022 8:44:36 AM',17.17
8199,'01/05/2022 11:26:08 AM',40.39
8200,'01/05/2022 6:43:08 PM',37.23
8201,'01/05/2022 10:54:10 PM',88.27
8202,'01/05/2022 11:09:09 PM',95.93
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

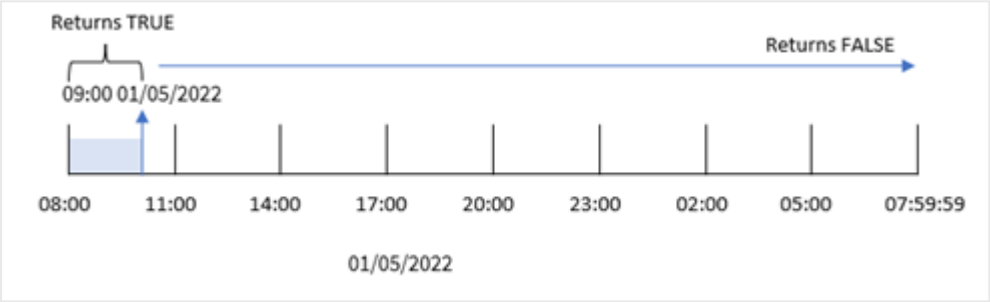
- `date`
- `in_day_to_time`

Results table

date	in_day_to_time
01/04/2022 3:41:54 AM	0
01/04/2022 4:19:43 AM	0
01/04/2022 04:53:47 AM	0
01/04/2022 8:38:53 AM	0
01/04/2022 10:37:52 AM	0
01/04/2022 1:54:10 PM	0
01/04/2022 5:53:23 PM	0

date	in_day_to_time
01/04/2022 8:13:26 PM	0
01/04/2022 10:00:49 PM	0
01/05/2022 7:45:37 AM	0
01/05/2022 8:44:36 AM	-1
01/05/2022 11:26:08 AM	0
01/05/2022 6:43:08 PM	0
01/05/2022 10:54:10 PM	0
01/05/2022 11:09:09 PM	0

Example 3 diagram of `indaytotime` function with transactions from 8:00 AM to 9:00 AM.,



Because the `start_day` argument of 8/24, which equates to 8:00 AM, is used in the `indaytotime()` function, each day begins and ends at 8:00 AM. Therefore, the `indaytotime()` function will return a Boolean result of TRUE for any transaction that took place between 8:00 AM and 9:00 AM on January 5.

Example 4 – Chart object

Load script and chart expression

Overview

The same dataset and scenario as the first example are used.

However, in this example, the dataset is unchanged and loaded into the application. You will calculate to determine if a transaction takes place on January 5 before 9:00 AM by creating a measure in a chart object.

Load script

```
Transactions:
Load
*
Inline
[
id,date,amount
8188,'01/04/2022 3:41:54 AM',25.66
8189,'01/04/2022 4:19:43 AM',87.21
```

```
8190,'01/04/2022 4:53:47 AM',53.80
8191,'01/04/2022 8:38:53 AM',69.98
8192,'01/04/2022 10:37:52 AM',57.42
8193,'01/04/2022 1:54:10 PM',45.89
8194,'01/04/2022 5:53:23 PM',82.77
8195,'01/04/2022 8:13:26 PM',36.23
8196,'01/04/2022 10:00:49 PM',76.11
8197,'01/05/2022 7:45:37 AM',82.06
8198,'01/05/2022 8:44:36 AM',17.17
8199,'01/05/2022 11:26:08 AM',40.39
8200,'01/05/2022 6:43:08 PM',37.23
8201,'01/05/2022 10:54:10 PM',88.27
8202,'01/05/2022 11:09:09 PM',95.93
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension:

`date`.

To determine if a transaction takes place on January 5 before 9:00 AM, create the following measure:

```
=indaytotime(date,'01/05/2022 9:00:00 AM',0)
```

Results table

date	=indaytotime(date,'01/05/2022 9:00:00 AM',0)
01/04/2022 3:41:54 AM	0
01/04/2022 4:19:43 AM	0
01/04/2022 04:53:47 AM	0
01/04/2022 8:38:53 AM	0
01/04/2022 10:37:52 AM	0
01/04/2022 1:54:10 PM	0
01/04/2022 5:53:23 PM	0
01/04/2022 8:13:26 PM	0
01/04/2022 10:00:49 PM	0
01/05/2022 7:45:37 AM	-1
01/05/2022 8:44:36 AM	-1
01/05/2022 11:26:08 AM	0
01/05/2022 6:43:08 PM	0
01/05/2022 10:54:10 PM	0
01/05/2022 11:09:09 PM	0

The `in_day_to_time` measure is created in the chart object by using the `indaytotime()` function and passing the date field, a hard-coded timestamp for 9:00 AM on January 5 and an offset of `0` as the function's arguments. Any transactions that occur between midnight and 9:00 AM on January 5 return TRUE. This is validated in the results table.

Example 5 – Scenario

Load script and results

Overview

In this example, a dataset containing ticket sales for a local cinema is loaded into a table called Ticket_Sales. Today is May 3, 2022 and it is 11:00 AM.

The user would like a KPI chart object to show the revenue earned from all shows that have taken place so far today.

Load script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';

Ticket_Sales:
Load
*
Inline
[
sale ID, show time, ticket price
1,05/01/2022 09:30:00 AM,10.50
2,05/03/2022 05:30:00 PM,21.00
3,05/03/2022 09:30:00 AM,10.50
4,05/03/2022 09:30:00 AM,31.50
5,05/03/2022 09:30:00 AM,10.50
6,05/03/2022 12:00:00 PM,42.00
7,05/03/2022 12:00:00 PM,10.50
8,05/03/2022 05:30:00 PM,42.00
9,05/03/2022 08:00:00 PM,31.50
10,05/04/2022 10:30:00 AM,31.50
11,05/04/2022 12:00:00 PM,10.50
12,05/04/2022 05:30:00 PM,10.50
13,05/05/2022 05:30:00 PM,21.00
14,05/06/2022 12:00:00 PM,21.00
15,05/07/2022 09:30:00 AM,42.00
16,05/07/2022 10:30:00 AM,42.00
17,05/07/2022 10:30:00 AM,10.50
18,05/07/2022 05:30:00 PM,10.50
19,05/08/2022 05:30:00 PM,21.00
20,05/11/2022 09:30:00 AM,10.50
];
```

Results

Do the following:

1. Create a KPI object.
2. Create a measure that will show the sum of all ticket sales for shows that have taken place today so far using the `indaytotime()` function:

```
=sum(if(indaytotime([show time],'05/03/2022 11:00:00 AM',0),[ticket price],0))
```

3. Create a label for the KPI object, 'Current Revenue'.
4. Set the measure's **Number Formatting** to **Money**.

The sum total of ticket sales up to 11:00 AM on May 3, 2022 is $52.50.

The `indaytotime ()` function returns a Boolean value when comparing the show times of each of the ticket sales to the current time ('05/03/2022 11:00:00 AM'). For any show on May 3 before 11:00 AM, the `indaytotime ()` function returns a Boolean value of TRUE and its ticket price will be included in the sum total.

inlunarweek

This function finds if **timestamp** lies inside the lunar week containing **base_date**. Lunar weeks in Qlik Sense are defined by counting 1 January as the first day of the week.

Syntax:

```
InLunarWeek (timestamp, base_date, period_no[, first_week_day])
```

Return data type: Boolean

Arguments:

Arguments

Argument	Description
timestamp	The date that you want to compare with **base_date**.
base_date	Date that is used to evaluate the lunar week.
period_no	The lunar week can be offset by **period_no**. period_no is an integer, where the value 0 indicates the lunar week which contains **base_date**. Negative values in **period_no** indicate preceding lunar weeks and positive values indicate succeeding lunar weeks.
first_week_day	An offset that may be greater than or less than zero. This changes the beginning of the year by the specified number of days and/or fractions of a day.

Examples and results:

Scripting examples

Example	Result
`inlunarweek ('12/01/2013', '14/01/2013', 0)`	Returns True. Because the value of timestamp, 12/01/2013 falls in the week 08/01/2013 to 14/01/2013.

Example	Result
`inlunarweek ('12/01/2013', '07/01/2013', 0)`	Returns False. Because the base_date 07/01/2013 is in the lunar week defined as 01/01/2013 to 07/01/2013.
`inlunarweek ('12/01/2013', '14/01/2013', -1)`	Returns False. Because specifying a value of period_no as -1 shifts the week to the previous week, 01/01/2013 to 07/01/2013.
`inlunarweek ('07/01/2013', '14/01/2013', -1)`	Returns True. In comparison with the previous example, the timestamp is in the week after taking into account the shift backwards.
`inlunarweek ('11/01/2006', '08/01/2006', 0, 3)`	Returns False. Because specifying a value for first_week_day as 3 means the start of the year is calculated from 04/01/2013, and so the value of base_date falls in the first week, and the value of timestamp falls in the week 11/01/2013 to 17/01/2013.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example checks if an invoice date falls in the week shifted from the value of base_date by four weeks.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
InLunarWeek(InvDate, '11/01/2013', 4) AS InLWeekPlus4
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the inlunarweek() function.

The function returns True for the value of InvDate5/2/2013 because the value of base_date, 11/01/2013, is shifted by four weeks, and so falls in the week 5/02/2013 to 11/02/2013.

Results table

InvDate	InLWeekPlus4
28/03/2012	0 (False)
10/12/2012	0 (False)
5/2/2013	-1 (True)
31/3/2013	0 (False)
19/5/2013	0 (False)
15/9/2013	0 (False)
11/12/2013	0 (False)
2/3/2014	0 (False)
14/5/2014	0 (False)
13/6/2014	0 (False)
7/7/2014	0 (False)
4/8/2014	0 (False)

inlunarweektodate

This function finds if **timestamp** lies inside the part of the lunar week up to and including the last millisecond of **base_date**. Lunar weeks in Qlik Sense are defined by counting 1 January as the first day of the week.

Syntax:

```
InLunarWeekToDate (timestamp, base_date, period_no [, first_week_day])
```

Arguments:

Arguments

Argument	Description
timestamp	The date that you want to compare with **base_date**.
base_date	Date that is used to evaluate the lunar week.
period_no	The lunar week can be offset by **period_no**. period_no is an integer, where the value 0 indicates the lunar week which contains **base_date**. Negative values in **period_no** indicate preceding lunar weeks and positive values indicate succeeding lunar weeks.
first_week_day	An offset that may be greater than or less than zero. This changes the beginning of the year by the specified number of days and/or fractions of a day.

Examples and results:

Scripting examples

Example	Result
`inlunarweektodate ('12/01/2013', '13/01/2013', 0)`	Returns True. Because the value of timestamp, 12/01/2013 falls in the part of the week 08/01/2013 to 13/01/2013.
`inlunarweektodate ('12/01/2013', '11/01/2013', 0)`	Returns False. Because the value of timestamp is later than the value base_date even though the two dates are in the same lunar week before 12/01/2012.
`inlunarweektodate ('12/01/2006', '05/01/2006', 1)`	Returns True. Specifying a value of 1 for period_no shifts the base_date forward one week, so the value of timestamp falls in the part of the lunar week.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example checks if an invoice date falls in the part of the week shifted from the value of base_date by four weeks.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
InLunarWeekToDate(InvDate, '07/01/2013', 4) AS InLWeek2DPlus4
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the inlunarweek() function.

The function returns True for the value of InvDate5/2/2013 because the value of base_date, 11/01/2013, is shifted by four weeks, and so falls in the part of the week 5/02/2013 to 07/02/2013.

Results table

InvDate	InLWeek2DPlus4
28/03/2012	0 (False)
10/12/2012	0 (False)
5/2/2013	-1 (True)
31/3/2013	0 (False)
19/5/2013	0 (False)
15/9/2013	0 (False)
11/12/2013	0 (False)
2/3/2014	0 (False)
14/5/2014	0 (False)
13/6/2014	0 (False)
7/7/2014	0 (False)
4/8/2014	0 (False)

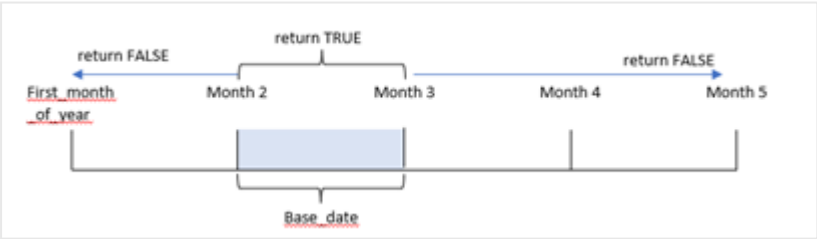
inmonth

This function returns True if **timestamp** lies inside the month containing **base_date**.

Syntax:

```
InMonth (timestamp, base_date, period_no)
```

Diagram of `indaytotime` function.



In other words, the `inmonth()` function determines if a set of dates fall into this month and returns a Boolean value based on a `base_date` that identifies the month.

When to use it

The `inmonth()` function returns a Boolean result. Typically, this type of function will be used as a condition in an `if expression`. This returns an aggregation or calculation depending on whether a date occurred in the month, including the date in question.

For example, the `inmonth()` function can be used to identify all equipment manufactured in a specific month.

Return data type: Boolean

In Qlik Sense, the Boolean true value is represented by -1, and the false value is represented by 0.

Arguments

Argument	Description
timestamp	The date that you want to compare with `base_date`.
base_date	Date that is used to evaluate the month. It is important to note that the `base_date` can be any day within a month.
period_no	The month can be offset by `period_no`. `period_no` is an integer, where the value 0 indicates the month which contains `base_date`. Negative values in `period_no` indicate preceding months and positive values indicate succeeding months.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Function examples

Example	Result
`inmonth ('25/01/2013', '01/01/2013', 0)`	Returns True
`inmonth('25/01/2013', '23/04/2013', 0)`	Returns False
`inmonth ('25/01/2013', '01/01/2013', -1)`	Returns False
`inmonth ('25/12/2012', '17/01/2013', -1)`	Returns True

Example 1 – No additional arguments

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions for the first half of 2022.
- A preceding load with an additional variable, ‘`in_month`’, that determines whether transactions took place in April.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
  Load
    *,
    inmonth(date,'04/01/2022', 0) as in_month
    ;
Load
*
Inline
[
id,date,amount
8188,'1/10/2022',37.23
8189,'1/14/2022',17.17
8190,'1/20/2022',88.27
8191,'1/22/2022',57.42
8192,'2/1/2022',53.80
8193,'2/2/2022',82.06
8194,'2/20/2022',40.39
8195,'4/11/2022',87.21
8196,'4/13/2022',95.93
8197,'4/15/2022',45.89
8198,'4/25/2022',36.23
8199,'5/20/2022',25.66
8200,'5/22/2022',82.77
8201,'6/19/2022',69.98
8202,'6/22/2022',76.11
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `in_month`

Function examples

date	in_month
1/10/2022	0
1/14/2022	0
1/20/2022	0
1/22/2022	0

date	in_month
2/1/2022	0
2/2/2022	0
2/20/2022	0
4/11/2022	-1
4/13/2022	-1
4/15/2022	-1
4/25/2022	-1
5/20/2022	0
5/22/2022	0
6/19/2022	0
6/22/2022	0

The '`in_month`' field is created in the preceding load statement by using the `inmonth()` function and passing the date field, a hard-coded date of April `1`, as our `base_date` and a `period_no` of `0` as the function's arguments.

The `base_date` identifies the month that will return a Boolean result of TRUE. Therefore, all transactions that occurred in April return TRUE which is validated in the results table.

Example 2 – period_no

Load script and results

Overview

The same dataset and scenario from the first example are used.

However, in this example, you will create a field, '`2_months_prior`', that determines whether the transactions took place two months before April.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
Load
   *,
   inmonth(date,'04/01/2022', -2) as [2_months_prior]
Inline
[
id,date,amount
8188,'1/10/2022',37.23
8189,'1/14/2022',17.17
8190,'1/20/2022',88.27
8191,'1/22/2022',57.42
```

```
8192,'2/1/2022',53.80
8193,'2/2/2022',82.06
8194,'2/20/2022',40.39
8195,'4/11/2022',87.21
8196,'4/13/2022',95.93
8197,'4/15/2022',45.89
8198,'4/25/2022',36.23
8199,'5/20/2022',25.66
8200,'5/22/2022',82.77
8201,'6/19/2022',69.98
8202,'6/22/2022',76.11
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `2_months_prior`

Function examples

date	2_months_prior
1/10/2022	0
1/14/2022	0
1/20/2022	0
1/22/2022	0
2/1/2022	-1
2/2/2022	-1
2/20/2022	-1
4/11/2022	0
4/13/2022	0
4/15/2022	0
4/25/2022	0
5/20/2022	0
5/22/2022	0
6/19/2022	0
6/22/2022	0

Using -2 as the `period_no` argument in the `inmonth()` function shifts the month defined by the `base_date` argument two months prior. In this example it changes the defined month from April to February.

Therefore, any transaction that takes place in February will return a Boolean result of TRUE.

Example 3 – Chart object

Load script and chart expression

Overview

The same dataset and scenario from the previous examples are used.

However, in this example, the dataset is unchanged and loaded into the application. The calculation that determines whether transactions took place in April is created as a measure in a chart object of the application.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
Load
*
Inline
[
id,date,amount
8188,'1/10/2022',37.23
8189,'1/14/2022',17.17
8190,'1/20/2022',88.27
8191,'1/22/2022',57.42
8192,'2/1/2022',53.80
8193,'2/2/2022',82.06
8194,'2/20/2022',40.39
8195,'4/11/2022',87.21
8196,'4/13/2022',95.93
8197,'4/15/2022',45.89
8198,'4/25/2022',36.23
8199,'5/20/2022',25.66
8200,'5/22/2022',82.77
8201,'6/19/2022',69.98
8202,'6/22/2022',76.11
];
```

Chart object

Load the data and open a sheet. Create a new table and add this field as a dimension:

```
date
```

To calculate whether a transaction takes place in April, create the following measure:

```
=inmonth(date,'04/01/2022', 0)
```

Results

	Function examples
date	**=inmonth(date,'04/01/2022', 0)**
1/10/2022	0
1/14/2022	0
1/20/2022	0
1/22/2022	0
2/1/2022	0
2/2/2022	0
2/20/2022	0
4/11/2022	-1
4/13/2022	-1
4/15/2022	-1
4/25/2022	-1
5/20/2022	0
5/22/2022	0
6/19/2022	0
6/22/2022	0

Example 4 – Scenario

Load script and results

Overview

In this example, a dataset is loaded into a table called ‘`Products`’. The table contains the following fields:

- Product ID
- Manufacture date
- Cost price

Due to equipment error, products that were manufactured in the month of July 2022 were defective. The issue was resolved on July 27, 2022.

The end user would like a chart that displays, by month, the status of products that were manufactured as ‘defective’ (Boolean TRUE) or ‘faultless’ (Boolean FALSE) and the cost of the products manufactured in that month.

Load script

```
Products:
Load
*
Inline
[
product_id,manufacture_date,cost_price
8188,'1/19/2022',37.23
8189,'1/7/2022',17.17
8190,'2/28/2022',88.27
8191,'2/5/2022',57.42
8192,'3/16/2022',53.80
8193,'4/1/2022',82.06
8194,'5/7/2022',40.39
8195,'5/16/2022',87.21
8196,'6/15/2022',95.93
8197,'6/26/2022',45.89
8198,'7/9/2022',36.23
8199,'7/22/2022',25.66
8200,'7/23/2022',82.77
8201,'7/27/2022',69.98
8202,'8/2/2022',76.11
8203,'8/8/2022',25.12
8204,'8/19/2022',46.23
8205,'9/26/2022',84.21
8206,'10/14/2022',96.24
8207,'10/29/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension:

`=monthname(manufacture_date)`

Create the following measures:

- `=sum(cost_price)`
- `=if(only(inmonth(manufacture_date,makedate(2022,07,01),0)),'Defective','Faultless')`

1. Set the measure's **Number Formatting** to **Money**.
2. Under **Appearance**, turn off **Totals**.

Results table

monthname (manufacture_date)	=if(only(inmonth(manufacture_date,makedate (2022,07,01),0)),'Defective','Faultless')	sum(cost_ price)
Jan 2022	Faultless	$54.40
Feb 2022	Faultless	$145.69
Mar 2022	Faultless	$53.80

monthname (manufacture_date)	=if(only(inmonth(manufacture_date,makedate (2022,07,01),0)),'Defective','Faultless')	sum(cost_ price)
Apr 2022	Faultless	$82.06
May 2022	Faultless	$127.60
Jun 2022	Faultless	$141.82
Jul 2022	Defective	$214.64
Aug 2022	Faultless	$147.46
Sep 2022	Faultless	$84.21
Oct 2022	Faultless	$163.91

The `inmonth()` function returns a Boolean value when evaluating the manufacturing dates of each of the products. For any product manufactured in July 2022, the `inmonth()` function returns a Boolean value of True and marks the products as 'Defective'. For any product returning a value of False, and therefore not manufactured in July, it marks the products as 'Faultless'.

inmonths

This function finds if a timestamp falls within the same month, bi-month, quarter, four-month period, or half-year as a base date.It is also possible to find if the timestamp falls within a previous or following time period.

Syntax:

```
InMonths(n_months, timestamp, base_date, period_no [, first_month_of_year])
```

Arguments:

Arguments

Argument	Description
n_months	The number of months that defines the period. An integer or expression that resolves to an integer that must be one of: 1 (equivalent to the inmonth() function), 2 (bi-month), 3 (equivalent to the inquarter() function), 4 (four-month period), or 6 (half year).
timestamp	The date that you want to compare with **base_date**.
base_date	Date that is used to evaluate the period.
period_no	The period can be offset by **period_no**, an integer, or expression resolving to an integer, where the value 0 indicates the period that contains **base_date**. Negative values in **period_no** indicate preceding periods and positive values indicate succeeding periods.
first_month_ of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

Month	Value
February	2
March	3
April	4
May	5
June	6
July	7
August	8
September	9
October	10
November	11
December	12

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`inmonths(4, '25/01/2013', '25/04/2013', 0)`	Returns True. Because the value of timestamp, 25/01/2013, lies within the four-month period 01/01/2013 to 30/04/2013, in which the value of base_date, 25/04/2013 lies.
`inmonths(4, '25/05/2013', '25/04/2013', 0)`	Returns False. Because 25/05/2013 is outside the same period as the previous example.
`inmonths(4, '25/11/2012', '01/02/2013', -1 )`	Returns True. Because the value of period_no, -1, shifts the search period back one period of four months (the value of n-months), which makes the search period 01/09/2012 to 31/12/2012.
`inmonths( 4, '25/05/2006', '01/03/2006', 0, 3)`	Returns True. Because the value of first_month_of_year is set to 3, which makes the search period 01/03/2006 to 30/07/2006 instead of 01/01/2006 to 30/04/2006.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example checks if the invoice date in the table falls in the bi-month period that includes the base_date shifted forwards by one bi-month period (by specifying period_no as 1).

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
InMonths(2, InvDate, '11/02/2013', 1) AS InMthsPlus1
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the InMonths() function.

The search period is 01/03/2013 to 30/04/2013, because the value of base_date is shifted forwards two months from the value in the function (11/02/2013).

Results table

InvDate	InMthsPlus1
28/03/2012	0 (False)
10/12/2012	0 (False)
5/2/2013	0 (False)
31/3/2013	-1 (True)
19/5/2013	0 (False)
15/9/2013	0 (False)
11/12/2013	0 (False)
2/3/2014	0 (False)
14/5/2014	0 (False)

InvDate	InMthsPlus1
13/6/2014	0 (False)
7/7/2014	0 (False)
4/8/2014	0 (False)

inmonthstodate

This function finds if a timestamp falls within the part a period of the month, bi-month, quarter, four-month period, or half-year up to and including the last millisecond of `base_date`. It is also possible to find if the timestamp falls within a previous or following time period.

Syntax:

```
InMonths (n_months, timestamp, base_date, period_no[, first_month_of_year ])
```

Return data type: Boolean

Arguments:

Arguments

Argument	Description
n_months	The number of months that defines the period. An integer or expression that resolves to an integer that must be one of: 1 (equivalent to the inmonth() function), 2 (bi-month), 3 (equivalent to the inquarter() function), 4 (four-month period), or 6 (half year).
timestamp	The date that you want to compare with **base_date**.
base_date	Date that is used to evaluate the period.
period_no	The period can be offset by **period_no**, an integer, or expression resolving to an integer, where the value 0 indicates the period that contains **base_date**. Negative values in **period_no** indicate preceding periods and positive values indicate succeeding periods.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`inmonthstodate(4, '25/01/2013', '25/04/2013', 0)`	Returns True. Because the value of timestamp, 25/01/2013, lies within the four-month period 01/01/2013 up to the end of 25/04/2013, in which the value of base_date, 25/04/2013 lies.

Example	Result
`inmonthstodate(4, '26/04/2013', '25/04/2006', 0)`	Returns False. Because 26/04/2013 is outside the same period as the previous example.
`inmonthstodate(4, '25/09/2005', '01/02/2006', -1)`	Returns True. Because the value of period_no, -1, shifts the search period back one period of four months (the value of n-months), which makes the search period 01/09/2005 to 01/02/2006.
`inmonthstodate(4, '25/04/2006', '01/06/2006', 0, 3)`	Returns True. Because the value of first_month_of_year is set to 3, which makes the search period 01/03/2006 to 01/06/2006 instead of 01/05/2006 to 01/06/2006.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example checks if the invoice date in the table falls in the part of the bi-month period up to and including the base_date shifted forwards by four bi-month periods (by specifying period_no as 4).

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
InMonthsToDate(2, InvDate, '15/02/2013', 4) AS InMths2DPlus4
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the InMonths() function.

The search period is 01/09/2013 to 15/10/2013, because the value of base_date is shifted forwards eight months from the value in the function (15/02/2013).

Results table

InvDate	InMths2DPlus4
28/03/2012	0 (False)
10/12/2012	0 (False)
5/2/2013	0 (False)
31/3/2013	0 (False)
19/5/2013	0 (False)
15/9/2013	-1 (True)
11/12/2013	0 (False)
2/3/2014	0 (False)
14/5/2014	0 (False)
13/6/2014	0 (False)
7/7/2014	0 (False)
4/8/2014	0 (False)

inmonthtodate

Returns True if **date** lies inside the part of month containing **basedate** up until and including the last millisecond of **basedate**.

Syntax:

```
InMonthToDate (timestamp, base_date, period_no)
```

Return data type: Boolean

Arguments:

Overview

The same dataset and scenario as the first example are used.

However, in this example, the dataset, unchanged, is loaded into the app. The calculation that determines whether transactions took place between July 1 and July 26 is created as a measure in a chart of the app.

Load script

Scripting examples

Example	Result
`inmonthtodate ('25/01/2013', '25/01/2013', 0)`	Returns True
`inmonthtodate ('25/01/2013', '24/01/2013', 0)`	Returns False
`inmonthtodate ('25/01/2013', '28/02/2013', -1)`	Returns True

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

By specifying period_no as 4, this example checks if an invoice date falls in the fourth month after the month in base_date but before the end of the day specified in base_date.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
InMonthToDate(InvDate, '31/01/2013', 4) AS InMthPlus42D
Resident TempTable;
```

```
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the inmonthtodate() function.

Results table

InvDate	InMthPlus42D
28/03/2012	0 (False)
10/12/2012	0 (False)
5/2/2013	0 (False)
31/3/2013	0 (False)
19/5/2013	-1 (True)
15/9/2013	0 (False)
11/12/2013	0 (False)
2/3/2014	0 (False)
14/5/2014	0 (False)
13/6/2014	0 (False)
7/7/2014	0 (False)
4/8/2014	0 (False)

inquarter

This function returns True if **timestamp** lies inside the quarter containing **base_date**.

Syntax:

```
InQuarter (timestamp, base_date, period_no[, first_month_of_year])
```

Return data type: Boolean

Arguments:

Month	Value
February	2
March	3
April	4
May	5
June	6
July	7

Month	Value
August	8
September	9
October	10
November	11
December	12

Example	Result
inquarter ('01/25/2013', '01/01/2013', 0)	Returns TRUE
inquarter ('01/25/2013', '04/01/2013', 0)	Returns FALSE
inquarter ('01/25/2013', '01/01/2013', -1)	Returns FALSE
inquarter ('12/25/2012', '01/01/2013', -1)	Returns TRUE
inquarter ('01/25/2013', '03/01/2013', 0, 3)	Returns FALSE
inquarter ('03/25/2013', '03/01/2013', 0, 3)	Returns TRUE

Examples and results:

Scripting examples

Example	Result
`inquarter ('25/01/2013', '01/01/2013', 0)`	Returns True
`inquarter ('25/01/2013', '01/04/2013', 0)`	Returns False
`inquarter ('25/01/2013', '01/01/2013', -1)`	Returns False
`inquarter ('25/12/2012', '01/01/2013', -1)`	Returns True
`inquarter ('25/01/2013', '01/03/2013', 0, 3)`	Returns False
`inquarter ('25/03/2013', '01/03/2013', 0, 3)`	Returns True

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example checks if an invoice date falls in the fourth quarter of the fiscal year specified by setting the value of first_month_of_year to 4, and having the base_date 31/01/2013.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
```

```
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
InQuarter(InvDate, '31/01/2013', 0, 4) AS Qtr4FinYr1213
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the inquarter() function.

Results table

InvDate	Qtr4Fin1213
28/03/2012	0 (False)
10/12/2012	0 (False)
5/2/2013	-1 (True)
31/3/2013	-1 (True)
19/5/2013	0 (False)
15/9/2013	0 (False)
11/12/2013	0 (False)
2/3/2014	0 (False)
14/5/2014	0 (False)
13/6/2014	0 (False)
7/7/2014	0 (False)
4/8/2014	0 (False)

inquartertodate

This function returns True if **timestamp** lies inside the part of the quarter containing **base_date** up until and including the last millisecond of **base_date**.

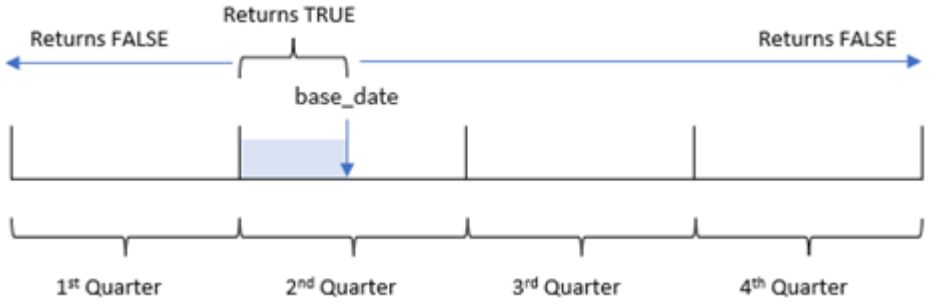
Syntax:

```
InQuarterToDate (timestamp, base_date, period_no [, first_month_of_year])
```

Return data type: Boolean

In Qlik Sense, the Boolean true value is represented by -1, and the false value is represented by 0.

Diagram of inquartertodate function



The `inquartertodate()` function divides the year into four equal quarters between January 1 and December 31 (or the user-defined start of year and its corresponding end date). Using the `base_date`, the function will then segment a particular quarter, with the `base_date` identifying both which quarter and the maximum allowed date for that quarter segment. Finally, the function returns a Boolean result when comparing the prescribed date values to that segment.

Arguments

Argument	Description
timestamp	The date that you want to compare with **base_date**.
base_date	Date that is used to evaluate the quarter.
period_no	The quarter can be offset by **period_no**. **period_no** is an integer, where the value 0 indicates the quarter which contains **base_date**. Negative values in **period_no** indicate preceding quarters and positive values indicate succeeding quarters.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

When to use it

The `inquartertodate()` function returns a Boolean result. Typically, this type of function will be used as a condition in an `if` expression. The `inquartertodate()` function would be used to return an aggregation or calculation dependent on whether a date evaluated occurred in the quarter up to and including the date in question.

For example, the `inquartertodate()` function can be used to identify all equipment manufactured in a quarter up to a specific date.

Function examples

Example	Result
`inquartertodate ('01/25/2013', '03/25/2013', 0)`	Returns `TRUE`, since the value of `timestamp`, 01/25/2013, lies within the three-month period from 01/01/2013 to 03/25/2013, in which the value of `base_date`, 03/25/2013, lies.
`inquartertodate ('04/26/2013', '03/25/2013', 0)`	Returns `FALSE`, since 04/26/2013 is outside the same period as the previous example.
`inquartertodate ('02/25/2013', '06/09/2013', -1)`	Returns `TRUE`, since the value of `period_no`, -1, shifts the search period back one period of three months (one quarter of the year). This makes the search period 01/01/2013 to 03/09/2013.
`inquartertodate ('03/25/2006', '04/15/2006', 0, 2)`	Returns `TRUE`, since the value of `first_month_of_year` is set to 2, which makes the search period 02/01/2006 to 04/15/2006 instead of 04/01/2006 to 04/15/2006.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 – No additional arguments

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions for 2022, which is loaded into a table called `Transactions`. The date field has been provided in the `DateFormat` system variable (MM/DD/YYYY) format.
- The date field provided in the `DateFormat` system variable (MM/DD/YYYY) format.
- The creation of a field, `in_quarter_to_date`, that determines which transactions took place in the quarter up until May 15, 2022.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
        *,
        inquartertodate(date,'05/15/2022', 0) as in_quarter_to_date
        ;
Load
*
Inline
[
id,date,amount
8188,'1/19/2022',37.23
8189,'1/7/2022',17.17
8190,'2/28/2022',88.27
8191,'2/5/2022',57.42
8192,'3/16/2022',53.80
8193,'4/1/2022',82.06
8194,'5/7/2022',40.39
8195,'5/16/2022',87.21
8196,'6/15/2022',95.93
8197,'6/26/2022',45.89
8198,'7/9/2022',36.23
8199,'7/22/2022',25.66
8200,'7/23/2022',82.77
8201,'7/27/2022',69.98
8202,'8/2/2022',76.11
8203,'8/8/2022',25.12
8204,'8/19/2022',46.23
8205,'9/26/2022',84.21
8206,'10/14/2022',96.24
8207,'10/29/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `in_quarter_to_date`

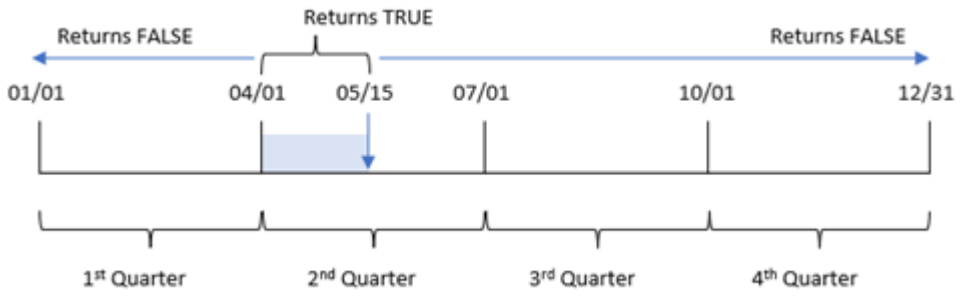
Results table

date	in_quarter_to_date
1/7/2022	0
1/19/2022	0
2/5/2022	0
2/28/2022	0

date	in_quarter_to_date
3/16/2022	0
4/1/2022	-1
5/7/2022	-1
5/16/2022	0
6/15/2022	0
6/26/2022	0
7/9/2022	0
7/22/2022	0
7/23/2022	0
7/27/2022	0
8/2/2022	0
8/8/2022	0
8/19/2022	0
9/26/2022	0
10/14/2022	0
10/29/2022	0

The `in_quarter_to_date` field is created in the preceding load statement by using the `inquartertodate()` function. The first argument provided identifies which field is being evaluated. The second argument is a hard-coded date for the for May 15, which is the `base_date` that identifies which quarter to segment and defines the end boundary of that segment. A `period_no` of 0 is the final argument, meaning that the function is not comparing quarters preceding or following the segmented quarter.

Diagram of inquartertodate function, no additional arguments



Any transaction that occurs in between April 1 and May 15 returns a Boolean result of `TRUE`. Transactions dates of May 16 and later will return `FALSE`, as do any transactions before April 1.

Example 2 – period_no

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset and scenario as the first example.
- The creation of a field, `previous_qtr_to_date`, that determines which transactions took place a full quarter before the quarter segment ending on May 15, 2022.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
        *,
        inquartertodate(date,'05/15/2022', -1) as previous_qtr_to_date
        ;
Load
*
Inline
[
id,date,amount
8188,'1/19/2022',37.23
8189,'1/7/2022',17.17
8190,'2/28/2022',88.27
8191,'2/5/2022',57.42
8192,'3/16/2022',53.80
8193,'4/1/2022',82.06
8194,'5/7/2022',40.39
8195,'5/16/2022',87.21
8196,'6/15/2022',95.93
8197,'6/26/2022',45.89
8198,'7/9/2022',36.23
8199,'7/22/2022',25.66
8200,'7/23/2022',82.77
8201,'7/27/2022',69.98
8202,'8/2/2022',76.11
8203,'8/8/2022',25.12
8204,'8/19/2022',46.23
8205,'9/26/2022',84.21
8206,'10/14/2022',96.24
8207,'10/29/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

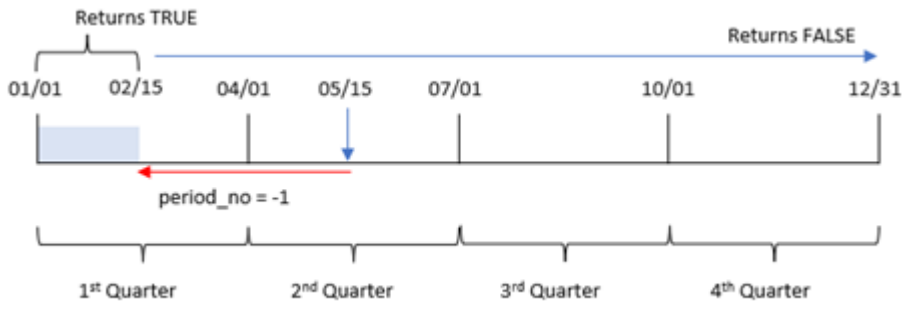
- `date`
- `previous_qtr_to_date`

Results table

date	previous_qtr_to_date
1/7/2022	-1
1/19/2022	-1
2/5/2022	-1
2/28/2022	0
3/16/2022	0
4/1/2022	0
5/7/2022	0
5/16/2022	0
6/15/2022	0
6/26/2022	0
7/9/2022	0
7/22/2022	0
7/23/2022	0
7/27/2022	0
8/2/2022	0
8/8/2022	0
8/19/2022	0
9/26/2022	0
10/14/2022	0
10/29/2022	0

A `period_no` value of -1 indicates that the `inquartertodate ()` function compares the input quarter segment to the preceding quarter. May 15 falls into the second quarter of the year, so the segment initially equates to between April 1 and May 15. The `period_no` then offsets this segment by three months earlier, causing the date boundaries to become January 1 to February 15.

Diagram of inquartertodate function, period_no example



Therefore, any transaction that occurs between January 1 and February 15 will return a Boolean result of`TRUE`.

Example 3 – first_month_of_year

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset and scenario as the first example.
- The creation of a field, `in_quarter_to_date`, that determines which transactions took place in the same quarter up to May 15, 2022.

In this example, we set March as the first month of the fiscal year.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
        *,
        inquartertodate(date,'05/15/2022', 0,3) as in_quarter_to_date
        ;
Load
*
Inline
[
id,date,amount
8188,'1/19/2022',37.23
8189,'1/7/2022',17.17
8190,'2/28/2022',88.27
8191,'2/5/2022',57.42
8192,'3/16/2022',53.80
8193,'4/1/2022',82.06
8194,'5/7/2022',40.39
```

```
8195,'5/16/2022',87.21
8196,'6/15/2022',95.93
8197,'6/26/2022',45.89
8198,'7/9/2022',36.23
8199,'7/22/2022',25.66
8200,'7/23/2022',82.77
8201,'7/27/2022',69.98
8202,'8/2/2022',76.11
8203,'8/8/2022',25.12
8204,'8/19/2022',46.23
8205,'9/26/2022',84.21
8206,'10/14/2022',96.24
8207,'10/29/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `in_quarter_to_date`

Results table

date	in_quarter_to_date
1/7/2022	0
1/19/2022	0
2/5/2022	0
2/28/2022	0
3/16/2022	-1
4/1/2022	-1
5/7/2022	-1
5/16/2022	0
6/15/2022	0
6/26/2022	0
7/9/2022	0
7/22/2022	0
7/23/2022	0
7/27/2022	0
8/2/2022	0
8/8/2022	0
8/19/2022	0

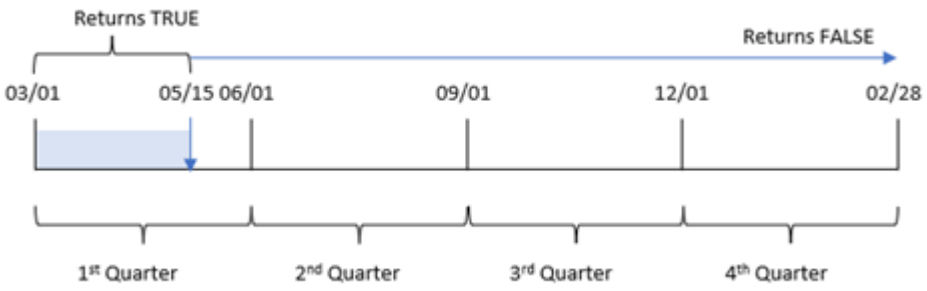
date	in_quarter_to_date
9/26/2022	0
10/14/2022	0
10/29/2022	0

By using 3 as the `first_month_of_year` argument in the `inquartertodate()` function, the function begins the year on March 1, and then divides the year into quarters. Therefore, the quarter segments are:

- March to May
- June to August
- September to November
- December to February

The `base_date` of May 15 then segments the March to May quarter by setting its end boundary as May 15.

Diagram of inquartertodate function, first_month_of_year example



Therefore, any transaction that occurs in between the March 1 and May 15 will return a Boolean result of TRUE, while transactions with dates outside these boundaries will return a value of FALSE.

Example 4 – Chart object example

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains the same dataset and scenario as the first example. However, in this example, the unchanged dataset is loaded into the application. The calculation that determines which transactions took place in the same quarter as May 15 is created as a measure in the chart object.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
Load
```

```
*
Inline
[
id,date,amount
8188,'1/19/2022',37.23
8189,'1/7/2022',17.17
8190,'2/28/2022',88.27
8191,'2/5/2022',57.42
8192,'3/16/2022',53.80
8193,'4/1/2022',82.06
8194,'5/7/2022',40.39
8195,'5/16/2022',87.21
8196,'6/15/2022',95.93
8197,'6/26/2022',45.89
8198,'7/9/2022',36.23
8199,'7/22/2022',25.66
8200,'7/23/2022',82.77
8201,'7/27/2022',69.98
8202,'8/2/2022',76.11
8203,'8/8/2022',25.12
8204,'8/19/2022',46.23
8205,'9/26/2022',84.21
8206,'10/14/2022',96.24
8207,'10/29/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension:`date`.

Create the following measure:

```
=inquartertodate(date,'05/15/2022', 0)
```

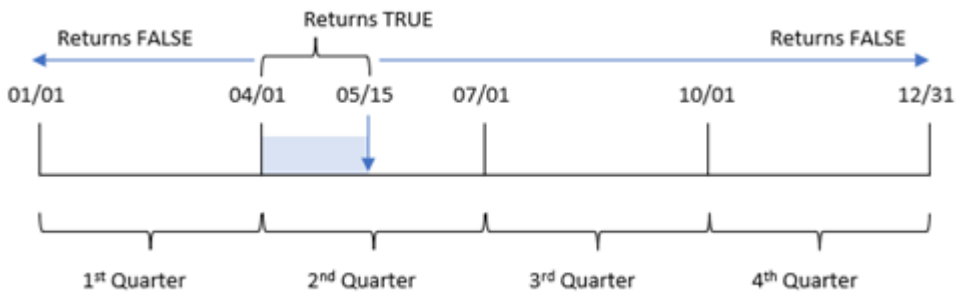
Results table

date	=inquartertodate(date,'05/15/2022', 0)
1/7/2022	0
1/19/2022	0
2/5/2022	0
2/28/2022	0
3/16/2022	0
4/1/2022	-1
5/7/2022	-1
5/16/2022	0
6/15/2022	0

date	=inquartertodate(date,'05/15/2022', 0)
6/26/2022	0
7/9/2022	0
7/22/2022	0
7/23/2022	0
7/27/2022	0
8/2/2022	0
8/8/2022	0
8/19/2022	0
9/26/2022	0
10/14/2022	0
10/29/2022	0

The `in_quarter_to_date` measure is created in a chart object by using the `inquartertodate()` function. The first argument is the date field being evaluated. The second argument is a hard-coded date for May 15, which is the `base_date` that identifies which quarter to segment and defines the end boundary of that segment. A `period_no` of 0 is the final argument, meaning that the function is not comparing quarters preceding or following the segmented quarter.

Diagram of inquartertodate function, chart object example



Any transaction that occurs between April 1 and May 15 returns a Boolean result of TRUE. Transactions on May 16 and later will return FALSE, as do any transactions before April 1.

Example 5 – Scenario

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset which is loaded into a table called `Products`.
- Information concerning product ID, manufacture date, and cost price.

On May 15, 2022, a piece of equipment error was identified in the manufacturing process and resolved. Products that were manufactured in that quarter up to this date will be defective. The end user would like a chart object that displays, by quarter name, the status of whether the product is ‘defective’ or ‘faultless’ and the cost of the products manufactured in that quarter to date.

Load script

```
Products:
Load
*
Inline
[
product_id,manufacture_date,cost_price
8188,'1/19/2022',37.23
8189,'1/7/2022',17.17
8190,'2/28/2022',88.27
8191,'2/5/2022',57.42
8192,'3/16/2022',53.80
8193,'4/1/2022',82.06
8194,'5/7/2022',40.39
8195,'5/16/2022',87.21
8196,'6/15/2022',95.93
8197,'6/26/2022',45.89
8198,'7/9/2022',36.23
8199,'7/22/2022',25.66
8200,'7/23/2022',82.77
8201,'7/27/2022',69.98
8202,'8/2/2022',76.11
8203,'8/8/2022',25.12
8204,'8/19/2022',46.23
8205,'9/26/2022',84.21
8206,'10/14/2022',96.24
8207,'10/29/2022',67.67
];
```

Results

Do the following:

1. Load the data and open a sheet. Create a new table. Create a dimension to show the quarter names: `=quartername(manufacture_date)`
2. Next, create a dimension to identify which of the products are defective and which are faultless: `=if(inquartertodate(manufacture_date,makedate(2022,05,15),0),'Defective','Faultless')`
3. Create a measure to sum the `cost_price` of the products: `=sum(cost_price)`
4. Set the measure's **Number formatting** to **Money**.

Results table

quartername (manufacture_date)	if(inquartertodate(manufacture_date,makedate (2022,05,15),0),'Defective','Faultless')	Sum(cost_ price)
Jan-Mar 2022	Faultless	$253.89
Apr-Jun 2022	Faultless	$229.03
Apr-Jun 2022	Defective	$122.45
Jul-Sep 2022	Faultless	$446.31
Oct-Dec 2022	Faultless	$163.91

The `inquartertodate()` function returns a Boolean value when evaluating the manufacturing dates of each of the products. For those that return a Boolean value of `TRUE`, it marks the products as `'Defective'`. For any product returning a value of `FALSE`, and therefore not made in the quarter up to and including May 15, it marks the products as `'Faultless'`.

inweek

This function returns True if **timestamp** lies inside the week containing **base_date**.

Syntax:

```
InWeek (timestamp, base_date, period_no[, first_week_day])
```

Arguments:

Examples and results:

Scripting examples

Example	Result
`inweek ('12/01/2006', '14/01/2006', 0)`	Returns True
`inweek ('12/01/2006', '20/01/2006', 0 )`	Returns False
`inweek ('12/01/2006', '14/01/2006', -1 )`	Returns False
`inweek ('07/01/2006', '14/01/2006', -1)`	Returns True
`inweek ('12/01/2006', '09/01/2006', 0, 3)`	Returns False Because first_week_day is specified as 3 (Thursday), which makes 12/01/2006 the first day of the week following the week containing 09/01/2006.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example checks if an invoice date falls at any time in the fourth week after the week in base_date, by specifying period_no as 4.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
InWeek(InvDate, '11/01/2013', 4) AS InWeekPlus4
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the inweek() function.

The InvDate5/2/2013 falls within the week that is four weeks after the base_date: 11/1/2013.

Results table

InvDate	InWeekPlus4
28/03/2012	0 (False)
10/12/2012	0 (False)
5/2/2013	-1 (True)
31/3/2013	0 (False)
19/5/2013	0 (False)
15/9/2013	0 (False)
11/12/2013	0 (False)
2/3/2014	0 (False)
14/5/2014	0 (False)
13/6/2014	0 (False)
7/7/2014	0 (False)
4/8/2014	0 (False)

inweektodate

This function returns True if **timestamp** lies inside the part of week containing **base_date** up until and including the last millisecond of **base_date**.

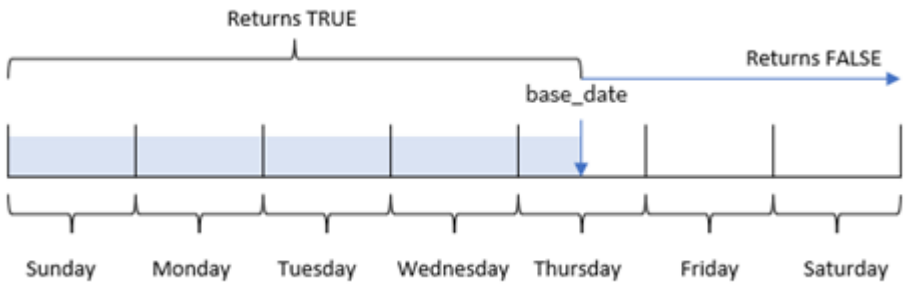
Syntax:

```
InWeekToDate (timestamp, base_date, period_no [, first_week_day])
```

Return data type: Boolean

> *In Qlik Sense, the Boolean true value is represented by -1, and the false value is represented by 0.*

Diagram of inweektodate function



The `inweektodate()` function uses the `base_date` parameter to identify a maximum boundary date of a week segment, as well as its corresponding date for the start of the week, which is based on the `FirstWeekDay` system variable (or user-defined `first_week_day` parameter). Once this week segment has been defined, the function will then return Boolean results when comparing the prescribed date values to that segment.

When to use it

The `inweektodate()` function returns a Boolean result. Typically, this type of function will be used as a condition in an `if` expression. This will return an aggregation or calculation dependent on whether a date evaluated occurred during the week in question up to and including a particular date.

For example, the `inweektodate()` function can be used to calculate all sales made during a specified week up to a particular date.

Arguments

Argument	Description
timestamp	The date that you want to compare with **base_date**.
base_date	Date that is used to evaluate the week.

Argument	Description
period_no	The week can be offset by **period_no**. **period_no** is an integer, where the value 0 indicates the week which contains **base_date**. Negative values in **period_no** indicate preceding weeks and positive values indicate succeeding weeks.
first_week_day	By default, the first day of the week is Sunday (as determined by the FirstWeekDay system variable), starting at midnight between Saturday and Sunday. The **first_week_day** parameter supersedes the **FirstWeekDay** variable. To indicate the week starting on another day, specify a flag between 0 and 6. For a week starting on Monday and ending on Sunday, use a flag of 0 for Monday, 1 for Tuesday, 2 for Wednesday, 3 for Thursday, 4 for Friday, 5 for Saturday, and 6 for Sunday.

Function examples

Example	Interaction
`inweektodate ('01/12/2006', '01/12/2006', 0)`	Returns `TRUE`.
`inweektodate ('01/12/2006', '01/11/2006', 0)`	Returns `FALSE`.
`inweektodate ('01/12/2006', '01/18/2006', -1)`	Returns `FALSE`. Because `period_no` is specified as -1, the effective data that `timestamp` is measured against is 01/11/2006.
`inweektodate ('01/11/2006', '01/12/2006', 0, 3 )`	Returns `FALSE`, since `first_week_day` is specified as `3` (Thursday), which makes 01/12/2006 the first day of the week following the week containing 01/12/2006.

These topics may help you work with this function:

Related topics

Topic	Default Flag / Value	Description
FirstWeekDay (page 182)	6 / Sunday	Defines the start day of each week.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 – No additional arguments

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions for the month of January 2022, which is loaded into a table called `Transactions`.
- The data field provided in the `TimestampFormat='M/D/YYYY h:mm:ss[.fff]'` format.
- The creation of a field, `in_week_to_date`, which determines which transactions took place in the week up until January 14, 2022.
- The creation of an additional field, named `weekday`, using the `weekday()` function. This new field is created to show which day of the week corresponds to each date.

Load script

```
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff]';
SET FirstWeekDay=6;
Transactions:
    Load
        *,
        weekday(date) as week_day,
        inweektodate(date,'01/14/2022', 0) as in_week_to_date
        ;
Load
*
Inline
[
id,date,amount
8188,'2022-01-02 12:22:06',37.23
8189,'2022-01-05 01:02:30',17.17
8190,'2022-01-06 15:36:20',88.27
8191,'2022-01-08 10:58:35',57.42
8192,'2022-01-09 08:53:32',53.80
8193,'2022-01-10 21:13:01',82.06
8194,'2022-01-11 00:57:13',40.39
8195,'2022-01-12 09:26:02',87.21
8196,'2022-01-13 15:05:09',95.93
8197,'2022-01-14 18:44:57',45.89
8198,'2022-01-15 06:10:46',36.23
8199,'2022-01-16 06:39:27',25.66
```

```
8200,'2022-01-17 10:44:16',82.77
8201,'2022-01-18 18:48:17',69.98
8202,'2022-01-26 04:36:03',76.11
8203,'2022-01-27 08:07:49',25.12
8204,'2022-01-28 12:24:29',46.23
8205,'2022-01-30 11:56:56',84.21
8206,'2022-01-30 14:40:19',96.24
8207,'2022-01-31 05:28:21',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `week_day`
- `in_week_to_date`

Results table

date	week_day	in_week_to_date
2022-01-02 12:22:06	Sun	0
2022-01-05 01:02:30	Wed	0
2022-01-06 15:36:20	Thu	0
2022-01-08 10:58:35	Sat	0
2022-01-09 08:53:32	Sun	-1
2022-01-10 21:13:01	Mon	-1
2022-01-11 00:57:13	Tue	-1
2022-01-12 09:26:02	Wed	-1
2022-01-13 15:05:09	Thu	-1
2022-01-14 18:44:57	Fri	-1
2022-01-15 06:10:46	Sat	0
2022-01-16 06:39:27	Sun	0
2022-01-17 10:44:16	Mon	0
2022-01-18 18:48:17	Tue	0
2022-01-26 04:36:03	Wed	0
2022-01-27 08:07:49	Thu	0
2022-01-28 12:24:29	Fri	0
2022-01-30 11:56:56	Sun	0

date	week_day	in_week_to_date
2022-01-30 14:40:19	Sun	0
2022-01-31 05:28:21	Mon	0

The `in_week_to_date` field is created in the preceding load statement by using the `inweektodate()` function. The first argument provided identifies which field is being evaluated. The second argument is a hard-coded date for January 14, which is the `base_date` that identifies which week to segment and defines the end boundary of that segment. A `period_no` of 0 is the final argument, meaning that the function is not comparing weeks preceding or following the segmented week.

The `FirstWeekDay` system variable determines that weeks begin on a Sunday and end on a Saturday. Therefore, January would be broken into weeks according to the diagram below, with the dates between January 9 and 14 providing the valid period for the `inweekdodate()` calculation:

Calendar diagram showing transaction dates which would return a Boolean result of TRUE

Sun	Mon	Tue	Wed	Thur	Fri	Sat
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Any transaction that occurs in between January 9 and 14 returns a Boolean result of `TRUE`. Transactions before and after the dates return a Boolean result of `FALSE`.

Example 2 – period_no

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset and scenario as the first example.
- The creation of a field, `prev_week_to_date`, that determines which transactions took place a full week before the week segment ending on January 14, 2022.

- The creation of an additional field, named `weekday`, using the `weekday()` function. This is to show which day of the week corresponds to each date.

Load script

```
SET FirstWeekDay=6;
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff]';
Transactions:
    Load
        *,
        weekday(date) as week_day,
        inweektodate(date,'01/14/2022', -1) as prev_week_to_date
        ;
Load
*
Inline
[
id,date,amount
8188,'2022-01-02 12:22:06',37.23
8189,'2022-01-05 01:02:30',17.17
8190,'2022-01-06 15:36:20',88.27
8191,'2022-01-08 10:58:35',57.42
8192,'2022-01-09 08:53:32',53.80
8193,'2022-01-10 21:13:01',82.06
8194,'2022-01-11 00:57:13',40.39
8195,'2022-01-12 09:26:02',87.21
8196,'2022-01-13 15:05:09',95.93
8197,'2022-01-14 18:44:57',45.89
8198,'2022-01-15 06:10:46',36.23
8199,'2022-01-16 06:39:27',25.66
8200,'2022-01-17 10:44:16',82.77
8201,'2022-01-18 18:48:17',69.98
8202,'2022-01-26 04:36:03',76.11
8203,'2022-01-27 08:07:49',25.12
8204,'2022-01-28 12:24:29',46.23
8205,'2022-01-30 11:56:56',84.21
8206,'2022-01-30 14:40:19',96.24
8207,'2022-01-31 05:28:21',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `week_day`
- `prev_week_to_date`

Results table

date	week_day	prev_week_to_date
2022-01-02 12:22:06	Sun	-1
2022-01-05 01:02:30	Wed	-1
2022-01-06 15:36:20	Thu	-1
2022-01-08 10:58:35	Sat	0
2022-01-09 08:53:32	Sun	0
2022-01-10 21:13:01	Mon	0
2022-01-11 00:57:13	Tue	0
2022-01-12 09:26:02	Wed	0
2022-01-13 15:05:09	Thu	0
2022-01-14 18:44:57	Fri	0
2022-01-15 06:10:46	Sat	0
2022-01-16 06:39:27	Sun	0
2022-01-17 10:44:16	Mon	0
2022-01-18 18:48:17	Tue	0
2022-01-26 04:36:03	Wed	0
2022-01-27 08:07:49	Thu	0
2022-01-28 12:24:29	Fri	0
2022-01-30 11:56:56	Sun	0
2022-01-30 14:40:19	Sun	0
2022-01-31 05:28:21	Mon	0

A `period_no` value of -1 indicates that the `inweektodate ()` function compares the input quarter segment to the preceding week. The week segment initially equates to between January 9 and January 14. The `period_no` then offsets both the start and end boundary of this segment to one week earlier, causing the date boundaries to become January 2 to January 7.

Calendar diagram showing transaction dates which would return a Boolean result of TRUE

Sun	Mon	Tue	Wed	Thu	Fri	Sat
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Therefore, any transaction that occurs between January 2 and 8 (not including January 8 itself) will return a Boolean result of TRUE.

Example 3 – first_week_day

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset and scenario as the first example.
- The creation of a field, `in_week_to_date`, that determines which transactions took place in the week up until January 14, 2022.
- The creation of an additional field, named `weekday`, using the `weekday()` function. This is to show which day of the week corresponds to each date.

In this example, we consider Monday as the first day of the week.

Load script

```
SET FirstWeekDay=6;
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff]';

Transactions:
    Load
        *,
        weekday(date) as week_day,
        inweektodate(date,'01/14/2022', 0, 0) as in_week_to_date
        ;
Load
*
```

```
Inline
[
id,date,amount
8188,'2022-01-02 12:22:06',37.23
8189,'2022-01-05 01:02:30',17.17
8190,'2022-01-06 15:36:20',88.27
8191,'2022-01-08 10:58:35',57.42
8192,'2022-01-09 08:53:32',53.80
8193,'2022-01-10 21:13:01',82.06
8194,'2022-01-11 00:57:13',40.39
8195,'2022-01-12 09:26:02',87.21
8196,'2022-01-13 15:05:09',95.93
8197,'2022-01-14 18:44:57',45.89
8198,'2022-01-15 06:10:46',36.23
8199,'2022-01-16 06:39:27',25.66
8200,'2022-01-17 10:44:16',82.77
8201,'2022-01-18 18:48:17',69.98
8202,'2022-01-26 04:36:03',76.11
8203,'2022-01-27 08:07:49',25.12
8204,'2022-01-28 12:24:29',46.23
8205,'2022-01-30 11:56:56',84.21
8206,'2022-01-30 14:40:19',96.24
8207,'2022-01-31 05:28:21',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `week_day`
- `in_week_to_date`

Results table

date	week_day	in_week_to_date
2022-01-02 12:22:06	Sun	0
2022-01-05 01:02:30	Wed	0
2022-01-06 15:36:20	Thu	0
2022-01-08 10:58:35	Sat	0
2022-01-09 08:53:32	Sun	0
2022-01-10 21:13:01	Mon	-1
2022-01-11 00:57:13	Tue	-1
2022-01-12 09:26:02	Wed	-1
2022-01-13 15:05:09	Thu	-1

date	week_day	in_week_to_date
2022-01-14 18:44:57	Fri	-1
2022-01-15 06:10:46	Sat	0
2022-01-16 06:39:27	Sun	0
2022-01-17 10:44:16	Mon	0
2022-01-18 18:48:17	Tue	0
2022-01-26 04:36:03	Wed	0
2022-01-27 08:07:49	Thu	0
2022-01-28 12:24:29	Fri	0
2022-01-30 11:56:56	Sun	0
2022-01-30 14:40:19	Sun	0
2022-01-31 05:28:21	Mon	0

By using 0 as the `first_week_day` argument in the `inweektodate()` function, the function argument supersedes the `FirstWeekDay` system variable and sets Monday as the first day of the week.

Calendar diagram showing transaction dates which would return a Boolean result of TRUE

Mon	Tue	Wed	Thu	Fri	Sat	Sun
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	17
24	25	26	27	28	29	30
31						

Therefore, any transaction that occurs in between January 10 and 14 will return a Boolean result of TRUE, while transactions with dates outside these boundaries will return a value of FALSE.

Example 4 – Chart object example

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains the same dataset and scenario as the first example. However, in this example, the unchanged dataset is loaded into the application. The calculation that determines which transactions took place in the week up until January 14, 2022 is created as a measure in the chart object.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
Load
*
Inline
[
id,date,amount
8188,'2022-01-02 12:22:06',37.23
8189,'2022-01-05 01:02:30',17.17
8190,'2022-01-06 15:36:20',88.27
8191,'2022-01-08 10:58:35',57.42
8192,'2022-01-09 08:53:32',53.80
8193,'2022-01-10 21:13:01',82.06
8194,'2022-01-11 00:57:13',40.39
8195,'2022-01-12 09:26:02',87.21
8196,'2022-01-13 15:05:09',95.93
8197,'2022-01-14 18:44:57',45.89
8198,'2022-01-15 06:10:46',36.23
8199,'2022-01-16 06:39:27',25.66
8200,'2022-01-17 10:44:16',82.77
8201,'2022-01-18 18:48:17',69.98
8202,'2022-01-26 04:36:03',76.11
8203,'2022-01-27 08:07:49',25.12
8204,'2022-01-28 12:24:29',46.23
8205,'2022-01-30 11:56:56',84.21
8206,'2022-01-30 14:40:19',96.24
8207,'2022-01-31 05:28:21',67.67
];
```

Results

Do the following:

1. Load the data and open a sheet. Create a new table and add this field as a dimension: `date`.
2. To calculate whether transactions took place in the same week up until the 14th of January, create the following measure:
   `=inweektodate(date,'01/14/2022',0)`
3. To show which day of the week corresponds to each date, create an additional measure:
   `=weekday(date)`

Results table

date	week_day	in_week_to_date
2022-01-02 12:22:06	Sun	0
2022-01-05 01:02:30	Wed	0
2022-01-06 15:36:20	Thu	0
2022-01-08 10:58:35	Sat	0
2022-01-09 08:53:32	Sun	-1
2022-01-10 21:13:01	Mon	-1
2022-01-11 00:57:13	Tue	-1
2022-01-12 09:26:02	Wed	-1
2022-01-13 15:05:09	Thu	-1
2022-01-14 18:44:57	Fri	-1
2022-01-15 06:10:46	Sat	0
2022-01-16 06:39:27	Sun	0
2022-01-17 10:44:16	Mon	0
2022-01-18 18:48:17	Tue	0
2022-01-26 04:36:03	Wed	0
2022-01-27 08:07:49	Thu	0
2022-01-28 12:24:29	Fri	0
2022-01-30 11:56:56	Sun	0
2022-01-30 14:40:19	Sun	0
2022-01-31 05:28:21	Mon	0

The `in_week_to_date` field is created as a measure in the chart object using the `inweektodate()` function. The first argument provided identifies which field is being evaluated. The second argument is a hard-coded date for January 14, which is the `base_date` that identifies which week to segment and defines the end boundary of that segment. A `period_no` of 0 is the final argument, meaning that the function is not comparing weeks preceding or following the segmented week.

The `FirstWeekDay` system variable determines that weeks begin on a Sunday and end on a Saturday. Therefore, January would be broken into weeks according to the diagram below, with the dates between January 9 and 14 providing the valid period for the `inweekdodate()` calculation:

Calendar diagram showing transaction dates which would return a Boolean result of TRUE

Sun	Mon	Tue	Wed	Thu	Fri	Sat
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Any transaction that occurs in between January 9 and 14 returns a Boolean result of `TRUE`. Transactions before and after the dates return a Boolean result of `FALSE`.

Example 5 – Scenario

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset which is loaded into a table called `Products`.
- Information concerning product ID, manufacture date, and cost price.

It has been identified that due to equipment error, products that were manufactured in the week of January 12 were defective. The issue was resolved on January 13. The end user would like a chart object that displays, by week, the status of whether the products manufactured are ‘defective’ or ‘faultless’, and the cost of the products manufactured in that week.

Load script

```
Products:
Load
*
Inline
[
product_id,manufacture_date,cost_price
8188,'2022-01-02 12:22:06',37.23
8189,'2022-01-05 01:02:30',17.17
8190,'2022-01-06 15:36:20',88.27
8191,'2022-01-08 10:58:35',57.42
8192,'2022-01-09 08:53:32',53.80
8193,'2022-01-10 21:13:01',82.06
8194,'2022-01-11 00:57:13',40.39
8195,'2022-01-12 09:26:02',87.21
8196,'2022-01-13 15:05:09',95.93
8197,'2022-01-14 18:44:57',45.89
8198,'2022-01-15 06:10:46',36.23
8199,'2022-01-16 06:39:27',25.66
8200,'2022-01-17 10:44:16',82.77
8201,'2022-01-18 18:48:17',69.98
8202,'2022-01-26 04:36:03',76.11
8203,'2022-01-27 08:07:49',25.12
8204,'2022-01-28 12:24:29',46.23
8205,'2022-01-30 11:56:56',84.21
8206,'2022-01-30 14:40:19',96.24
8207,'2022-01-31 05:28:21',67.67
];
```

Results

Do the following:

1. Load the data and open a sheet. Create a new table. Create a dimension to show the week names: `=weekname(manufacture_date)`
2. Next, create a dimension to identify which of the products are defective and which are faultless: `=if(inweektodate(manufacture_date,makedate(2022,01,12),0),'Defective','Faultless')`
3. Create a measure to sum the `cost_price` of the products: `=sum(cost_price)`
4. Set the measure's **Number formatting** to **Money**.

Results table

weekname(manufacture_date)	if(inweektodate(manufacture_date,makedate(2022,01,12),0),'Defective','Faultless')	Sum(cost_price)
2022/02	Faultless	$200.09
2022/03	Defective	$263.46
2022/03	Faultless	$178.05

weekname(manufacture_date)	if(inweektodate(manufacture_date,makedate (2022,01,12),0),'Defective','Faultless')	Sum(cost_price)
2022/04	Faultless	$178.41
2022/05	Faultless	$147.46
2022/06	Faultless	$248.12

The `inweektodate()` function returns a Boolean value when evaluating the manufacturing dates of each of the products. For those that return a Boolean value of `TRUE`, it marks the products as `'Defective'`. For any product returning a value of `FALSE`, and therefore not made in the week up to January 12, it marks the products as `'Faultless'`.

inyear

This function returns True if **timestamp** lies inside the year containing **base_date**.

Syntax:

```
InYear (timestamp, base_date, period_no [, first_month_of_year])
```

Arguments:

Arguments

Argument	Description
timestamp	The date that you want to compare with **base_date**.
base_date	Date that is used to evaluate the year.
period_no	The year can be offset by **period_no**. **period_no** is an integer, where the value 0 indicates the year that contains **base_date**. Negative values in **period_no** indicate preceding years, and positive values indicate succeeding years.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`inyear ('25/01/2013', '01/01/2013', 0 )`	Returns True
`inyear ('25/01/2012', '01/01/2013', 0)`	Returns False
`inyear ('25/01/2013', '01/01/2013', -1)`	Returns False

Example	Result
`inyear ('25/01/2012', '01/01/2013', -1 )`	Returns True
`inyear ('25/01/2013', '01/01/2013', 0, 3)`	Returns True The value of base_date and first_month_of_year specify that timestamp must fall within 01/03/2012 and 28/02/2013
`inyear ('25/03/2013', '01/07/2013', 0, 3 )`	Returns True

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example checks if an invoice date falls in the fiscal year specified by setting the value of first_month_of_year to 4, and having the base_date between 1/4/2012 and 31/03/2013.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];
```

Test if InvDate is in the financial year 1/04/2012 to 31/03/2013:

```
InvoiceData:
LOAD *,
InYear(InvDate, '31/01/2013', 0, 4) AS FinYr1213
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the inyear() function.

Results table

InvDate	FinYr1213
28/03/2012	0 (False)
10/12/2012	-1 (True)

InvDate	FinYr1213
5/2/2013	-1 (True)
31/3/2013	-1 (True)
19/5/2013	0 (False)
15/9/2013	0 (False)
11/12/2013	0 (False)
2/3/2014	0 (False)
14/5/2014	0 (False)
13/6/2014	0 (False)
7/7/2014	0 (False)
4/8/2014	0 (False)

inyeartodate

This function returns True if **timestamp** lies inside the part of year containing **base_date** up until and including the last millisecond of **base_date**.

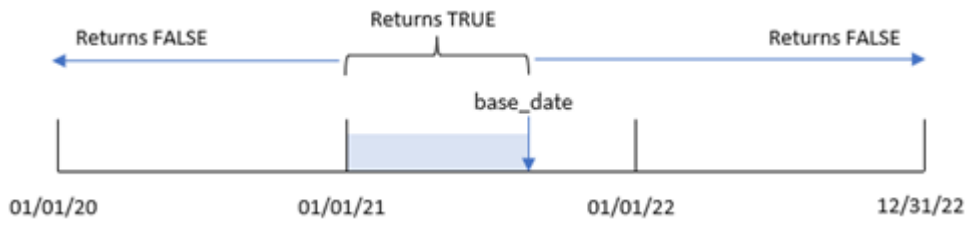
Syntax:

```
InYearToDate (timestamp, base_date, period_no[, first_month_of_year])
```

Return data type: Boolean

In Qlik Sense, the Boolean true value is represented by -1, and the false value is represented by 0.

Diagram of inyeartodate function



The `inyeartodate()` function will segment a particular portion of the year with the `base_date`, identifying the maximum allowed date for that year segment. The function then evaluates whether a date field or value falls into this segment and returns a Boolean result.

Arguments

Argument	Description
timestamp	The date that you want to compare with **base_date**.

Argument	Description
base_date	Date that is used to evaluate the year.
period_no	The year can be offset by **period_no**. **period_no** is an integer, where the value 0 indicates the year that contains **base_date**. Negative values in **period_no** indicate preceding years, and positive values indicate succeeding years.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

When to use it

The `inyeartodate()` function returns a Boolean result. Typically, this type of function will be used as a condition in an `if` expression. This would return an aggregation or calculation dependent on whether a date evaluated occurred in the year up to and including the date in question.

For example, the `inyeartodate()` function can be used to identify all equipment manufactured in a year up to a specific date.

These examples use the date format MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement at the top of your data load script. Change the format in the examples to suit your requirements.

Function examples

Example	Result
`inyeartodate ('01/25/2013', '02/01/2013', 0)`	Returns `TRUE`.
`inyeartodate ('01/25/2012', '01/01/2013', 0)`	Returns `FALSE`.
`inyeartodate ('01/25/2012', '02/01/2013', -1)`	Returns `TRUE`.
`inyeartodate ('11/25/2012', '01/31/2013', 0, 4)`	Returns `TRUE`. The value of `timestamp` falls inside the fiscal year beginning in the fourth month and before the value of `base_date`.
`inyeartodate ('3/31/2013', '01/31/2013', 0, 4 )`	Returns `FALSE`. Compared with the previous example, the value of `timestamp` is still inside the fiscal year, but it is after the value of `base_date`, so it falls outside the part of the year.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may

be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 – No additional arguments

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions between 2020 and 2022, which is loaded into a table called `Transactions`.
- The date field provided in the `DateFormat` system variable (MM/DD/YYYY) format.
- The creation of a field, `in_year_to_date`, that determines which transactions took place in the year up until July 26, 2021.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
        *,
        inyeartodate(date,'07/26/2021', 0) as in_year_to_date
        ;
Load
*
Inline
[
id,date,amount
8188,'01/13/2020',37.23
8189,'02/26/2020',17.17
8190,'03/27/2020',88.27
8191,'04/16/2020',57.42
8192,'05/21/2020',53.80
8193,'06/14/2020',82.06
8194,'08/07/2020',40.39
8195,'09/05/2020',87.21
8196,'01/22/2021',95.93
8197,'02/03/2021',45.89
8198,'03/17/2021',36.23
```

```
8199,'04/23/2021',25.66
8200,'05/04/2021',82.77
8201,'06/30/2021',69.98
8202,'07/26/2021',76.11
8203,'07/27/2021',25.12
8204,'06/06/2022',46.23
8205,'07/18/2022',84.21
8206,'11/14/2022',96.24
8207,'12/12/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `in_year_to_date`

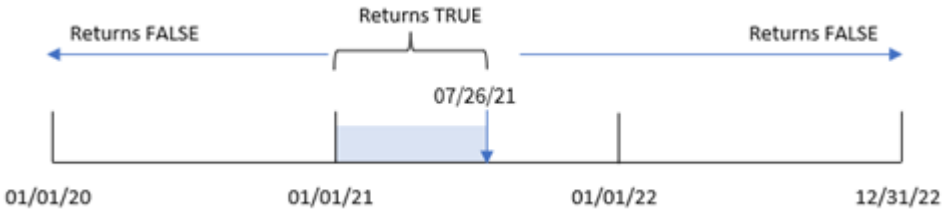
Results table

date	in_year_to_date
01/13/2020	0
02/26/2020	0
03/27/2020	0
04/16/2020	0
05/21/2020	0
06/14/2020	0
08/07/2020	0
09/05/2020	0
01/22/2021	-1
02/03/2021	-1
03/17/2021	-1
04/23/2021	-1
05/04/2021	-1
06/30/2021	-1
07/26/2021	-1
07/27/2021	0
06/06/2022	0
07/18/2022	0
11/14/2022	0
12/12/2022	0

The `in_year_to_date` field is created in the preceding load statement by using the `inyeartodate()` function. The first argument provided identifies which field is being evaluated.

The second argument is a hard-coded date for the for the July 26, 2021, which is the `base_date` that identifies the end boundary of the year segment. A `period_no` of 0 is the final argument, meaning that the function is not comparing years preceding or following the segmented year.

Diagram of inyeartodate function, no additional arguments



Any transaction that occurs in between January 1 and July 26 returns a Boolean result of TRUE. Transactions dates before 2021 and beyond July 26, 2021 return FALSE.

Example 2 – period_no

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset and scenario as the first example.
- The creation of a field, `previous_year_to_date`, that determines which transactions took place a full year before the year segment ending on July 26, 2021.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
        *,
        inyeartodate(date,'07/26/2021', -1) as previous_year_to_date
        ;
Load
*
Inline
[
id,date,amount
8188,'01/13/2020',37.23
8189,'02/26/2020',17.17
8190,'03/27/2020',88.27
8191,'04/16/2020',57.42
```

```
8192,'05/21/2020',53.80
8193,'06/14/2020',82.06
8194,'08/07/2020',40.39
8195,'09/05/2020',87.21
8196,'01/22/2021',95.93
8197,'02/03/2021',45.89
8198,'03/17/2021',36.23
8199,'04/23/2021',25.66
8200,'05/04/2021',82.77
8201,'06/30/2021',69.98
8202,'07/26/2021',76.11
8203,'07/27/2021',25.12
8204,'06/06/2022',46.23
8205,'07/18/2022',84.21
8206,'11/14/2022',96.24
8207,'12/12/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `previous_year_to_date`

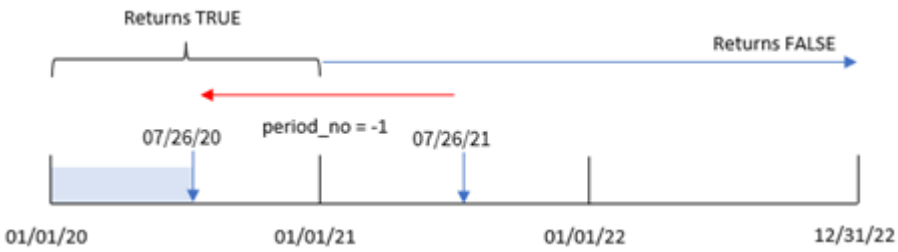
Results table

date	previous_year_to_date
01/13/2020	-1
02/26/2020	-1
03/27/2020	-1
04/16/2020	-1
05/21/2020	-1
06/14/2020	-1
08/07/2020	0
09/05/2020	0
01/22/2021	0
02/03/2021	0
03/17/2021	0
04/23/2021	0
05/04/2021	0
06/30/2021	0

date	previous_year_to_date
07/26/2021	0
07/27/2021	0
06/06/2022	0
07/18/2022	0
11/14/2022	0
12/12/2022	0

A `period_no` value of -1 indicates that the `inyeartodate ()` function compares the input quarter segment to the preceding year. With an input date of July 26, 2021, the segment from January 1, 2021 to July 26, 2021 was initially identified as the year-to-date. The `period_no` then offsets this segment by a full year earlier, causing the date boundaries to become January 1 to July 26, 2020.

Diagram of inyeartodate function, period_no example



Therefore, any transaction that occurs between January 1 and July 26, 2020 will return a Boolean result of TRUE.

Example 3 – first_month_of_year

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset and scenario as the first example.
- The creation of a field, `in_year_to_date`, that determines which transactions took place in the same year up to July 26, 2021.

In this example, we set March as the first month of the fiscal year.

Load script

```
SET DateFormat='MM/DD/YYYY';
```

```
Transactions:
    Load
        *,
        inyeartodate(date,'07/26/2021', 0,3) as in_year_to_date
        ;
Load
*
Inline
[
id,date,amount
8188,'01/13/2020',37.23
8189,'02/26/2020',17.17
8190,'03/27/2020',88.27
8191,'04/16/2020',57.42
8192,'05/21/2020',53.80
8193,'06/14/2020',82.06
8194,'08/07/2020',40.39
8195,'09/05/2020',87.21
8196,'01/22/2021',95.93
8197,'02/03/2021',45.89
8198,'03/17/2021',36.23
8199,'04/23/2021',25.66
8200,'05/04/2021',82.77
8201,'06/30/2021',69.98
8202,'07/26/2021',76.11
8203,'07/27/2021',25.12
8204,'06/06/2022',46.23
8205,'07/18/2022',84.21
8206,'11/14/2022',96.24
8207,'12/12/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `in_year_to_date`

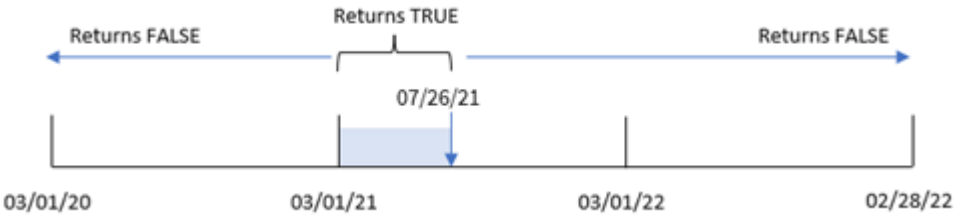
Results table

date	in_year_to_date
01/13/2020	0
02/26/2020	0
03/27/2020	0
04/16/2020	0
05/21/2020	0
06/14/2020	0

date	in_year_to_date
08/07/2020	0
09/05/2020	0
01/22/2021	0
02/03/2021	0
03/17/2021	-1
04/23/2021	-1
05/04/2021	-1
06/30/2021	-1
07/26/2021	-1
07/27/2021	0
06/06/2022	0
07/18/2022	0
11/14/2022	0
12/12/2022	0

By using 3 as the `first_month_of_year` argument in the `inyeartodate()` function, the function begins the year on March 1. The `base_date` of July 26, 2021 then sets the end date for that year segment.

Diagram of inyeartodate function, first_month_of_year example



Therefore, any transaction that occurs between March 1 and July 26, 2021 will return a Boolean result of TRUE, while transactions with dates outside these boundaries will return a value of FALSE.

Example 4 – Chart object example

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains the same dataset and scenario as the first example. However, in this example, the unchanged dataset is loaded into the application. The calculation that determines which transactions took place in the same year up to July 26, 2021 is created as a measure in a chart object in the application.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
Load
*
Inline
[
id,date,amount
8188,'01/13/2020',37.23
8189,'02/26/2020',17.17
8190,'03/27/2020',88.27
8191,'04/16/2020',57.42
8192,'05/21/2020',53.80
8193,'06/14/2020',82.06
8194,'08/07/2020',40.39
8195,'09/05/2020',87.21
8196,'01/22/2021',95.93
8197,'02/03/2021',45.89
8198,'03/17/2021',36.23
8199,'04/23/2021',25.66
8200,'05/04/2021',82.77
8201,'06/30/2021',69.98
8202,'07/26/2021',76.11
8203,'07/27/2021',25.12
8204,'06/06/2022',46.23
8205,'07/18/2022',84.21
8206,'11/14/2022',96.24
8207,'12/12/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension:`date`.

Create the following measure:

```
=inyeartodate(date,'07/26/2021', 0)
```

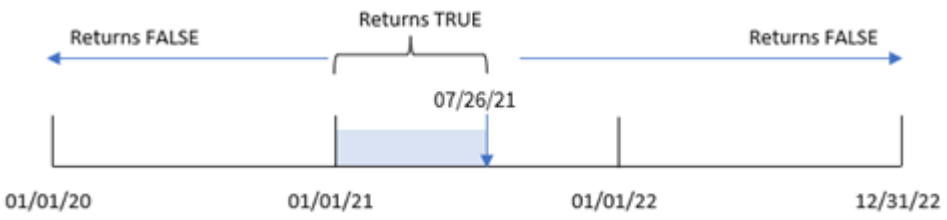
Results table

date	=inyeartodate(date,'07/26/2021', 0)
01/13/2020	0
02/26/2020	0
03/27/2020	0

date	=inyeartodate(date,'07/26/2021', 0)
04/16/2020	0
05/21/2020	0
06/14/2020	0
08/07/2020	0
09/05/2020	0
01/22/2021	-1
02/03/2021	-1
03/17/2021	-1
04/23/2021	-1
05/04/2021	-1
06/30/2021	-1
07/26/2021	-1
07/27/2021	0
06/06/2022	0
07/18/2022	0
11/14/2022	0
12/12/2022	0

The `in_year_to_date` measure is created in the chart object by using the `inyeartodate()` function. The first argument provided identifies which field is being evaluated. The second argument is a hard-coded date for July 26, 2021, which is the `base_date` that identifies the end boundary of the comparator year segment. A `period_no` of 0 is the final argument, meaning that the function is not comparing years preceding or following the segmented year.

Diagram of inyeartodate function, chart object example



Any transaction that occurs between January 1 and July 26, 2021 returns a Boolean result of TRUE. Transaction dates before 2021 and after July 26, 2021 return FALSE.

Example 5 – Scenario

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset which is loaded into a table called `Products`.
- Information concerning product ID, product type, manufacture date, and cost price.

The end user would like a chart object that displays, by product type, the cost of the products manufactured in 2021 up to July 26.

Load script

```
Products:
Load
*
Inline
[
product_id,product_type,manufacture_date,cost_price
8188,product A,'01/13/2020',37.23
8189,product B,'02/26/2020',17.17
8190,product B,'03/27/2020',88.27
8191,product C,'04/16/2020',57.42
8192,product D,'05/21/2020',53.80
8193,product D,'08/14/2020',82.06
8194,product C,'10/07/2020',40.39
8195,product B,'12/05/2020',87.21
8196,product A,'01/22/2021',95.93
8197,product B,'02/03/2021',45.89
8198,product C,'03/17/2021',36.23
8199,product C,'04/23/2021',25.66
8200,product B,'05/04/2021',82.77
8201,product D,'06/30/2021',69.98
8202,product D,'07/26/2021',76.11
8203,product D,'12/27/2021',25.12
8204,product C,'06/06/2022',46.23
8205,product C,'07/18/2022',84.21
8206,product A,'11/14/2022',96.24
8207,product B,'12/12/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension:`product_type`.

Create a measure that calculates the sum of each product that was manufactured in 2021 before July 27:

```
=sum(if(inyeartodate(manufacture_date,makedate(2021,07,26),0),cost_price,0))
```

Set the measure's **Number formatting** to **Money**.

Results table

product_type	=sum(if(inyeartodate(manufacture_date,makedate (2021,07,26),0),cost_price,0))
product A	$95.93
product B	$128.66
product C	$61.89
product D	$146.09

The `inyeartodate()` function returns a Boolean value when evaluating the manufacturing dates of each of the products. For any product manufactured in 2021 before July 27, the `inyeartodate()` function returns a Boolean value of `TRUE` and sums the `cost_price`.

Product D is the only product that was also manufactured after July 26th in 2021. The entry with `product_ID` 8203 was manufactured on December 27 and cost $25.12. Therefore, this cost was not included in the total for Product D in the chart object.

lastworkdate

The **lastworkdate** function returns the earliest ending date to achieve **no_of_workdays** (Monday-Friday) if starting at **start_date** taking into account any optionally listed **holiday**. **start_date** and **holiday** should be valid dates or timestamps.

Syntax:

```
lastworkdate(start_date, no_of_workdays {, holiday})
```

Arguments:

Arguments

Argument	Description
start_date	The start date to evaluate.
no_of_workdays	The number of working days to achieve.
holiday	Holiday periods to exclude from working days. A holiday period is stated as a start date and an end date, separated by commas. **Example:** `'25/12/2013', '26/12/2013'` You can specify more than one holiday period, separated by commas. **Example:** `'25/12/2013', '26/12/2013', '31/12/2013', '01/01/2014'` Holiday periods to exclude from working days. A holiday is stated as a string constant date. You can specify multiple holiday dates, separated by commas.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`lastworkdate ('19/12/2014', 9)`	Returns '31/12/2014'
`lastworkdate ('19/12/2014', 9, '2014-12-25', '2014-12-26')`	Returns '02/01/2015 as a holiday period of two days is taken into account.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
ProjectTable:
LOAD *, recno() as InvID, INLINE [
StartDate
28/03/2014
10/12/2014
5/2/2015
31/3/2015
19/5/2015
15/9/2015
] ;
NrDays:
Load *,
LastWorkDate(StartDate,120) As EndDate
Resident ProjectTable;
Drop table ProjectTable;
```

The resulting table shows the returned values of LastWorkDate for each of the records in the table.

Results table

InvID	StartDate	EndDate
1	28/03/2014	11/09/2014
2	10/12/2014	26/05/2015
3	5/2/2015	27/07/2015
4	31/3/2015	14/09/2015
5	19/5/2015	02/11/2015
6	15/9/2015	29/02/2016

localtime

This function returns a timestamp of the current time for a specified time zone.

Syntax:

```
LocalTime([timezone [, ignoreDST ]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
timezone	The **timezone** is specified as a string containing any of the geographical places listed under **Time Zone** in the **Windows Control Panel** for **Date and Time** or as a string in the form 'GMT+hh:mm'. If no time zone is specified the local time will be returned.
ignoreDST	If **ignoreDST** is -1 (True) daylight savings time will be ignored.

Examples and results:

The examples below are based on the function being called on 2014-10-22 12:54:47 local time, with the local time zone being GMT+01:00.

Scripting examples

Example	Result
`localtime ()`	Returns the local time 2014-10-22 12:54:47.
`localtime ('London')`	Returns the local time in London, 2014-10-22 11:54:47.
`localtime ('GMT+02:00')`	Returns the local time in the timezone of GMT+02:00, 2014-10-22 13:54:47.
`localtime ('Paris','-1')`	Returns the local time in Paris with daylight savings time ignored, 2014-10-22 11:54:47.

lunarweekend

This function returns a value corresponding to a timestamp of the last millisecond of the lunar week containing **date**. Lunar weeks in Qlik Sense are defined by counting 1 January as the first day of the week.

Syntax:

```
LunarweekEnd(date[, period_no[, first_week_day]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
date	The date to evaluate.
period_no	**period_no** is an integer or expression resolving to an integer, where the value 0 indicates the lunar week which contains **date**. Negative values in **period_no** indicate preceding lunar weeks and positive values indicate succeeding lunar weeks.
first_week_day	An offset that may be greater than or less than zero. This changes the beginning of the year by the specified number of days and/or fractions of a day.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`lunarweekend('12/01/2013')`	Returns 14/01/2013 23:59:59.
`lunarweekend('12/01/2013', -1)`	Returns 7/01/2013 23:59:59.
`lunarweekend('12/01/2013', 0, 1)`	Returns 15/01/2013 23:59:59.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example finds the final day of the lunar week of each invoice date in the table, where the date is shifted by one week by specifying period_no as 1.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
```

```
];

InvoiceData:
LOAD *,
LunarWeekEnd(InvDate, 1) AS LWkEnd
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the lunarweekend() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	LWkEnd
28/03/2012	07/04/2012
10/12/2012	22/12/2012
5/2/2013	18/02/2013
31/3/2013	08/04/2013
19/5/2013	27/05/2013
15/9/2013	23/09/2013
11/12/2013	23/12/2013
2/3/2014	11/03/2014
14/5/2014	27/05/2014
13/6/2014	24/06/2014
7/7/2014	15/07/2014
4/8/2014	12/08/2014

lunarweekname

This function returns a display value showing the year and lunar week number corresponding to a timestamp of the first millisecond of the first day of the lunar week containing **date**. Lunar weeks in Qlik Sense are defined by counting 1 January as the first day of the week.

Syntax:

```
LunarWeekName(date [, period_no[, first_week_day]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
date	The date to evaluate.
period_no	**period_no** is an integer or expression resolving to an integer, where the value 0 indicates the lunar week which contains **date**. Negative values in **period_no** indicate preceding lunar weeks and positive values indicate succeeding lunar weeks.
first_week_day	An offset that may be greater than or less than zero. This changes the beginning of the year by the specified number of days and/or fractions of a day.

Examples and results:

Scripting examples

Example	Result
`lunarweekname('12/01/2013')`	Returns 2006/02.
`lunarweekname('12/01/2013', -1)`	Returns 2006/01.
`lunarweekname('12/01/2013', 0, 1)`	Returns 2006/02.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

In this example, for each invoice date in the table, the lunar week name is created from the year in which the week lies and its associated lunar week number, shifted one week by specifying period_no as 1.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
```

```
LunarWeekName(InvDate, 1) AS LWkName
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the lunarweekname() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	LWkName
28/03/2012	2012/14
10/12/2012	2012/51
5/2/2013	2013/07
31/3/2013	2013/14
19/5/2013	2013/21
15/9/2013	2013/38
11/12/2013	2013/51
2/3/2014	2014/10
14/5/2014	2014/21
13/6/2014	2014/25
7/7/2014	2014/28
4/8/2014	2014/32

lunarweekstart

This function returns a value corresponding to a timestamp of the first millisecond of the lunar week containing **date**. Lunar weeks in Qlik Sense are defined by counting 1 January as the first day of the week.

Syntax:

```
LunarweekStart(date[, period_no[, first_week_day]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
date	The date to evaluate.

Argument	Description
period_no	**period_no** is an integer or expression resolving to an integer, where the value 0 indicates the lunar week which contains **date**. Negative values in **period_no** indicate preceding lunar weeks and positive values indicate succeeding lunar weeks.
first_week_day	An offset that may be greater than or less than zero. This changes the beginning of the year by the specified number of days and/or fractions of a day.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`lunarweekstart ('12/01/2013')`	Returns 08/01/2013.
`lunarweekstart ('12/01/2013', -1)`	Returns 01/01/2013.
`lunarweekstart ('12/01/2013', 0, 1 )`	Returns 09/01/2013. Because the offset specified by setting first_week_day to 1 means the beginning of the year is changed to 02/01/2013.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example finds the first day of the lunar week of each invoice date in the table, where the date is shifted by one week by specifying period_no as 1.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
LunarWeekStart(InvDate, 1) AS LWkStart
```

```
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the lunarweekstart() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	LWkStart
28/03/2012	01/04/2012
10/12/2012	16/12/2012
5/2/2013	12/02/2013
31/3/2013	02/04/2013
19/5/2013	21/05/2013
15/9/2013	17/09/2013
11/12/2013	17/12/2013
2/3/2014	05/03/2014
14/5/2014	21/05/2014
13/6/2014	18/06/2014
7/7/2014	09/07/2014
4/8/2014	06/08/2014

makedate

This function returns a date calculated from the year **YYYY**, the month **MM** and the day **DD**.

Syntax:

```
MakeDate(YYYY [ , MM [ , DD ] ])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
YYYY	The year as an integer.
MM	The month as an integer. If no month is stated, 1 (January) is assumed.
DD	The day as an integer. If no day is stated, 1 (the 1st) is assumed.

Example: Chart expression

Chart expressions examples

Example	Result
`makedate(2012)`	returns 2012-01-01
`makedate(12)`	returns 0012-01-01
`makedate(2012,12)`	returns 2012-12-01
`makedate(2012,2,14)`	returns 2012-02-14

Example: Load script

makedate can be used in load script to combine date data from different fields, into one new date field. In the example below the year, month, and day data from fields transaction_year, transaction_month, and transaction_day are combined into a new field called Transaction Date.

In the **Data load editor**, create a new section, and then add the example script and run it. Then add, at least, the fields listed in the results column to a sheet in your app to see the result.

Load script

```
SET DateFormat='DD/MM/YYYY';
SET TimestampFormat='DD/MM/YYYY h:mm:ss[.fff] TT';
SET FirstWeekDay=0;
SET BrokenWeeks=1;
SET ReferenceDay=0;
SET DayNames='Mon;Tue;Wed;Thu;Fri;Sat;Sun';
SET LongDayNames='Monday;Tuesday;Wednesday;Thursday;Friday;Saturday;Sunday';

Transactions:
Load
*,
MakeDate(transaction_year, transaction_month, transaction_day) as "Transaction Date",
;

Load * Inline [
transaction_id, transaction_year, transaction_month, transaction_day, transaction_amount,
transaction_quantity, discount, customer_id, size, color_code
3750, 2018, 08, 30, 12423.56, 23, 0,2038593, L, Red
3751, 2018, 09, 07, 5356.31, 6, 0.1, 203521, m, orange
3752, 2018, 09, 16, 15.75, 1, 0.22, 5646471, S, blue
3753, 2018, 09, 22, 1251, 7, 0, 3036491, l, Black
3754, 2018, 09, 22, 21484.21, 1356, 75, 049681, xs, Red
3756, 2018, 09, 22, -59.18, 2, 0.333333333333333, 2038593, M, Blue
3757, 2018, 09, 23, 3177.4, 21, .14, 203521, XL, Black
];
```

Results

Qlik Sense table showing results of the makedate function being used in the load script.

transaction_id	Transaction Date
3750	30/08/2018
3751	07/09/2018
3752	16/09/2018
3753	22/09/2018
3754	22/09/2018
3756	22/09/2018
3757	23/09/2018

maketime

This function returns a time calculated from the hour **hh**, the minute **mm**, and the second **ss**.

Syntax:

```
MakeTime(hh [ , mm [ , ss ] ])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
hh	The hour as an integer.
mm	The minute as an integer. If no minute is stated, 00 is assumed.
ss	The second as an integer. If no second is stated, 00 is assumed.

Examples and results:

Scripting examples

Example	Result
`maketime( 22 )`	returns 22:00:00

Example	Result
`maketime( 22, 17 )`	returns 22:17:00
`maketime( 22, 17, 52 )`	returns 22:17:52

makeweekdate

This function returns a date calculated from the year, the week number, and the day of week .

Syntax:

```
MakeWeekDate(weekyear [, week [, weekday [, first_week_day [, broken_weeks [, reference_day]]]]])
```

Return data type: dual

The `makeweekdate()` function is available both as script and chart function. The function will calculate the date based on the parameters passed into the function.

Arguments

Argument	Description
weekyear	The year as defined by the `weekYear()` function for the specific date, that is the year to which the week number belongs. *The week year can in some cases be different from the calendar year, for example if week 1 starts already in December of the previous year.*
week	The week number as defined by the `Week()` function for the specific date. If no week number is stated, 1 is assumed.

Argument	Description
weekday	The day-of-week as defined by the `WeekDay()` function for the date in question. 0 is the first day of the week, and 6 is the last day of the week. If no day-of-week is stated, 0 is assumed. *Even though 0 always means first day of the week and 6 is always the last, which weekdays that corresponds to is determined by the **first_week_day** parameter. If omitted, the value of variable **FirstWeekDay** is used.* If broken weeks are used, together with an impossible combination of parameters, this may lead to a result that does not belong to the chosen year. **Example:** `MakeWeekDate(2021,1,0,6,1)` Returns 'Dec 27 2020' since this day is the first day (the Sunday) of the specified week. Jan 1 2021 was a Friday.
first_week_day	Specifies the day on which the week starts. If omitted, the value of variable **FirstWeekDay** is used. The possible values **first_week_day** are 0 for Monday, 1 for Tuesday, 2 for Wednesday, 3 for Thursday, 4 for Friday, 5 for Saturday, and 6 for Sunday. For more information about the system variable, see *FirstWeekDay (page 182)*.
broken_weeks	If you don't specify **broken_weeks**, the value of variable **BrokenWeeks** is used to define whether weeks are broken or not.
reference_day	If you don't specify **reference_day**, the value of variable **ReferenceDay** is used to define which day in January to set as reference day to define week 1.

When to use it

The `makeweekdate()` function would commonly be used in the script for data generation to generate a list of dates, or to construct dates when the year, week and day-of-week are provided in the input data.

The following examples assume:

```
SET FirstWeekDay=0;
SET BrokenWeeks=0;
SET ReferenceDay=4;
```

Function examples

Example	Result
`makeweekdate(2014,6,6)`	returns 02/09/2014
`makeweekdate(2014,6,1)`	returns 02/04/2014
`makeweekdate(2014,6)`	returns 02/03/2014 (weekday 0 is assumed)

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 – day included

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing weekly sales total for 2022 in a table called `Sales`.
- Transaction dates provided across three fields: `year`, `week`, and `sales`.
- A preceding load, which is used to create a measure, `end_of_week`, using the `makeweekdate()` function to return the date for the Friday of that week in the format MM/DD/YYYY.

To prove that the date returned is a Friday, the `end_of_week` expression is also wrapped in the `weekday()` function to show the day of the week.

Load script

```
SET DateFormat='MM/DD/YYYY';
SET FirstWeekDay=0;
SET BrokenWeeks=0;
SET ReferenceDay=4;

Transactions:
    Load
        *,
        makeweekdate(transaction_year, transaction_week,4) as end_of_week,
        weekday(makeweekdate(transaction_year, transaction_week,4)) as week_day
        ;
Load * Inline [
transaction_year, transaction_week, sales
2022, 01, 10000
2022, 02, 11250
2022, 03, 9830
```

```
2022, 04, 14010
2022, 05, 28402
2022, 06, 9992
2022, 07, 7292
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `transaction_year`
- `transaction_week`
- `end_of_week`
- `week_day`

Results table

transaction_year	transaction_week	end_of_week	week_day
2022	01	01/07/2022	Fri
2022	02	01/14/2022	Fri
2022	03	01/21/2022	Fri
2022	04	01/28/2022	Fri
2022	05	02/04/2022	Fri
2022	06	02/11/2022	Fri
2022	07	02/18/2022	Fri

The `end_of_week` field is created in the preceding load statement by using the `makeweekdate()` function. The `transaction_year`, `transaction_week` fields are passed through the function as the year and week arguments. A value of 4 is used for the day argument.

The function then combines and converts these values into a date field, returning the results in the format of the `DateFormat` system variable.

The `makeweekdate()` function, and its arguments are also wrapped in a `weekday()` function to return the `week_day` field; and as can be seen in the table above, the `week_day` field shows that these dates do occur on a Friday.

Example 2 – day excluded

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing weekly sales totals for 2022 in a table called `Sales`.
- Transaction dates provided across three fields: `year`, `week`, and `sales`.
- A preceding load, which is used to create a measure, `first_day_of_week`, using the `makeweekdate()` function. This will return the date for the Monday of that week in the format MM/DD/YYYY.

To prove that the date returned is a Monday, the `first_day_of_week` expression is also wrapped in the `weekday()` function to show the day of the week.

Load script

```
SET DateFormat='MM/DD/YYYY';
SET FirstWeekDay=0;
SET BrokenWeeks=0;
SET ReferenceDay=4;

Transactions:
    Load
        *,
        makeweekdate(transaction_year, transaction_week) as first_day_of_week,
        weekday(makeweekdate(transaction_year, transaction_week)) as week_day
        ;
Load * Inline [
transaction_year, transaction_week, sales
2022, 01, 10000
2022, 02, 11250
2022, 03, 9830
2022, 04, 14010
2022, 05, 28402
2022, 06, 9992
2022, 07, 7292
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `transaction_year`
- `transaction_week`
- `first_day_of_week`
- `week_day`

Results table

transaction_year	transaction_week	first_day_of_week	week_day
2022	01	01/03/2022	Mon
2022	02	01/10/2022	Mon
2022	03	01/17/2022	Mon
2022	04	01/24/2022	Mon

transaction_year	transaction_week	first_day_of_week	week_day
2022	05	01/31/2022	Mon
2022	06	02/07/2022	Mon
2022	07	02/14/2022	Mon

The `first_day_of_week` field is created in the preceding load statement by using the `makeweekdate()` function. The `transaction_year` and `transaction_week` parameters are passed as function arguments, and the day parameter is left blank.

The function then combines and converts these values into a date field, returning the results in the format of the `DateFormat` system variable.

The `makeweekdate()` function and its arguments are also wrapped in a `weekday()` function to return the `week_day` field. As can be seen in the table above, the `week_day` field returns Monday in all cases since that parameter was left blank in the `makeweekdate()` function, which defaults to `0` (first day of the week), and first day of the week is set to Monday by the `FirstWeekDay` system variable.

Example 3 – Chart object example

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing weekly sales totals for 2022 in a table called `Sales`.
- Transaction dates provided across three fields: `year`, `week`, and `sales`.

In this example, a chart object will be used to create a measure equivalent to the `end_of_week` calculation from the first example. This measure will use the `makeweekdate()` function to return the date for the Friday of that week in the format MM/DD/YYYY.

To prove that the date returned is a Friday, a second measure is created to return the day of the week.

Load script

```
SET DateFormat='MM/DD/YYYY';
SET FirstWeekDay=0;
SET BrokenWeeks=0;
SET ReferenceDay=4;

Master_Calendar:
Load * Inline [
transaction_year, transaction_week, sales
2022, 01, 10000
2022, 02, 11250
2022, 03, 9830
```

```
2022, 04, 14010
2022, 05, 28402
2022, 06, 9992
2022, 07, 7292
];
```

Results

Do the following:

1. Load the data and open a sheet. Create a new table and add these fields as dimensions:
   - `transaction_year`
   - `transaction_week`
2. To perform the calculation equivalent to that of the `end_of_week`field from the first example, create the following measure:
   `=makeweekdate(transaction_year,transaction_week,4)`
3. To calculate the day of the week for each transaction, create the following measure:
   `=weekday(makeweekdate(transaction_year,transaction_week,4))`

Results table

transaction_ year	transaction_ week	=makeweekdate (transaction_ year,transaction_week,4)	=weekday(makeweekdate (transaction_year,transaction_ week,4))
2022	01	01/07/2022	Fri
2022	02	01/14/2022	Fri
2022	03	01/21/2022	Fri
2022	04	01/28/2022	Fri
2022	05	02/04/2022	Fri
2022	06	02/11/2022	Fri
2022	07	02/18/2022	Fri

An equivalent field to `end_of_week` is created in the chart object as a measure by using the `makeweekdate()` function. The `transaction_year` and `transaction_week` fields are passed as year and week arguments. A value of 4 is used for the day argument.

The function then combines and converts these values into a date field, returning the results in the format of the `DateFormat` system variable.

The `makeweekdate()` function and its arguments are also wrapped in a `weekday()` function to return a calculation equivalent to that of the `week_day` field from the first example. As can be seen in the table above, the last column on the right shows that these dates do occur on a Friday.

Example 4 – Scenario

Load script and chart expression

Overview

In this example, create a list of dates containing all the Fridays for the year 2022.

Open the Data load editor and add the load script below to a new tab.

Load script

```
SET DateFormat='MM/DD/YYYY';
SET FirstWeekDay=0;
SET BrokenWeeks=0;
SET ReferenceDay=4;

Calendar:
    load
           *,
        weekday(date) as weekday
    where year(date)=2022;
load
    makeweekdate(2022,recno()-2,4) as date
AutoGenerate 60;
```

Results

Results table

date	weekday
01/07/2022	Fri
01/14/2022	Fri
01/21/2022	Fri
01/28/2022	Fri
02/04/2022	Fri
02/11/2022	Fri
02/18/2022	Fri
02/25/2022	Fri
03/04/2022	Fri
03/11/2022	Fri
03/18/2022	Fri
03/25/2022	Fri

date	weekday
04/01/2022	Fri
04/08/2022	Fri
04/15/2022	Fri
04/22/2022	Fri
04/29/2022	Fri
05/06/2022	Fri
05/13/2022	Fri
05/20/2022	Fri
05/27/2022	Fri
06/03/2022	Fri
06/10/2022	Fri
06/17/2022	Fri
+ 27 more rows	

The `makeweekdate()` function finds each Friday in 2022. Using a week parameter of -2 ensures that no dates are missed. Finally, a preceding load creates an additional `weekday` field for clarity, to show that each `date` value is a Friday.

minute

This function returns an integer representing the minute when the fraction of the **expression** is interpreted as a time according to the standard number interpretation.

Syntax:

```
minute(expression)
```

Return data type: integer

Examples and results:

Scripting examples

Example	Result
`minute ( '09:14:36' )`	returns 14
`minute ( '0.5555' )`	returns 19 (Because 0.5555 = 13:19:55)

month

This function returns a dual value: a month name as defined in the environment variable **MonthNames** and an integer between 1-12. The month is calculated from the date interpretation of the expression, according to the standard number interpretation.

The function returns the name of the month in the format of the `MonthName` system variable for a particular date. It is commonly used to create a day field as a dimension in a Master Calendar.

Syntax:

```
month(expression)
```

Return data type: integer

Function examples

Example	Result
`month( 2012-10-12 )`	returns Oct
`month( 35648 )`	returns Aug, because 35648 = 1997-08-06

Example 1 – DateFormat dataset (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates named `Master_Calendar`. The `DateFormat` system variable is set to DD/MM/YYYY.
- A preceding load that creates an additional field, named`month_name`, using the `month()` function.
- An additional field, named`long_date`, using the `date()` function to express the full date.

Load script

```
SET DateFormat='DD/MM/YYYY';

Master_Calendar:
Load
        date,
        date(date,'dd-MMMM-YYYY') as long_date,
        month(date) as month_name
Inline
[
date
03/01/2022
03/02/2022
```

```
03/03/2022
03/04/2022
03/05/2022
03/06/2022
03/07/2022
03/08/2022
03/09/2022
03/10/2022
03/11/2022
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `long_date`
- `month_name`

Results table

date	long_date	month_name
03/01/2022	03-January- 2022	Jan
03/02/2022	03-February- 2022	Feb
03/03/2022	03-March- 2022	Mar
03/04/2022	03-April- 2022	Apr
03/05/2022	03-May- 2022	May
03/06/2022	03-June- 2022	Jun
03/07/2022	03-July- 2022	Jul
03/08/2022	03-August- 2022	Aug
03/09/2022	03-September- 2022	Sep
03/10/2022	03-October- 2022	Oct
03/11/2022	03-November- 2022	Nov

The month name is correctly evaluated by the `month()` function in the script.

Example 2 – ANSI dates (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates named `Master_Calendar`. The DateFormat system variable DD/MM/YYYY is used. However, the dates that are included in the dataset are in ANSI standard date format.
- A preceding load that creates an additional field, named`month_name`, using the month() function.
- An additional field, named`long_date`, using the date() function to express the full date.

Load script

```
SET DateFormat='DD/MM/YYYY';
Master_Calendar:
Load
       date,
       date(date,'dd-MMMM-YYYY') as long_date,
       month(date) as month_name

Inline
[
date
2022-01-11
2022-02-12
2022-03-13
2022-04-14
2022-05-15
2022-06-16
2022-07-17
2022-08-18
2022-09-19
2022-10-20
2022-11-21
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `long_date`
- `month_name`

Results table

date	long_date	month_name
03/11/2022	11-March- 2022	11
03/12/2022	12-March- 2022	12
03/13/2022	13-March- 2022	13
03/14/2022	14-March- 2022	14
03/15/2022	15-March- 2022	15
03/16/2022	16-March- 2022	16

date	long_date	month_name
03/17/2022	17-March- 2022	17
03/18/2022	18-March- 2022	18
03/19/2022	19-March- 2022	19
03/20/2022	20-March- 2022	20
03/21/2022	21-March- 2022	21

The month name is correctly evaluated by the `month()` function in the script.

Example 3 – Unformatted dates (script)

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of dates named `Master_Calendar`. The `DateFormat` system variable DD/MM/YYYY is used.
- A preceding load that creates an additional field, named `month_name`, using the `month()` function.
- The original unformatted date, named `unformatted_date`.
- An additional field, named `long_date`, using the `date()` function to express the full date.

Load script

```
SET DateFormat='DD/MM/YYYY';

Master_Calendar:
Load
      unformatted_date,
      date(unformatted_date,'dd-MMMM-YYYY') as long_date,
      month(unformatted_date) as month_name

Inline
[
unformatted_date
44868
44898
44928
44958
44988
45018
45048
45078
45008
45038
45068
```

```
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `unformatted_date`
- `long_date`
- `month_name`

Results table

unformatted_date	long_date	month_name
44868	03-January- 2022	Jan
44898	03-February- 2022	Feb
44928	03-March- 2022	Mar
44958	03-April- 2022	Apr
44988	03-May- 2022	May
45018	03-June- 2022	Jun
45048	03-July- 2022	Jul
45078	03-August- 2022	Aug
45008	03-September- 2022	Sep
45038	03-October- 2022	Oct
45068	03-November- 2022	Nov

The month name is correctly evaluated by the `month()` function in the script.

Example 4 – Calculating expiry month

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset of orders placed in March named `Subscriptions`. The table contains three fields:
  - id
  - order_date
  - amount

Load script

```
Subscriptions:
Load
        id,
        order_date,
        amount
Inline
[
id,order_date,amount
1,03/01/2022,231.24
2,03/02/2022,567.28
3,03/03/2022,364.28
4,03/04/2022,575.76
5,03/05/2022,638.68
6,03/06/2022,785.38
7,03/07/2022,967.46
8,03/08/2022,287.67
9,03/09/2022,764.45
10,03/10/2022,875.43
11,03/11/2022,957.35
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension: `order_date`.

To calculate the month an order will expire, create this measure: `=month(order_date+180)`.

Results table

order_date	=month(order_date+180)
03/01/2022	Jul
03/02/2022	Aug
03/03/2022	Aug
03/04/2022	Sep
03/05/2022	Oct
03/06/2022	Nov
03/07/2022	Dec
03/08/2022	Jan
03/09/2022	Mar
03/10/2022	Apr
03/11/2022	May

The `month()` function correctly determines that an order placed on the 11th of March would expire in July.

monthend

This function returns a value corresponding to a timestamp of the last millisecond of the last day of the month containing `date`. The default output format will be the `DateFormat` set in the script.

Syntax:

```
MonthEnd(date[, period_no])
```

Return data type: dual

Arguments:

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`monthend('19/02/2012')`	Returns 29/02/2012 23:59:59.
`monthend('19/02/2001', -1)`	Returns 31/01/2001 23:59:59.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example finds the last day in the month of each invoice date in the table, where the base date is shifted by four months by specifying *period_no* as 4.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
MonthEnd(InvDate, 4) AS MthEnd
```

```
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the monthend() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	MthEnd
28/03/2012	31/07/2012
10/12/2012	30/04/2013
5/2/2013	30/06/2013
31/3/2013	31/07/2013
19/5/2013	30/09/2013
15/9/2013	31/01//2014
11/12/2013	30/04//2014
2/3/2014	31/07//2014
14/5/2014	30/09/2014
13/6/2014	31/10/2014
7/7/2014	30/11/2014
4/8/2014	31/12/2014

monthname

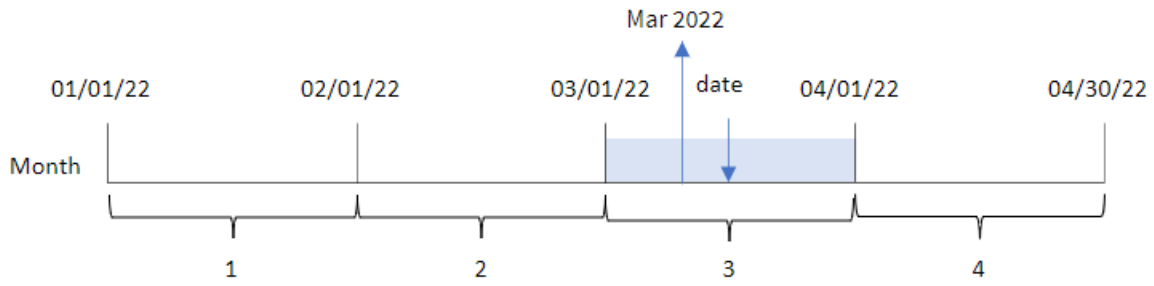
This function returns a display value showing the month (formatted according to the **MonthNames** script variable) and year with an underlying numeric value corresponding to a timestamp of the first millisecond of the first day of the month.

Syntax:

```
MonthName(date[, period_no])
```

Return data type: dual

Diagram of monthname function



Arguments

Argument	Description
date	The date to evaluate.
period_no	**period_no** is an integer, which, if 0 or omitted, indicates the month that contains **date**. Negative values in **period_no** indicate preceding months and positive values indicate succeeding months.

Function examples

Example	Result
`monthname('10/19/2013')`	Returns Oct 2013
`monthname('10/19/2013', -1)`	Returns Sep 2013

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 – Basic example

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions for 2022, which is loaded into a table called `Transactions`.
- The date field provided in the `DateFormat` system variable (MM/DD/YYYY) format.
- The creation of a field, `transaction_month`, that returns the month in which the transactions took place.

Load script

```
SET DateFormat='MM/DD/YYYY';
SET MonthNames='Jan;Feb;Mar;Apr;May;Jun;Jul;Aug;Sep;Oct;Nov;Dec';

Transactions:
    Load
        *,
        monthname(date) as transaction_month
        ;
Load
*
Inline
[
id,date,amount
8188,1/7/2022,17.17
8189,1/19/2022,37.23
8190,2/28/2022,88.27
8191,2/5/2022,57.42
8192,3/16/2022,53.80
8193,4/1/2022,82.06
8194,5/7/2022,40.39
8195,5/16/2022,87.21
8196,6/15/2022,95.93
8197,6/26/2022,45.89
8198,7/9/2022,36.23
8199,7/22/2022,25.66
8200,7/23/2022,82.77
8201,7/27/2022,69.98
8202,8/2/2022,76.11
8203,8/8/2022,25.12
8204,8/19/2022,46.23
8205,9/26/2022,84.21
8206,10/14/2022,96.24
8207,10/29/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

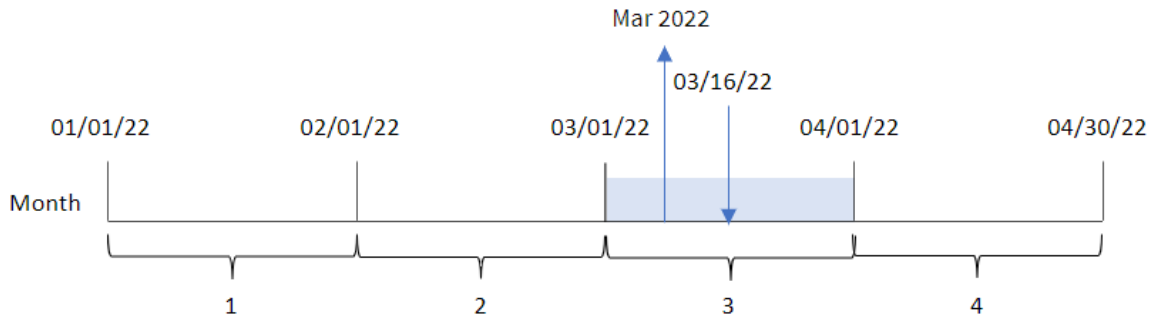
- `date`
- `transaction_month`

Results table

date	transaction_month
1/7/2022	Jan 2022
1/19/2022	Jan 2022
2/5/2022	Feb 2022
2/28/2022	Feb 2022
3/16/2022	Mar 2022
4/1/2022	Apr 2022
5/7/2022	May 2022
5/16/2022	May 2022
6/15/2022	Jun 2022
6/26/2022	Jun 2022
7/9/2022	Jul 2022
7/22/2022	Jul 2022
7/23/2022	Jul 2022
7/27/2022	Jul 2022
8/2/2022	Aug 2022
8/8/2022	Aug 2022
8/19/2022	Aug 2022
9/26/2022	Sep 2022
10/14/2022	Oct 2022
10/29/2022	Oct 2022

The `transaction_month` field is created in the preceding load statement by using the `monthname()` function and passing the `date` field as the function's argument.

Diagram of monthname function, basic example



The `monthname()` function identifies that transaction 8192 took place in March 2022, and returns this value using the `MonthNames` system variable.

Example 2 – period_no

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same inline dataset and scenario as the first example.
- The creation of a field, `transaction_previous_month`, that returns the timestamp for the end of the month before the transaction took place.

Load script

```
SET DateFormat='MM/DD/YYYY';
SET MonthNames='Jan;Feb;Mar;Apr;May;Jun;Jul;Aug;Sep;Oct;Nov;Dec';

Transactions:
    Load
        *,
        monthname(date,-1) as transaction_previous_month
        ;
Load
*
Inline
[
id,date,amount
8188,1/7/2022,17.17
8189,1/19/2022,37.23
8190,2/28/2022,88.27
8191,2/5/2022,57.42
8192,3/16/2022,53.80
8193,4/1/2022,82.06
8194,5/7/2022,40.39
```

```
8195,5/16/2022,87.21
8196,6/15/2022,95.93
8197,6/26/2022,45.89
8198,7/9/2022,36.23
8199,7/22/2022,25.66
8200,7/23/2022,82.77
8201,7/27/2022,69.98
8202,8/2/2022,76.11
8203,8/8/2022,25.12
8204,8/19/2022,46.23
8205,9/26/2022,84.21
8206,10/14/2022,96.24
8207,10/29/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `transaction_previous_month`

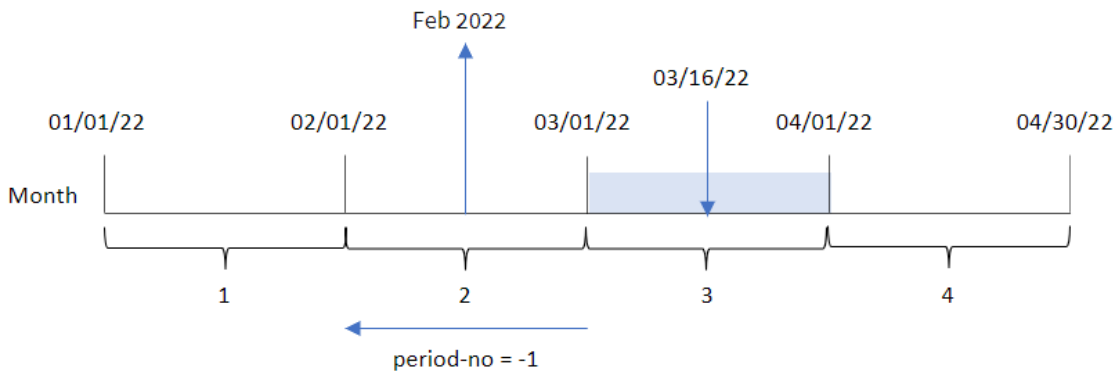
Results table

date	transaction_previous_month
1/7/2022	Dec 2021
1/19/2022	Dec 2021
2/5/2022	Jan 2022
2/28/2022	Jan 2022
3/16/2022	Feb 2022
4/1/2022	Mar 2022
5/7/2022	Apr 2022
5/16/2022	Apr 2022
6/15/2022	May 2022
6/26/2022	May 2022
7/9/2022	Jun 2022
7/22/2022	Jun 2022
7/23/2022	Jun 2022
7/27/2022	Jun 2022
8/2/2022	Jul 2022
8/8/2022	Jul 2022
8/19/2022	Jul 2022

date	transaction_previous_month
9/26/2022	Aug 2022
10/14/2022	Sep 2022
10/29/2022	Sep 2022

In this instance, because a `period_no` of -1 was used as the offset argument in the `monthname()` function, the function first identifies the month that the transactions take place in. It then shifts to one month prior and returns the month name and year.

Diagram of monthname function, period_no example



Transaction 8192 took place on March 16. The `monthname()` function identifies that the month before the transaction took place was February and returns the month, in the `MonthNames` system variable format, along with the year 2022.

Example 3 – Chart object example

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains the same inline dataset and scenario as the first example. However, in this example, the unchanged dataset is loaded into the application. The calculation that returns a timestamp for the end of the month when the transactions took place is created as a measure in a chart object of the application.

Load script

```
SET DateFormat='MM/DD/YYYY';
SET MonthNames='Jan;Feb;Mar;Apr;May;Jun;Jul;Aug;Sep;Oct;Nov;Dec';

Transactions:
Load
*
```

```
Inline
[
id,date,amount
8188,1/7/2022,17.17
8189,1/19/2022,37.23
8190,2/28/2022,88.27
8191,2/5/2022,57.42
8192,3/16/2022,53.80
8193,4/1/2022,82.06
8194,5/7/2022,40.39
8195,5/16/2022,87.21
8196,6/15/2022,95.93
8197,6/26/2022,45.89
8198,7/9/2022,36.23
8199,7/22/2022,25.66
8200,7/23/2022,82.77
8201,7/27/2022,69.98
8202,8/2/2022,76.11
8203,8/8/2022,25.12
8204,8/19/2022,46.23
8205,9/26/2022,84.21
8206,10/14/2022,96.24
8207,10/29/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension:`date`.

Create the following measure:

```
=monthname(date)
```

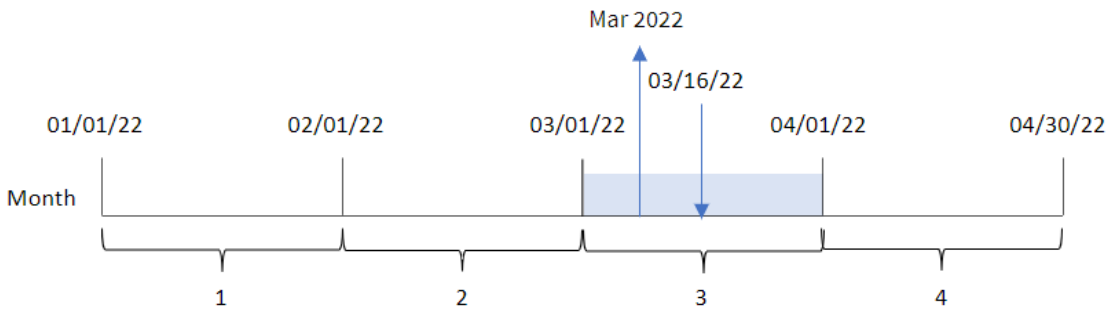
Results table

date	=monthname(date)
1/7/2022	Jan 2022
1/19/2022	Jan 2022
2/5/2022	Feb 2022
2/28/2022	Feb 2022
3/16/2022	Mar 2022
4/1/2022	Apr 2022
5/7/2022	May 2022
5/16/2022	May 2022
6/15/2022	Jun 2022
6/26/2022	Jun 2022

date	=monthname(date)
7/9/2022	Jul 2022
7/22/2022	Jul 2022
7/23/2022	Jul 2022
7/27/2022	Jul 2022
8/2/2022	Aug 2022
8/8/2022	Aug 2022
8/19/2022	Aug 2022
9/26/2022	Sep 2022
10/14/2022	Oct 2022
10/29/2022	Oct 2022

The `month_name` measure is created in the chart object by using the `monthname()` function and passing the `date` field as the function's argument.

Diagram of monthname function, chart object example



The `monthname()` function identifies that transaction 8192 took place in March 2022, and returns this value using the `MonthNames` system variable.

monthsend

This function returns a value corresponding to a timestamp of the last millisecond of the month, bi-month, quarter, four-month period, or half-year containing a base date. It is also possible to find the timestamp for a previous or following time period.

Syntax:

```
MonthsEnd(n_months, date[, period_no [, first_month_of_year]])
```

Return data type: dual

Arguments:

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`monthsend(4, '19/07/2013')`	Returns 31/08/2013.
`monthsend(4, '19/10/2013', -1)`	Returns 31/08/2013.
`monthsend(4, '19/10/2013', 0, 2)`	Returns 31/01/2014. Because the start of the year becomes month 2.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example finds the end of the final day of bi-month period for each invoice date, shifted forwards by one bi-month period.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
MonthsEnd(2, InvDate, 1) AS BiMthsEnd
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the MonthsEnd() function.

Results table

InvDate	BiMthsEnd
28/03/2012	30/06/2012
10/12/2012	28/02/2013
5/2/2013	30/04/2013
31/3/2013	30/04/2013
19/5/2013	31/08/2013
15/9/2013	31/12/2013
11/12/2013	28/02/2014
2/3/2014	30/06/2014
14/5/2014	31/08/2014
13/6/2014	31/08/2014
7/7/2014	31/10/2014
4/8/2014	31/10/2014

monthsname

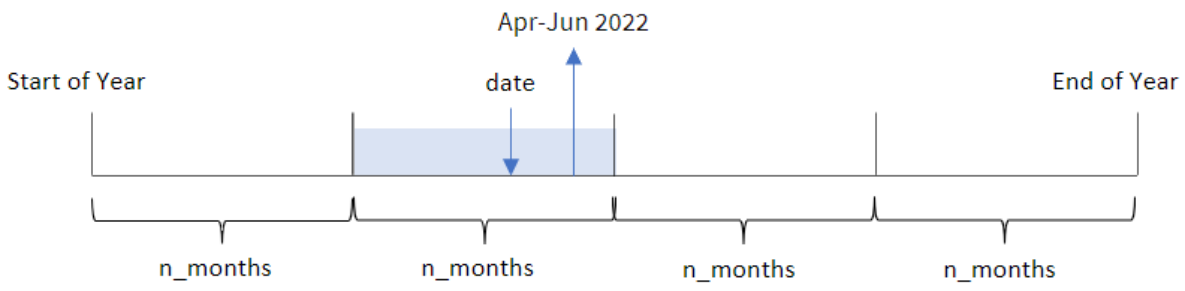
This function returns a display value representing the range of the months of the period (formatted according to the **MonthNames** script variable) as well as the year. The underlying numeric value corresponds to a timestamp of the first millisecond of the month, bi-month, quarter, four-month period, or half-year containing a base date.

Syntax:

```
MonthsName(n_months, date[, period_no[, first_month_of_year]])
```

Return data type: dual

Diagram of monthsname function



The `monthsname()` function divides the year into segments based on the `n_months` agrument provided. It then evaluates the segment to which each provided `date` belongs, and returns the start and end month names of that segment, as well as the year. The function also provides the ability to return these boundaries from preceding or following segments, as well as redefining which is the first month of the year.

The following segments of the year are available in the function as `n_month` arguments:

Possible `n_month` arguments

Periods	Number of months
month	1
bi-month	2
quarter	3
four months	4
half-year	6

Arguments

Argument	Description
n_months	The number of months that defines the period. An integer or expression that resolves to an integer that must be one of: 1 (equivalent to the inmonth() function), 2 (bi-month), 3 (equivalent to the inquarter() function), 4 (four-month period), or 6 (half year).
date	The date to evaluate.
period_no	The period can be offset by **period_no**, an integer, or expression resolving to an integer, where the value 0 indicates the period that contains **base_date**. Negative values in **period_no** indicate preceding periods and positive values indicate succeeding periods.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

When to use it

The `monthsname()` function is useful when you would like to provide the user with the functionality to compare aggregations by a period of their choosing. For example, you could provide an input variable to let the user see the total sales of products by month, quarter, or half-year.

These dimensions can be created either in the load script by adding the function as a field in a Master Calendar table, or alternatively, by creating the dimension directly in a chart as a calculated dimension.

Function examples

Example	Result
`monthsname(4, '10/19/2013')`	Returns 'Sep-Dec 2013.' In this and the other examples, the **SET Monthnames** statement is set to Jan;Feb;Mar, and so on.

Example	Result
`monthsname(4, '10/19/2013', -1)`	Returns 'May-Aug 2013'.
`monthsname(4, '10/19/2013', 0, 2)`	Returns 'Oct-Jan 2014', since the year is specified to begin in month 2. Therefore, the four-month period ends on the first month of the following year.

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Example 1 – Basic example

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions for 2022, which is loaded into a table called `Transactions`.
- The date field provided in the `DateFormat` system variable (MM/DD/YYYY) format.
- The creation of a field, `bi_monthly_range`, that groups transactions into bi-monthly segments and returns the boundary names of that segment for each transaction.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
        *,
        monthsname(2,date) as bi_monthly_range
        ;
Load
*
Inline
[
```

```
id,date,amount
8188,2/19/2022,37.23
8189,3/7/2022,17.17
8190,3/30/2022,88.27
8191,4/5/2022,57.42
8192,4/16/2022,53.80
8193,5/1/2022,82.06
8194,5/7/2022,40.39
8195,5/22/2022,87.21
8196,6/15/2022,95.93
8197,6/26/2022,45.89
8198,7/9/2022,36.23
8199,7/22/2022,25.66
8200,7/23/2022,82.77
8201,7/27/2022,69.98
8202,8/2/2022,76.11
8203,8/8/2022,25.12
8204,8/19/2022,46.23
8205,9/26/2022,84.21
8206,10/14/2022,96.24
8207,10/29/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `bi_monthly_range`

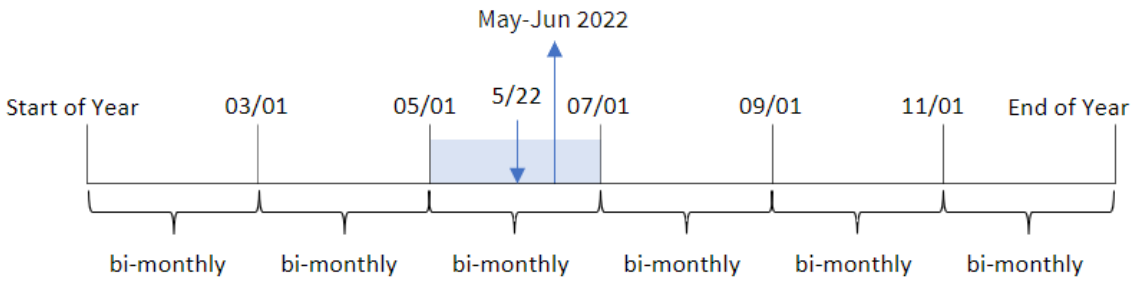
Results table

date	bi_monthly_range
2/19/2022	Jan-Feb 2022
3/7/2022	Mar-Apr 2022
3/30/2022	Mar-Apr 2022
4/5/2022	Mar-Apr 2022
4/16/2022	Mar-Apr 2022
5/1/2022	May-Jun 2022
5/7/2022	May-Jun 2022
5/22/2022	May-Jun 2022
6/15/2022	May-Jun 2022
6/26/2022	May-Jun 2022
7/9/2022	Jul-Aug 2022

date	bi_monthly_range
7/22/2022	Jul-Aug 2022
7/23/2022	Jul-Aug 2022
7/27/2022	Jul-Aug 2022
8/2/2022	Jul-Aug 2022
8/8/2022	Jul-Aug 2022
8/19/2022	Jul-Aug 2022
9/26/2022	Sep-Oct 2022
10/14/2022	Sep-Oct 2022
10/29/2022	Sep-Oct 2022

The `bi_monthly_range` field is created in the preceding load statement by using the `monthsname()` function. The first argument provided is 2, dividing the year into bi-monthly segments. The second argument identifies which field is being evaluated.

Diagram of monthsname function, basic example



Transaction 8195 takes place on May 22. The `monthsname()` function initially divides the year into bi-monthly segments. Transaction 8195 falls into the segment between May 1 and June 30. Therefore, the function returns these months in the `MonthNames` system variable format, as well as the year, May-Jun 2022.

Example 2 – period_no

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same inline dataset and scenario as the first example.
- The creation of a field, `prev_bi_monthly_range`, that groups transactions into bi-monthly segments and returns the previous segment boundary names for each transaction.

Add your other text here, as needed, with lists etc.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
        *,
        MonthsName(2,date,-1) as prev_bi_monthly_range
        ;
Load
*
Inline
[
id,date,amount
8188,2/19/2022,37.23
8189,3/7/2022,17.17
8190,3/30/2022,88.27
8191,4/5/2022,57.42
8192,4/16/2022,53.80
8193,5/1/2022,82.06
8194,5/7/2022,40.39
8195,5/22/2022,87.21
8196,6/15/2022,95.93
8197,6/26/2022,45.89
8198,7/9/2022,36.23
8199,7/22/2022,25.66
8200,7/23/2022,82.77
8201,7/27/2022,69.98
8202,8/2/2022,76.11
8203,8/8/2022,25.12
8204,8/19/2022,46.23
8205,9/26/2022,84.21
8206,10/14/2022,96.24
8207,10/29/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

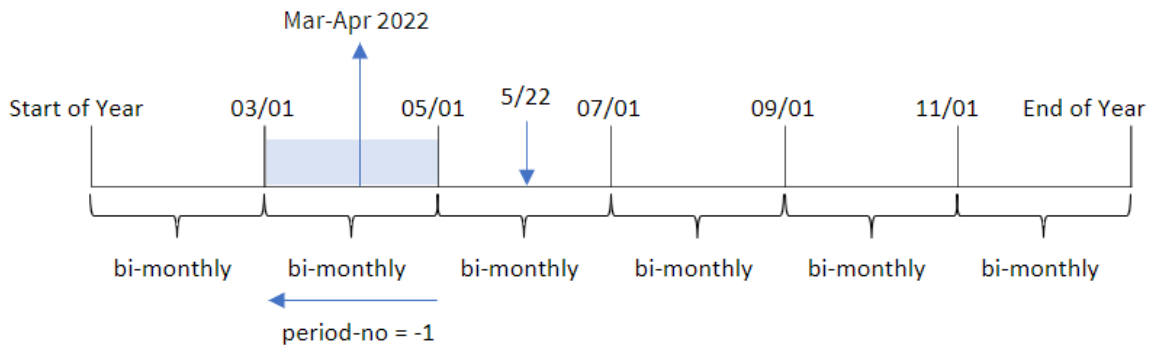
- `date`
- `prev_bi_monthly_range`

Results table

date	prev_bi_monthly_range
2/19/2022	Nov-Dec 2021
3/7/2022	Jan-Feb 2022
3/30/2022	Jan-Feb 2022
4/5/2022	Jan-Feb 2022
4/16/2022	Jan-Feb 2022
5/1/2022	Mar-Apr 2022
5/7/2022	Mar-Apr 2022
5/22/2022	Mar-Apr 2022
6/15/2022	Mar-Apr 2022
6/26/2022	Mar-Apr 2022
7/9/2022	May-Jun 2022
7/22/2022	May-Jun 2022
7/23/2022	May-Jun 2022
7/27/2022	May-Jun 2022
8/2/2022	May-Jun 2022
8/8/2022	May-Jun 2022
8/19/2022	May-Jun 2022
9/26/2022	Jul-Aug 2022
10/14/2022	Jul-Aug 2022
10/29/2022	Jul-Aug 2022

In this example, `-1` is used as the `period_no` argument in the `monthsname()` function. After initially dividing a year into bi-monthly segments, the function then returns the previous segment boundaries for when a transaction takes place.

Diagram of monthsname function, period_no example



Transaction 8195 occurs in the segment between May and June. Therefore, the previous bi-monthly segment was between March 1 and April 30, and so the function returns Mar-Apr 2022.

Example 3 – first_month_of_year

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same inline dataset and scenario as the first example.
- The creation of a different field, `bi_monthly_range`, that groups transactions into bi-monthly segments and returns the segment boundaries for each transaction.

However, in this example, we also need to set April as the first month of the financial year.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
        *,
        MonthsName(2,date,0,4) as bi_monthly_range
        ;
Load
*
Inline
[
id,date,amount
8188,2/19/2022,37.23
8189,3/7/2022,17.17
8190,3/30/2022,88.27
8191,4/5/2022,57.42
```

```
8192,4/16/2022,53.80
8193,5/1/2022,82.06
8194,5/7/2022,40.39
8195,5/22/2022,87.21
8196,6/15/2022,95.93
8197,6/26/2022,45.89
8198,7/9/2022,36.23
8199,7/22/2022,25.66
8200,7/23/2022,82.77
8201,7/27/2022,69.98
8202,8/2/2022,76.11
8203,8/8/2022,25.12
8204,8/19/2022,46.23
8205,9/26/2022,84.21
8206,10/14/2022,96.24
8207,10/29/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `bi_monthly_range`

Results table

date	bi_monthly_range
2/19/2022	Feb-Mar 2021
3/7/2022	Feb-Mar 2021
3/30/2022	Feb-Mar 2021
4/5/2022	Apr-May 2022
4/16/2022	Apr-May 2022
5/1/2022	Apr-May 2022
5/7/2022	Apr-May 2022
5/22/2022	Apr-May 2022
6/15/2022	Jun-Jul 2022
6/26/2022	Jun-Jul 2022
7/9/2022	Jun-Jul 2022
7/22/2022	Jun-Jul 2022
7/23/2022	Jun-Jul 2022
7/27/2022	Jun-Jul 2022

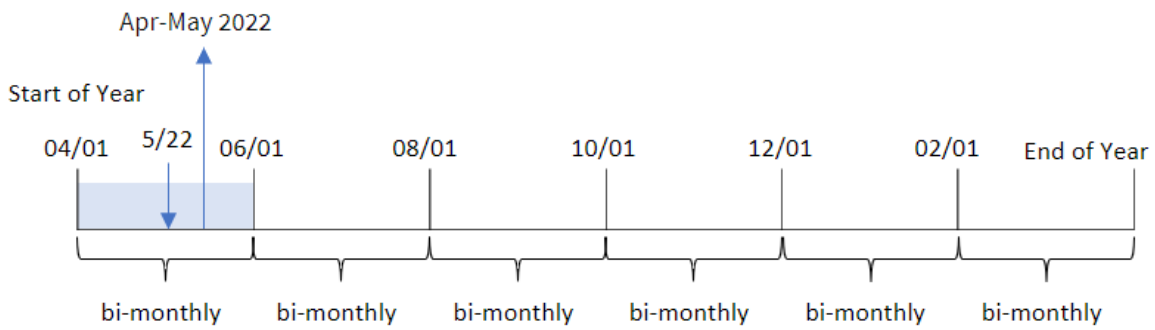
date	bi_monthly_range
8/2/2022	Aug-Sep 2022
8/8/2022	Aug-Sep 2022
8/19/2022	Aug-Sep 2022
9/26/2022	Aug-Sep 2022
10/14/2022	Oct-Nov 2022
10/29/2022	Oct-Nov 2022

By using 4 as the `first_month_of_year` argument in the `monthsname()` function, the function begins the year on April 1. It then divides the year into bi-monthly segments: Apr-May,Jun-Jul,Aug-Sep,Oct-Nov,Dec-Jan,Feb-Mar.

Paragraph text for Results.

Transaction 8195 took place on May 22 and falls into the segment between April 1 and May 31. Therefore, the function returns Apr-May 2022.

Diagram of monthsname function, first_month_of_year example



Example 4 – Chart object example

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains the same inline dataset and scenario as the first example. However, in this example, the unchanged dataset is loaded into the application. The calculation that groups transactions into bi-monthly segments and returns the segment boundaries for each transaction is created as a measure in a chart object of the application.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
Load
*
Inline
[
id,date,amount
8188,2/19/2022,37.23
8189,3/7/2022,17.17
8190,3/30/2022,88.27
8191,4/5/2022,57.42
8192,4/16/2022,53.80
8193,5/1/2022,82.06
8194,5/7/2022,40.39
8195,5/22/2022,87.21
8196,6/15/2022,95.93
8197,6/26/2022,45.89
8198,7/9/2022,36.23
8199,7/22/2022,25.66
8200,7/23/2022,82.77
8201,7/27/2022,69.98
8202,8/2/2022,76.11
8203,8/8/2022,25.12
8204,8/19/2022,46.23
8205,9/26/2022,84.21
8206,10/14/2022,96.24
8207,10/29/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension:`date`.

Create the following measure:

```
=monthsname(2,date)
```

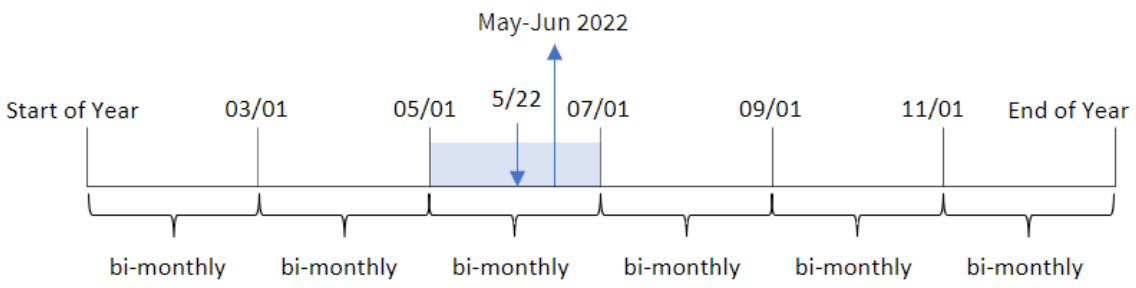
Results table

date	=monthsname(2,date)
2/19/2022	Jan-Feb 2022
3/7/2022	Mar-Apr 2022
3/30/2022	Mar-Apr 2022
4/5/2022	Mar-Apr 2022
4/16/2022	Mar-Apr 2022
5/1/2022	May-Jun 2022

date	=monthsname(2,date)
5/7/2022	May-Jun 2022
5/22/2022	May-Jun 2022
6/15/2022	May-Jun 2022
6/26/2022	May-Jun 2022
7/9/2022	Jul-Aug 2022
7/22/2022	Jul-Aug 2022
7/23/2022	Jul-Aug 2022
7/27/2022	Jul-Aug 2022
8/2/2022	Jul-Aug 2022
8/8/2022	Jul-Aug 2022
8/19/2022	Jul-Aug 2022
9/26/2022	Sep-Oct 2022
10/14/2022	Sep-Oct 2022
10/29/2022	Sep-Oct 2022

The `bi_monthly_range` field is created as a measure in the chart object by using the `monthsname()` function. The first argument provided is 2, dividing the year into bi-monthly segments. The second argument identifies which field is being evaluated.

Diagram of monthsname function, chart object example



Transaction 8195 takes place on May 22. The `monthsname()` function initially divides the year into bi-monthly segments. Transaction 8195 falls into the segment between May 1 and June 30. Therefore, the function returns these months in the `MonthNames` system variable format, as well as the year, May-Jun 2022.

Example 5 – Scenario

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing transactions for 2022, which is loaded into a table called `Transactions`.
- The date field provided in the `DateFormat` system variable (MM/DD/YYYY) format.

The end user would like a chart object that displays total sales by a period of their own choosing. This could be achieved even when this dimension is not available in the data model, using the `monthsname()` function as a calculated dimension that is dynamically modified by a variable input control.

Load script

```
SET vPeriod = 1;
SET DateFormat='MM/DD/YYYY';

Transactions:
Load
*
Inline
[
id,date,amount
8188,'1/7/2022',17.17
8189,'1/19/2022',37.23
8190,'2/28/2022',88.27
8191,'2/5/2022',57.42
8192,'3/16/2022',53.80
8193,'4/1/2022',82.06
8194,'5/7/2022',40.39
8195,'5/16/2022',87.21
8196,'6/15/2022',95.93
8197,'6/26/2022',45.89
8198,'7/9/2022',36.23
8199,'7/22/2022',25.66
8200,'7/23/2022',82.77
8201,'7/27/2022',69.98
8202,'8/2/2022',76.11
8203,'8/8/2022',25.12
8204,'8/19/2022',46.23
8205,'9/26/2022',84.21
8206,'10/14/2022',96.24
8207,'10/29/2022',67.67
];
```

Results

Load the data and open a sheet.

At the start of the load script, a variable (`vPeriod`) has been created that will be tied to the variable input control. Next, configure the variable as a custom object in the sheet.

Do the following:

1. In the assets panel, click **Custom objects**.
2. Select **Qlik Dashboard bundle**, and create a **Variable input** object.
3. Enter a title for the chart object.
4. Under **Variable**, select **vPeriod** as the Name and set the object to show as a **Drop down**.
5. Under **Values**, configure the object to use dynamic values. Enter the following:
   `='1~month|2~bi-month|3~quarter|4~tertial|6~half-year'`

Next, create the results table.

Do the following:

1. Create a new table and add the following calculated dimension:
   `=monthsname($(vPeriod),date)`
2. Add this measure to calculate the total sales:
   `=sum(amount)`
3. Set the measure's **Number formatting** to **Money**. Click ✔ **Done editing**. You can now modify the data shown in the table by adjusting the time segment in the variable object.

This is what the results table will look like when the `tertial` option is selected:

Results table

monthsname($(vPeriod),date)	=sum(amount)
Jan-Apr 2022	253.89
May-Aug 2022	713.58
Sep-Dec 2022	248.12

monthsstart

This function returns a value corresponding to the timestamp of the first millisecond of the month, bi-month, quarter, four-month period, or half-year containing a base date. It is also possible to find the timestamp for a previous or following time period.

Syntax:

```
MonthsStart(n_months, date[, period_no [, first_month_of_year]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
n_months	The number of months that defines the period. An integer or expression that resolves to an integer that must be one of: 1 (equivalent to the inmonth() function), 2 (bi-month), 3 (equivalent to the inquarter() function), 4 (four-month period), or 6 (half year).
date	The date to evaluate.
period_no	The period can be offset by **period_no**, an integer, or expression resolving to an integer, where the value 0 indicates the period that contains **base_date**. Negative values in **period_no** indicate preceding periods and positive values indicate succeeding periods.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`monthsstart(4, '19/10/2013')`	Returns 1/09/2013.
`monthsstart(4, '19/10/2013, -1)`	Returns 01/05/2013.
`monthsstart(4, '19/10/2013', 0, 2 )`	Returns 01/10/2013. Because the start of the year becomes month 2.

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example finds the first day of the bi-month period for each invoice date, shifted forwards by one bi-month period.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
```

```
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
MonthsStart(2, InvDate, 1) AS BiMthsStart
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the MonthsStart() function.

Results table

InvDate	BiMthsStart
28/03/2012	01/05/2012
10/12/2012	01/01/2013
5/2/2013	01/03/2013
31/3/2013	01/05/2013
19/5/2013	01/07/2013
15/9/2013	01/11/2013
11/12/2013	01/01/2014
2/3/2014	01/05/2014
14/5/2014	01/07/2014
13/6/2014	01/07/2014
7/7/2014	01/09/2014
4/8/2014	01/09/2014

monthstart

This function returns a value corresponding to a timestamp of the first millisecond of the first day of the month containing **date**. The default output format will be the **DateFormat** set in the script.

Syntax:

```
MonthStart(date[, period_no])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
date	The date to evaluate.
period_no	**period_no** is an integer, which, if 0 or omitted, indicates the month that contains **date**. Negative values in **period_no** indicate preceding months and positive values indicate succeeding months.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`monthstart('19/10/2001')`	Returns 01/10/2001.
`monthstart('19/10/2001', -1)`	Returns 01/09/2001.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example finds the first day in the month of each invoice date in the table, where the base_date is shifted by four months by specifying period_no as 4.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
MonthStart(InvDate, 4) AS MthStart
```

```
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the monthstart() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	MthStart
28/03/2012	01/07/2012
10/12/2012	01/04/2013
5/2/2013	01/06/2013
31/3/2013	01/07/2013
19/5/2013	01/09/2013
15/9/2013	01/01/2014
11/12/2013	01/04/2014
2/3/2014	01/07/2014
14/5/2014	01/09/2014
13/6/2014	01/10/2014
7/7/2014	01/11/2014
4/8/2014	01/12/2014

networkdays

The **networkdays** function returns the number of working days (Monday-Friday) between and including **start_date** and **end_date** taking into account any optionally listed **holiday**.

Syntax:

```
networkdays (start_date, end_date [, holiday])
```

Return data type: integer

Arguments:

Arguments

Argument	Description
start_date	The start date to evaluate.
end_date	The end date to evaluate.

Argument	Description
holiday	Holiday periods to exclude from working days. A holiday period is stated as a start date and an end date, separated by commas. **Example:** `'25/12/2013', '26/12/2013'` You can specify more than one holiday period, separated by commas. **Example:** `'25/12/2013', '26/12/2013', '31/12/2013', '01/01/2014'` Holiday periods to exclude from working days. A holiday is stated as a string constant date. You can specify multiple holiday dates, separated by commas.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`networkdays ('19/12/2013', '07/01/2014')`	Returns 14. This example does not take holidays into account.
`networkdays ('19/12/2013', '07/01/2014', '25/12/2013', '26/12/2013')`	Returns 12. This example takes the holiday 25/12/2013 to 26/12/2013 into account.
`networkdays ('19/12/2013', '07/01/2014', '25/12/2013', '26/12/2013', '31/12/2013', '01/01/2014')`	Returns 10. This example takes two holiday periods into account.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
PayTable:
LOAD recno() as InvID, * INLINE [
InvRec|InvPaid
28/03/2012|28/04/2012
10/12/2012|01/01/2013
5/2/2013|5/3/2013
31/3/2013|01/5/2013
19/5/2013|12/6/2013
15/9/2013|6/10/2013
11/12/2013|12/01/2014
2/3/2014|2/4/2014
14/5/2014|14/6/2014
13/6/2014|14/7/2014
7/7/2014|14/8/2014
4/8/2014|4/9/2014
] (delimiter is '|');
NrDays:
```

```
Load *,
NetWorkDays(InvRec,InvPaid) As PaidDays
Resident PayTable;
Drop table PayTable;
```

The resulting table shows the returned values of NetworkDays for each of the records in the table.

Results table

InvID	InvRec	InvPaid	PaidDays
1	28/03/2012	28/04/2012	23
2	10/12/2012	01/01/2013	17
3	5/2/2013	5/3/2013	21
4	31/3/2013	01/5/2013	23
5	19/5/2013	12/6/2013	18
6	15/9/2013	6/10/2013	15
7	11/12/2013	12/01/2014	23
8	2/3/2014	2/4/2014	23
9	14/5/2014	14/6/2014	23
10	13/6/2014	14/7/2014	22
11	7/7/2014	14/8/2014	29
12	4/8/2014	4/9/2014	24

now

This function returns a timestamp of the current time from the system clock. The default value is 1.

Syntax:

```
now([ timer_mode])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
timer_mode	Can have the following values: 0 (time at last finished data load) 1 (time at function call) 2 (time when the app was opened) *If you use the function in a data load script, **timer_mode=0** will result in the time of the last finished data load, while **timer_mode=1** will give the time of the function call in the current data load.*

Examples and results:

Scripting examples

Example	Result
`now( 0)`	Returns the time when the last data load completed.
`now( 1)`	When used in a visualization expression, this returns the time of the function call. When used in a data load script, this returns the time of the function call in the current data load.
`now( 2)`	Returns the time when the app was opened.

quarterend

This function returns a value corresponding to a timestamp of the last millisecond of the quarter containing **date**. The default output format will be the **DateFormat** set in the script.

Syntax:

```
QuarterEnd(date[, period_no[, first_month_of_year]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
date	The date to evaluate.
period_no	**period_no** is an integer, where the value 0 indicates the quarter which contains **date**. Negative values in **period_no** indicate preceding quarters and positive values indicate succeeding quarters.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`quarterend('29/10/2005')`	Returns 31/12/2005 23:59:59.
`quarterend('29/10/2005', -1)`	Returns 30/09/2005 23:59:59.
`quarterend('29/10/2005', 0, 3)`	Returns 30/11/2005 23:59:59.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example finds the last day in the quarter of each invoice date in the table, where the first month in the year is specified as month 3.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
```

```
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
QuarterEnd(InvDate, 0, 3) AS QtrEnd
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the quarterend() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	QtrEnd
28/03/2012	31/05/2012
10/12/2012	28/02/2013
5/2/2013	28/02/2013
31/3/2013	31/05/2013
19/5/2013	31/05/2013
15/9/2013	30/11/2013
11/12/2013	28/02/2014
2/3/2014	31/05/2014
14/5/2014	31/05/2014
13/6/2014	31/08/2014
7/7/2014	31/08/2014
4/8/2014	31/08/2014

quartername

This function returns a display value showing the months of the quarter (formatted according to the **MonthNames** script variable) and year with an underlying numeric value corresponding to a timestamp of the first millisecond of the first day of the quarter.

Syntax:

```
QuarterName(date[, period_no[, first_month_of_year]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
date	The date to evaluate.
period_no	**period_no** is an integer, where the value 0 indicates the quarter which contains **date**. Negative values in **period_no** indicate preceding quarters and positive values indicate succeeding quarters.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

Examples and results:

Scripting examples

Example	Result
`quartername('29/10/2013')`	Returns Oct-Dec 2013.
`quartername('29/10/2013', -1)`	Returns Jul-Sep 2013.
`quartername('29/10/2013', 0, 3)`	Returns Sep-Nov 2013.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

In this example, for each invoice date in the table, the quarter name is created based on the quarter containing *InvID*. The first month of the year is specified as month 4.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];
```

```
InvoiceData:
LOAD *,
QuarterName(InvDate, 0, 4) AS QtrName
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the quartername() function.

Results table

InvDate	QtrName
28/03/2012	Jan-Mar 2011
10/12/2012	Oct-Dec 2012
5/2/2013	Jan-Mar 2012
31/3/2013	Jan-Mar 2012
19/5/2013	Apr-Jun 2013
15/9/2013	Jul-Sep 2013
11/12/2013	Oct-Dec 2013
2/3/2014	Jan-Mar 2013
14/5/2014	Apr-Jun 2014
13/6/2014	Apr-Jun 2014
7/7/2014	Jul-Sep 2014
4/8/2014	Jul-Sep 2014

quarterstart

This function returns a value corresponding to a timestamp of the first millisecond of the quarter containing **date**. The default output format will be the **DateFormat** set in the script.

Syntax:

```
QuarterStart(date[, period_no[, first_month_of_year]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
date	The date to evaluate.

Argument	Description
period_no	**period_no** is an integer, where the value 0 indicates the quarter which contains **date**. Negative values in **period_no** indicate preceding quarters and positive values indicate succeeding quarters.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`quarterstart('29/10/2005')`	Returns 01/10/2005.
`quarterstart('29/10/2005', -1 )`	Returns 01/07/2005.
`quarterstart('29/10/2005', 0, 3)`	Returns 01/09/2005.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example finds the first day in the quarter of each invoice date in the table, where the first month in the year is specified as month 3.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
QuarterStart(InvDate, 0, 3) AS QtrStart
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the quarterstart() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	QtrStart
28/03/2012	01/03/2012
10/12/2012	01/12/2012
5/2/2013	01/12/2012
31/3/2013	01/03/2013
19/5/2013	01/03/2013
15/9/2013	01/09/2013
11/12/2013	01/12/2013
2/3/2014	01/03/2014
14/5/2014	01/03/2014
13/6/2014	01/06/2014
7/7/2014	01/06/2014
4/8/2014	01/06/2014

second

This function returns an integer representing the second when the fraction of the **expression** is interpreted as a time according to the standard number interpretation.

Syntax:

```
second (expression)
```

Return data type: integer

Examples and results:

Scripting examples

Example	Result
`second( '09:14:36' )`	returns 36
`second( '0.5555' )`	returns 55 (Because 0.5555 = 13:19:55)

setdateyear

This function takes as input a **timestamp** and a **year** and updates the **timestamp** with the **year** specified in input.

Syntax:

```
setdateyear (timestamp, year)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
timestamp	A standard Qlik Sense timestamp (often just a date).
year	A four-digit year.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`setdateyear ('29/10/2005', 2013)`	Returns '29/10/2013
`setdateyear ('29/10/2005 04:26:14', 2013)`	Returns '29/10/2013 04:26:14' To see the time part of the timestamp in a visualization, you must set the number formatting to Date and choose a value for Formatting that displays time values.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
SetYear:
Load *,
SetDateYear(testdates, 2013) as NewYear
Inline [
testdates
1/11/2012
10/12/2012
1/5/2013
2/1/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];
```

The resulting table contains the original dates and a column in which the year has be set to 2013.

Results table

testdates	NewYear
1/11/2012	1/11/2013
10/12/2012	10/12/2013
2/1/2012	2/1/2013
1/5/2013	1/5/2013
19/5/2013	19/5/2013
15/9/2013	15/9/2013
11/12/2013	11/12/2013
2/3/2014	2/3/2013
14/5/2014	14/5/2013
13/6/2014	13/6/2013
7/7/2014	7/7/2013
4/8/2014	4/8/2013

setdateyearmonth

This function takes as input a **timestamp**, a **month** and a **year** and updates the **timestamp** with the **year** and the **month** specified in input. .

Syntax:

```
SetDateYearMonth (timestamp, year, month)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
timestamp	A standard Qlik Sense timestamp (often just a date).
year	A four-digit year.
month	A one or two-digit month.

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`setdateyearmonth ('29/10/2005', 2013, 3)`	Returns '29/03/2013
`setdateyearmonth ('29/10/2005 04:26:14', 2013, 3)`	Returns '29/03/2013 04:26:14' To see the time part of the timestamp in a visualization, you must set the number formatting to Date and choose a value for Formatting that displays time values.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
SetYearMonth:
Load *,
SetDateYearMonth(testdates, 2013,3) as NewYearMonth
Inline [
testdates
1/11/2012
10/12/2012
2/1/2013
19/5/2013
15/9/2013
11/12/2013
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];
```

The resulting table contains the original dates and a column in which the year has be set to 2013.

Results table

testdates	NewYearMonth
1/11/2012	1/3/2013
10/12/2012	10/3/2013
2/1/2012	2/3/2013
19/5/2013	19/3/2013
15/9/2013	15/3/2013
11/12/2013	11/3/2013
14/5/2014	14/3/2013
13/6/2014	13/3/2013

testdates	NewYearMonth
7/7/2014	7/3/2013
4/8/2014	4/3/2013

timezone

This function returns the name of the current time zone, as defined in Windows.

Syntax:

```
TimeZone( )
```

Return data type: string

Example:

```
timezone( )
```

today

This function returns the current date from the system clock.

Syntax:

```
today([ timer_mode])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
timer_mode	Can have the following values: 0 (day of last finished data load) 1 (day of function call) 2 (day when the app was opened) *If you use the function in a data load script, **timer_mode=0** will result in the day of the last finished data load, while **timer_mode=1** will give the day of the current data load.*

Examples and results:

Scripting examples

Example	Result
`Today( 0)`	Returns the day of the last finished data load.
`Today( 1)`	When used in a visualization expression, this returns the day of the function call. When used in a data load script, this returns the day when the current data load started.
`Today( 2)`	Returns the day when the app was opened.

UTC

Returns the current Coordinated Universal Time.

Syntax:

```
UTC( )
```

Return data type: dual

Example:

```
utc( )
```

week

This function returns an integer representing the week number according to ISO 8601. The week number is calculated from the date interpretation of the expression, according to the standard number interpretation.

Syntax:

```
week(timestamp [, first_week_day [, broken_weeks [, reference_day]]])
```

Return data type: integer

Arguments

Argument	Description
timestamp	The date to evaluate as a timestamp or expression resolving to a timestamp, to convert, for example '2012-10-12'.

Argument	Description
first_week_day	If you don't specify **first_week_day**, the value of variable **FirstWeekDay** will be used as the first day of the week. If you want to use another day as the first day of the week, set **first_week_day** to: • 0 for Monday • 1 for Tuesday • 2 for Wednesday • 3 for Thursday • 4 for Friday • 5 for Saturday • 6 for Sunday The integer returned by the function will now use the first day of the week that you set with **first_week_day**.
broken_weeks	If you don't specify **broken_weeks**, the value of variable **BrokenWeeks** will be used to define if weeks are broken or not. By default Qlik Sense functions use unbroken weeks. This means that: • In some years, week 1 starts in December, and in other years, week 52 or 53 continues into January. • Week 1 always has at least 4 days in January. The alternative is to use broken weeks. • Week 52 or 53 do not continue into January. • Week 1 starts on January 1 and is, in most cases, not a full week. The following values can be used: • 0 (=use unbroken weeks) • 1 (= use broken weeks)

Argument	Description
reference_day	If you don't specify **reference_day**, the value of variable **ReferenceDay** will be used to define which day in January to set as reference day to define week 1. By default, Qlik Sense functions use 4 as the reference day. This means that week 1 must contain January 4, or put differently, that week 1 must always have at least 4 days in January. The following values can be used to set a different reference day: • 1 (= January 1) • 2 (= January 2) • 3 (= January 3) • 4 (= January 4) • 5 (= January 5) • 6 (= January 6) • 7 (= January 7)

Examples and results:

Scripting examples

Example	Result
`week( '2012-10-12' )`	returns 41.
`week( '35648' )`	returns 32, because 35648 = 1997-08-06
`week('2012-10-12', 0, 1)`	returns 42

weekday

This function returns a dual value with:

- A day name as defined in the environment variable **DayNames**.
- An integer between 0-6 corresponding to the nominal day of the week (0-6).

Syntax:

```
weekday(date [,first_week_day=0])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
date	The date to evaluate.

Argument	Description
first_week_day	If you do not specify **first_week_day**, the value of variable **FirstWeekDay** will be used as the first day of the week. If you want to use another day as the first day of the week, set **first_week_day** to: • 0 for Monday • 1 for Tuesday • 2 for Wednesday • 3 for Thursday • 4 for Friday • 5 for Saturday • 6 for Sunday The integer returned by the function will now use the first day of the week that you set with **first_week_day** as base (0). *FirstWeekDay (page 182)*

Example: Chart expression
Unless stated otherwise **FirstWeekDay** is set to 0 in these examples.

Scripting examples

Example	Result
`weekday( '1971-10-12' )`	returns 'Tue' and 1
`weekday( '1971-10-12' , 6)`	returns 'Tue' and 2. In this example we use Sunday (6) as the first day of the week.

`SET FirstWeekDay = 6;`
 	
...
`weekday( '1971-10-12')`
	returns 'Tue' and 2.

Example: Load script

Load script

weekday can be used in a load script to return a string and a number representing a day of the week, even if *FirstWeekDay* and *ReferenceDay* are already set in the script. The load script below includes specific *FirstWeekDay* and *ReferenceDay* values and then uses *weekday* to return both strings and numbers that represent days of the week from the data in the *transaction_date* column.

In the results shown, the *Day* column contains the strings returned, while *Numeric value of Day* and *Numeric value of week starting from Sunday* contain the numeric values returned. In the load script *weekday* is multiplied by 1 as a simple way to make sure that the data type returned is numeric.

In the **Data load editor**, create a new section, and then add the example script and run it. Then add, at least, the fields listed in the results column to a sheet in your app to see the result.

```
SET DateFormat='DD/MM/YYYY';
SET TimestampFormat='DD/MM/YYYY h:mm:ss[.fff] TT';
SET FirstWeekDay=0;
SET BrokenWeeks=1;
SET ReferenceDay=0;
SET DayNames='Mon;Tue;Wed;Thu;Fri;Sat;Sun';
SET LongDayNames='Monday;Tuesday;Wednesday;Thursday;Friday;Saturday;Sunday';

Transactions:
Load
*,
WeekDay(transaction_date) as [Day],
1*WeekDay(transaction_date) as [Numeric value of Day]
1*WeekDay(transaction_date, 6) as [Numeric value of a week starting from Sunday],
;
Load * Inline [
transaction_id, transaction_date, transaction_amount, transaction_quantity, discount,
customer_id, size, color_code
3750, 20180830, 12423.56, 23, 0,2038593, L, Red
3751, 20180907, 5356.31, 6, 0.1, 203521, m, orange
3752, 20180916, 15.75, 1, 0.22, 5646471, S, blue
3753, 20180922, 1251, 7, 0, 3036491, l, Black
3754, 20180922, 21484.21, 1356, 75, 049681, xs, Red
3756, 20180922, -59.18, 2, 0.333333333333333, 2038593, M, Blue
3757, 20180923, 3177.4, 21, .14, 203521, XL, Black
];
```

Results

Qlik Sense table showing results of the weekday function being used in the load script.

transaction_id	transaction_ date	Day	Numeric value of Day	Numeric value for a week starting from Sunday
3750	20180830	Thu	3	4
3751	20180907	Thu	3	4
3752	20180916	Sat	5	6
3753	20180922	Fri	4	5
3754	20180922	Fri	4	5
3756	20180922	Fri	4	5
3757	20180923	Sat	5	6

weekend

This function returns a value corresponding to a timestamp of the last millisecond of the last day (Sunday) of the calendar week containing **date** The default output format will be the **DateFormat** set in the script.

Syntax:

```
WeekEnd(date [, period_no[, first_week_day]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
date	The date to evaluate.
period_no	**shift** is an integer, where the value 0 indicates the week which contains **date**. Negative values in shift indicate preceding weeks and positive values indicate succeeding weeks.
first_week_day	Specifies the day on which the week starts. If omitted, the value of variable **FirstWeekDay** is used. The possible values **first_week_day** are: • 0 for Monday • 1 for Tuesday • 2 for Wednesday • 3 for Thursday • 4 for Friday • 5 for Saturday • 6 for Sunday *FirstWeekDay (page 182)*

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Example	Result
`weekend('10/01/2013')`	Returns 12/01/2013 23:59:59.
`weekend('10/01/2013', -1)`	Returns 06/01/2013 23:59:59.
`weekend('10/01/2013', 0, 1)`	Returns 14/01/2013 23:59:59.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example finds the final day in the week following the week of each invoice date in the table.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];

InvoiceData:
LOAD *,
WeekEnd(InvDate, 1) AS WkEnd
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the weekend() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	WkEnd
28/03/2012	08/04/2012
10/12/2012	23/12/2012
5/2/2013	17/02/2013
31/3/2013	07/04/2013
19/5/2013	26/05/2013
15/9/2013	22/09/2013
11/12/2013	22/12/2013
2/3/2014	09/03/2014
14/5/2014	25/05/2014

13/6/2014	22/06/2014
7/7/2014	20/07/2014
4/8/2014	17/08/2014

weekname

This function returns a value showing the year and week number with an underlying numeric value corresponding to a timestamp of the first millisecond of the first day of the week containing **date**.

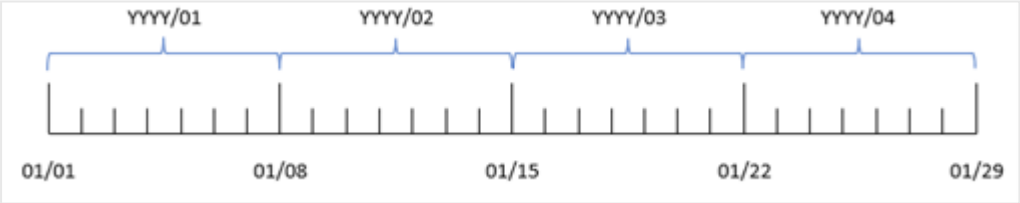
Syntax:

```
WeekName(date[, period_no[,first_week_day]])
```

The `weekname()` function determines which week the date falls into and returns the week number and year of that week. The first day of the week is determined by the `FirstWeekDay` system variable. However, you can also change the first day of the week by using the `first_week_day` argument in the `weekname()` function.

By default, Qlik Sense applications use broken weeks (defined by the `BrokenWeeks` system variable) and therefore the week number count begins on the January 1 and ends on the day prior to the `FirsWeekDay` system variable regardless of how many days have occurred.

Diagram of weekname function.



However, if your application is using unbroken weeks, week 1 can begin in the previous year or in the first few days in January. This depends on how you use the `ReferenceDay` and `FirstWeekDay` system variables.

When to use it

The `weekname()` function is useful for when you would like to compare aggregations by weeks.

For example, if you want to see the total sales of products by week. To maintain consistency with the `BrokenWeeks` system variable in the application, use `weekname()` instead of `lunarweekname()`. If the application is using unbroken weeks, week 1 may contain dates from December of the previous year or exclude dates in January of the current year. If the application is using broken weeks, week 1 may contain less than seven days.

Return data type: dual

Arguments

Argument	Description
date	The date to evaluate.
period_no	**shift** is an integer, where the value 0 indicates the week which contains **date**. Negative values in shift indicate preceding weeks and positive values indicate succeeding weeks.
first_week_day	Specifies the day on which the week starts. If omitted, the value of variable **FirstWeekDay** is used. *FirstWeekDay (page 182)*

You can use the following values to set the day on which the week starts in the `first_week_day` argument:

first_week_day values

Day	Value
Monday	0
Tuesday	1
Wednesday	2
Thursday	3
Friday	4
Saturday	5
Sunday	6

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Function examples

Example	Result
weekname('01/12/2013')	Returns 2013/02.
weekname('01/12/2013', -1)	Returns 2013/01.
weekname('01/12/2013', 0, 1)	Returns 2013/02.

Example 1 – Date with no additional arguments

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions for the last week of 2021 and first two weeks of 2022 is loaded into a table called 'Transactions'.
- The DateFormat system variable which is set to the MM/DD/YYYY format.
- The BrokenWeeks system variable which is set to 1.
- The FirstWeekDay system variable which is set to 6.
- A preceding load which contains the following:
  - The weekday() function which is set as the field, 'week_number', that returns the year and week number when the transactions took place.
  - The weekname() function which is set as the field called 'week_day', to show the weekday value of each transaction date.

Load script

```
SET BrokenWeeks=1;
SET DateFormat='MM/DD/YYYY';
SET FirstWeekDay=6;

Transactions:
   Load
     *,
     WeekDay(date) as week_day,
     Weekname(date) as week_number
     ;
Load
*
Inline
[
id,date,amount
8183,12/27/2021,58.27
8184,12/28/2021,67.42
8185,12/29/2021,23.80
```

```
8186,12/30/2021,82.06
8187,12/31/2021,40.56
8188,01/01/2022,37.23
8189,01/02/2022,17.17
8190,01/03/2022,88.27
8191,01/04/2022,57.42
8192,01/05/2022,53.80
8193,01/06/2022,82.06
8194,01/07/2022,40.56
8195,01/08/2022,53.67
8196,01/09/2022,26.63
8197,01/10/2022,72.48
8198,01/11/2022,18.37
8199,01/12/2022,45.26
8200,01/13/2022,58.23
8201,01/14/2022,18.52
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `id`
- `date`
- `week_day`
- `week_number`

Results table

id	date	week_day	week_number
8183	12/27/2021	Mon	2021/53
8184	12/28/2021	Tue	2021/53
8185	12/29/2021	Wed	2021/53
8186	12/30/2021	Thu	2021/53
8187	12/31/2021	Fri	2021/53
8188	01/01/2022	Sat	2022/01
8189	01/02/2022	Sun	2022/02
8190	01/03/2022	Mon	2022/02
8191	01/04/2022	Tue	2022/02
8192	01/05/2022	Wed	2022/02
8193	01/06/2022	Thu	2022/02
8194	01/07/2022	Fri	2022/02
8195	01/08/2022	Sat	2022/02

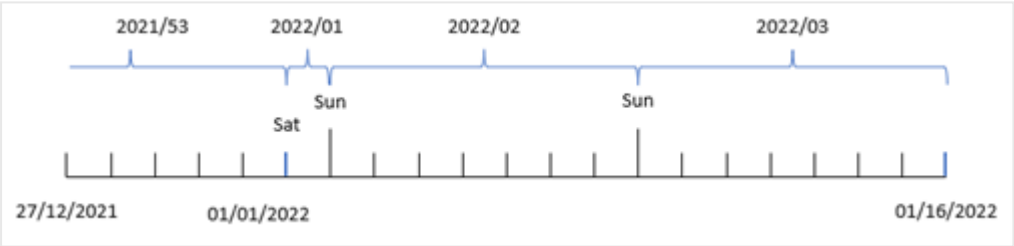
id	date	week_day	week_number
8196	01/09/2022	Sun	2022/03
8197	01/10/2022	Mon	2022/03
8198	01/11/2022	Tue	2022/03
8199	01/12/2022	Wed	2022/03
8200	01/13/2022	Thu	2022/03
8201	01/14/2022	Fri	2022/03

The 'week_number' field is created in the preceding load statement by using the `weekname()` function and passing the date field as the function's argument.

The `weekname()` function initially identifies which week the date value falls into and returns the week number count and the year the transaction takes place.

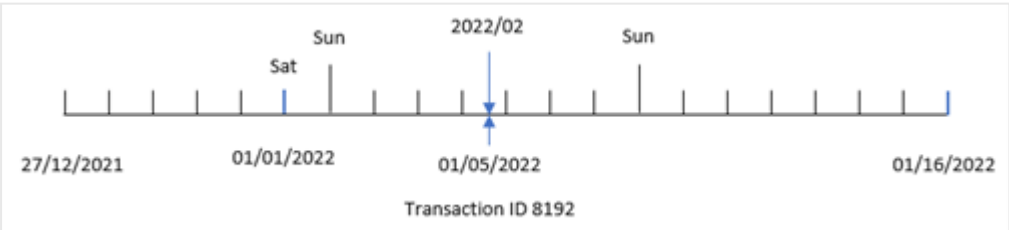
The `FirstWeekDay` system variable sets Sunday as the first day of the week. The `BrokenWeeks` system variable sets the application to use broken weeks, meaning that week 1 will begin on January 1.

Diagram of `weekname()` function with the default variables.



Week 1 begins on January 1, which is a Saturday, and therefore transactions occurring on this date return the value `2022/01` (the year and week number).

Diagram of `weekname()` function identifying the week number of transaction 8192.



Because the application is using broken weeks and the first weekday is Sunday, transactions occurring from January 2 to January 8 return the value `2022/02` (week number 2 in 2022.) An example of this would be transaction 8192 which took place on January 5 and returns the value `2022/02` for the 'week_number' field.

Example 2 – period_no

Load script and results

Overview

The same dataset and scenario as the first example are used.

However, in this example, the task is to create a field, ‘`previous_week_number`’, that returns the year, and week number, prior to when the transactions took place.

Open the Data load editor and add the following load script to a new tab.

Load script

```
SET BrokenWeeks=1;
SET FirstWeekDay=6;

Transactions:
   Load
     *,
     weekname(date,-1) as previous_week_number
     ;
Load
*
Inline
[
id,date,amount
8183,12/27/2021,58.27
8184,12/28/2021,67.42
8185,12/29/2021,23.80
8186,12/30/2021,82.06
8187,12/31/2021,40.56
8188,01/01/2022,37.23
8189,01/02/2022,17.17
8190,01/03/2022,88.27
8191,01/04/2022,57.42
8192,01/05/2022,53.80
8193,01/06/2022,82.06
8194,01/07/2022,40.56
8195,01/08/2022,53.67
8196,01/09/2022,26.63
8197,01/10/2022,72.48
8198,01/11/2022,18.37
8199,01/12/2022,45.26
8200,01/13/2022,58.23
8201,01/14/2022,18.52
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

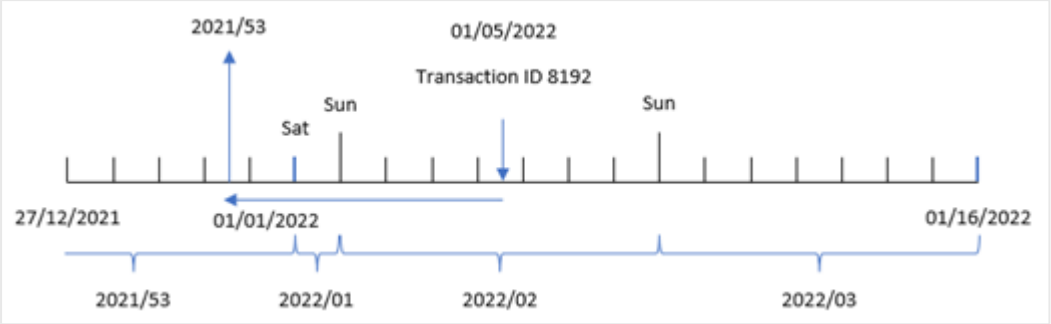
- `id`
- `date`
- `week_day`
- `week_number`

Results table

id	date	week_day	week_number
8183	12/27/2021	Mon	2021/52
8184	12/28/2021	Tue	2021/52
8185	12/29/2021	Wed	2021/52
8186	12/30/2021	Thu	2021/52
8187	12/31/2021	Fri	2021/52
8188	01/01/2022	Sat	2021/52
8189	01/02/2022	Sun	2021/53
8190	01/03/2022	Mon	2021/53
8191	01/04/2022	Tue	2021/53
8192	01/05/2022	Wed	2021/53
8193	01/06/2022	Thu	2021/53
8194	01/07/2022	Fri	2021/53
8195	01/08/2022	Sat	2022/01
8196	01/09/2022	Sun	2022/02
8197	01/10/2022	Mon	2022/02
8198	01/11/2022	Tue	2022/02
8199	01/12/2022	Wed	2022/02
8200	01/13/2022	Thu	2022/02
8201	01/14/2022	Fri	2022/02

Because a `period_no` of `-1` is used as the offset argument in the `weekname()` function, the function first identifies the week that the transactions take place in. It then looks one week prior and identifies the first millisecond of that week.

Diagram of `weekname()` function with a `period_no` offset of -1.



Transaction 8192 took place on January 5, 2022. The `weekname()` function looks one week prior, December 30, 2021, and returns the week number and year for that date – `2021/53`.

Example 3 – first_week_day

Load script and results

Overview

The same dataset and scenario as the first example are used.

However, in this example, the company policy is for the work week to begin on Tuesday.

Open the Data load editor and add the following load script to a new tab.

Load script

```
SET BrokenWeeks=1;
SET DateFormat='MM/DD/YYYY';

Transactions:
   Load
     *,
     weekday(date) as week_day,
     weekname(date,0,1) as week_number
     ;
Load
*
Inline
[
id,date,amount
8183,12/27/2021,58.27
8184,12/28/2021,67.42
8185,12/29/2021,23.80
8186,12/30/2021,82.06
8187,12/31/2021,40.56
8188,01/01/2022,37.23
8189,01/02/2022,17.17
8190,01/03/2022,88.27
8191,01/04/2022,57.42
8192,01/05/2022,53.80
```

```
8193,01/06/2022,82.06
8194,01/07/2022,40.56
8195,01/08/2022,53.67
8196,01/09/2022,26.63
8197,01/10/2022,72.48
8198,01/11/2022,18.37
8199,01/12/2022,45.26
8200,01/13/2022,58.23
8201,01/14/2022,18.52
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

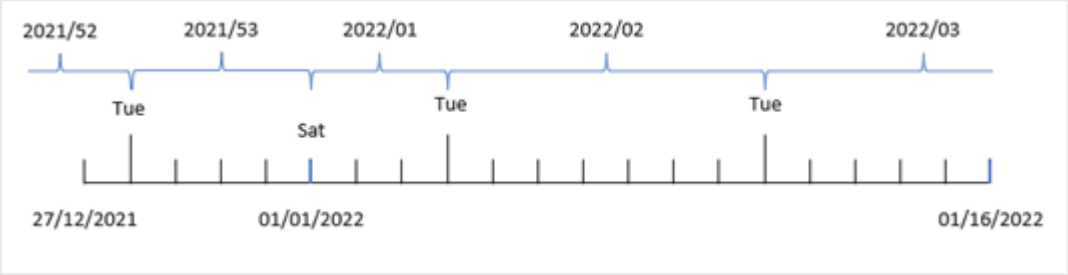
- `id`
- `date`
- `week_day`
- `week_number`

Results table

id	date	week_day	week_number
8183	12/27/2021	Mon	2021/52
8184	12/28/2021	Tue	2021/53
8185	12/29/2021	Wed	2021/53
8186	12/30/2021	Thu	2021/53
8187	12/31/2021	Fri	2021/53
8188	01/01/2022	Sat	2022/01
8189	01/02/2022	Sun	2022/01
8190	01/03/2022	Mon	2022/01
8191	01/04/2022	Tue	2022/02
8192	01/05/2022	Wed	2022/02
8193	01/06/2022	Thu	2022/02
8194	01/07/2022	Fri	2022/02
8195	01/08/2022	Sat	2022/02
8196	01/09/2022	Sun	2022/02
8197	01/10/2022	Mon	2022/02
8198	01/11/2022	Tue	2022/03
8199	01/12/2022	Wed	2022/03

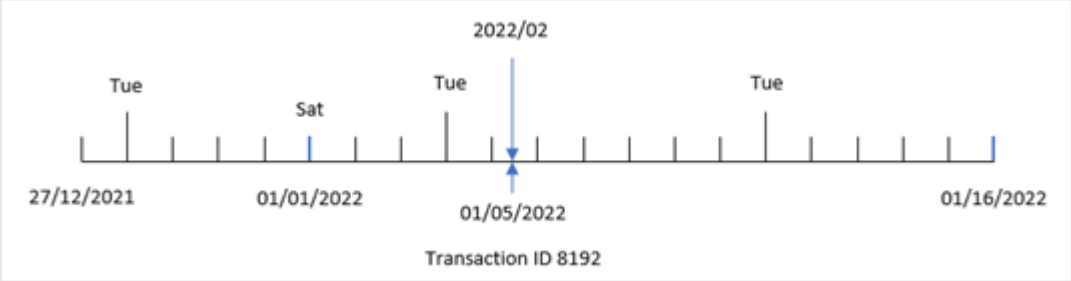
id	date	week_day	week_number
8200	01/13/2022	Thu	2022/03
8201	01/14/2022	Fri	2022/03

Diagram of `weekname()` function with Tuesday as the first day of the week.



Because the `first_week_date` argument of `1` is used in the `weekname()` function, it uses Tuesday as the first day of the week. The function therefore determines that week 53 of 2021 begins on Tuesday December 28; and, due to the application using broken weeks, week 1 begins on January 1, 2022, and ends on the last millisecond of Monday January 3, 2022.

Diagram showing week number of transaction 8192 with Tuesday as the first day of week.



Transaction 8192 took place on January 5, 2022. Therefore, using a `first_week_day` parameter of Tuesday, the `weekname()` function returns the value `2022/02` for the '`week_number`' field.

Example 4 – Chart object example

Load script and chart expression

Overview

The same dataset and scenario as the first example are used.

However, in this example, the dataset is unchanged and loaded into the application. The calculation that returns the year number of the week for when the transactions took place is created as a measure in a chart object of the application.

Load script

```
SET BrokenWeeks=1;
Transactions:
Load
*
Inline
[
id,date,amount
8183,12/27/2021,58.27
8184,12/28/2021,67.42
8185,12/29/2021,23.80
8186,12/30/2021,82.06
8187,12/31/2021,40.56
8188,01/01/2022,37.23
8189,01/02/2022,17.17
8190,01/03/2022,88.27
8191,01/04/2022,57.42
8192,01/05/2022,53.80
8193,01/06/2022,82.06
8194,01/07/2022,40.56
8195,01/08/2022,53.67
8196,01/09/2022,26.63
8197,01/10/2022,72.48
8198,01/11/2022,18.37
8199,01/12/2022,45.26
8200,01/13/2022,58.23
8201,01/14/2022,18.52
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `id`
- `date`
- `=week_day (date)`

To calculate the start of the week that a transaction takes place in, create the following measure:

```
=weekname(date)
```

Results table

id	date	=weekday(date)	=weekname(date)
8183	12/27/2021	Mon	2021/53
8184	12/28/2021	Tue	2021/53
8185	12/29/2021	Wed	2021/53
8186	12/30/2021	Thu	2021/53

id	date	=weekday(date)	=weekname(date)
8187	12/31/2021	Fri	2021/53
8188	01/01/2022	Sat	2022/01
8189	01/02/2022	Sun	2022/02
8190	01/03/2022	Mon	2022/02
8191	01/04/2022	Tue	2022/02
8192	01/05/2022	Wed	2022/02
8193	01/06/2022	Thu	2022/02
8194	01/07/2022	Fri	2022/02
8195	01/08/2022	Sat	2022/02
8196	01/09/2022	Sun	2022/03
8197	01/10/2022	Mon	2022/03
8198	01/11/2022	Tue	2022/03
8199	01/12/2022	Wed	2022/03
8200	01/13/2022	Thu	2022/03
8201	01/14/2022	Fri	2022/03

The ‘`week_number`’ field is created as a measure in the chart object by using the `weekname()` function and passing the date field as the function’s argument.

The `weekname()` function initially identifies which week the date value falls into and returns the week number count and the year that the transaction takes place.

The `FirstWeekDay` system variable sets Sunday as the first day of the week. The `BrokenWeeks` system variable sets the application to use broken weeks, meaning that week 1 begins on January 1.

Diagram showing week number with Sunday as the first day of the week.

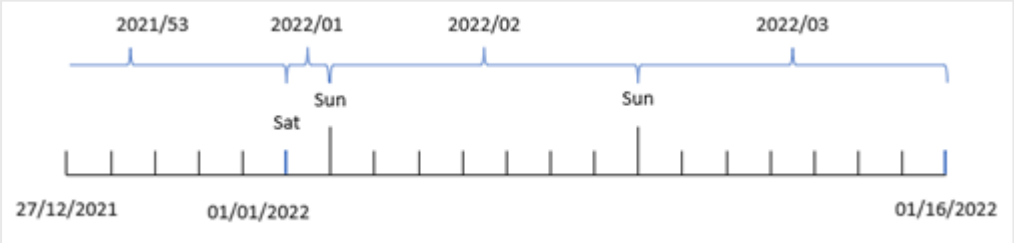
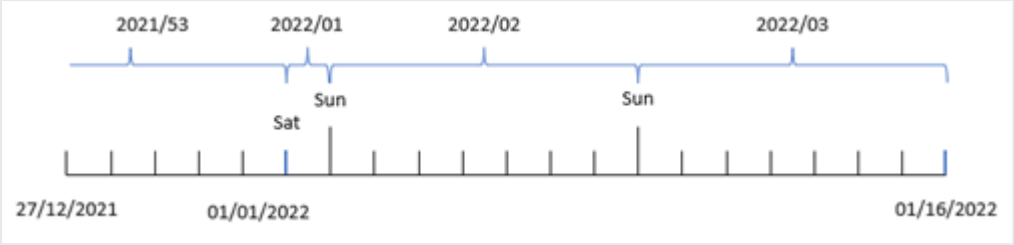


Diagram showing that transaction 8192 took place in week number two.



Because the application is using broken weeks and the first weekday is Sunday, transactions occurring from January 2 to January 8 return the value 2022/02, week number 2 in 2022. Note that transaction 8192 took place on January 5 and returns the value 2022/02 for the ‘`week_number`’ field.

Example 5 – Scenario

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions for the last week of 2019 and first two weeks of 2020 is loaded into a table called ‘`Transactions`’.
- The `BrokenWeeks` system variable which is set to `0`.
- The `ReferenceDay` system variable which is set to `2`.
- The `DateFormat` system variable which is set to the `MM/DD/YYYY` format.

Load script

```
SET BrokenWeeks=0;
SET ReferenceDay=2;
SET DateFormat='MM/DD/YYYY';

Transactions:
Load
*
Inline
[
id,date,amount
8183,12/27/2019,58.27
8184,12/28/2019,67.42
8185,12/29/2019,23.80
8186,12/30/2019,82.06
8187,12/31/2019,40.56
8188,01/01/2020,37.23
8189,01/02/2020,17.17
8190,01/03/2020,88.27
8191,01/04/2020,57.42
8192,01/05/2020,53.80
```

```
8193,01/06/2020,82.06
8194,01/07/2020,40.56
8195,01/08/2020,53.67
8196,01/09/2020,26.63
8197,01/10/2020,72.48
8198,01/11/2020,18.37
8199,01/12/2020,45.26
8200,01/13/2020,58.23
8201,01/14/2020,18.52
];
```

Results

Load the data and open a sheet. Create a new table.

Create a calculated dimension using the following expression:

```
=weekname(date)
```

To calculate total sales create the following aggregation measure:

```
=sum(amount)
```

Set the measure's **Number Formatting** to **Money**.

Results table

weekname(date)	=sum(amount)
2019/52	$125.69
2020/01	$346.51
2020/02	$347.57
2020/03	$122.01

To demonstrate the results of using the weekname() function in this scenario, add the following field as a dimension:

```
date
```

Results table with date field

weekname(date)	date	=sum(amount)
2019/52	12/27/2019	$58.27
2019/52	12/28/2019	$67.42
2020/01	12/29/2019	$23.80
2020/01	12/30/2019	$82.06
2020/01	12/31/2019	$40.56
2020/01	01/01/2020	$37.23

weekname(date)	date	=sum(amount)
2020/01	01/02/2020	$17.17
2020/01	01/03/2020	$88.27
2020/01	01/04/2020	$57.42
2020/02	01/05/2020	$53.80
2020/02	01/06/2020	$82.06
2020/02	01/07/2020	$40.56
2020/02	01/08/2020	$53.67
2020/02	01/09/2020	$26.63
2020/02	01/10/2020	$72.48
2020/02	01/11/2020	$18.37
2020/03	01/12/2020	$45.26
2020/03	01/13/2020	$58.23
2020/03	01/14/2020	$18.52

Because the application uses unbroken weeks, and week 1 requires a minimum of two days in January because of the `ReferenceDay` system variable, week 1 of 2020 includes transactions from December 29, 2019.

weekstart

This function returns a value corresponding to a timestamp of the first millisecond of the first day (Monday) of the calendar week containing **date**. The default output format is the **DateFormat** set in the script.

Syntax:

```
WeekStart(date [, period_no[, first_week_day]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
date	The date to evaluate.
period_no	**shift** is an integer, where the value 0 indicates the week which contains **date**. Negative values in shift indicate preceding weeks and positive values indicate succeeding weeks.

Argument	Description
first_week_day	Specifies the day on which the week starts. If omitted, the value of variable **FirstWeekDay** is used. The possible values **first_week_day** are: • 0 for Monday • 1 for Tuesday • 2 for Wednesday • 3 for Thursday • 4 for Friday • 5 for Saturday • 6 for Sunday *FirstWeekDay (page 182)*

Examples and results:
These examples use the date format **DD/MM/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
`weekstart('12/01/2013')`	Returns 07/01/2013.
`weekstart('12/01/2013', -1 )`	Returns 31/11/2012.
`weekstart('12/01/2013', 0, 1)`	Returns 08/01/2013.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

This example finds the first day of the week following the week of each invoice date in the table.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvDate
28/03/2012
10/12/2012
5/2/2013
31/3/2013
19/5/2013
15/9/2013
11/12/2013
2/3/2014
14/5/2014
13/6/2014
7/7/2014
4/8/2014
];
```

```
InvoiceData:
LOAD *,
WeekStart(InvDate, 1) AS WkStart
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the weekstart() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	WkStart
28/03/2012	02/04/2012
10/12/2012	17/12/2012
5/2/2013	11/02/2013
31/3/2013	01/04/2013
19/5/2013	20/05/2013
15/9/2013	16/09/2013
11/12/2013	16/12/2013
2/3/2014	03/03/2014
14/5/2014	19/05/2014
13/6/2014	16/06/2014
7/7/2014	14/07/2014
4/8/2014	11/08/2014

weekyear

This function returns the year to which the week number belongs according to ISO 8601. The week number ranges between 1 and approximately 52.

Syntax:

weekyear(expression**)**

Return data type: integer

Examples and results:

Scripting examples

Example	Result
weekyear('1996-12-30')	returns 1997, because week 1 of 1997 starts on 1996-12-30
weekyear('1997-01-02')	returns 1997

Example	Result
`weekyear( '1997-12-28' )`	returns 1997
`weekyear( '1997-12-30' )`	returns 1998, because week 1 of 1998 starts on 1997-12-29
`weekyear( '1999-01-02' )`	returns 1998, because week 53 of 1998 ends on 1999-01-03

Limitations:

Some years, week #1 starts in December, e.g. December 1997. Other years start with week #53 of previous year, e.g. January 1999. For those few days when the week number belongs to another year, the functions **year** and **weekyear** will return different values.

year

This function returns an integer representing the year when the **expression** is interpreted as a date according to the standard number interpretation.

Syntax:

```
year(expression)
```

Return data type: integer

Examples and results:

Scripting examples

Example	Result
`year( '2012-10-12' )`	returns 2012
`year( '35648' )`	returns 1997, because 35648 = 1997-08-06

yearend

This function returns a value corresponding to a timestamp of the last millisecond of the last day of the year containing **date**. The default output format will be the **DateFormat** set in the script.

Syntax:

```
YearEnd( date[, period_no[, first_month_of_year = 1]])
```

In other words, the `yearend()` function determines which year the date falls into. It then returns a timestamp, in date format, for the last millisecond of that year. The first month of the year is, by default, January. However, you can change which month is set as first by using the `first_month_of_year` argument in the `yearend()` function.

The `yearend()` function does not consider the `FirstMonthOfYear` system variable. The year begins on January 1 unless the `first_month_of_year` argument is used to change it.

Diagram of `yearend()` function.

When to use it

The `yearend()` function is used as part of an expression when you want the calculation to use the fraction of the year that has not yet occurred. For example, if you want to calculate the total interest not yet incurred during the year.

Return data type: dual

Arguments

Argument	Description
date	The date to evaluate.
period_no	**period_no** is an integer, where the value 0 indicates the year which contains **date**. Negative values in **period_no** indicate preceding years and positive values indicate succeeding years.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

You can use the following values to set the first month of year in the `first_month_of_year` argument:

first_month_of_year values

Month	Value
February	2
March	3
April	4
May	5
June	6
July	7
August	8
September	9
October	10
November	11
December	12

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Function examples

Example	Result
`yearend('10/19/2001')`	Returns 12/31/2001 23:59:59.
`yearend('10/19/2001', -1)`	Returns 12/31/2000 23:59:59.
`yearend('10/19/2001', 0, 4)`	Returns 03/31/2002 23:59:59.

Example 1 – No additional arguments

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions between 2020 and 2022 is loaded into a table called '`Transactions`'.
- The date field has been provided in the `DateFormat` system variable `(MM/DD/YYYY)` format.
- A preceding load statement which contains the following:
  - `yearend()` function which is set as the `year_end` field.
  - `Timestamp()` function which is set as the `year_end_timestamp` field.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
       *,
       yearend(date) as year_end,
       timestamp(yearend(date)) as year_end_timestamp
```

```
        ;
Load
*
Inline
[
id,date,amount
8188,01/13/2020,37.23
8189,02/26/2020,17.17
8190,03/27/2020,88.27
8191,04/16/2020,57.42
8192,05/21/2020,53.80
8193,08/14/2020,82.06
8194,10/07/2020,40.39
8195,12/05/2020,87.21
8196,01/22/2021,95.93
8197,02/03/2021,45.89
8198,03/17/2021,36.23
8199,04/23/2021,25.66
8200,05/04/2021,82.77
8201,06/30/2021,69.98
8202,07/26/2021,76.11
8203,12/27/2021,25.12
8204,06/06/2022,46.23
8205,07/18/2022,84.21
8206,11/14/2022,96.24
8207,12/12/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `id`
- `date`
- `year_end`
- `year_end_timestamp`

Results table

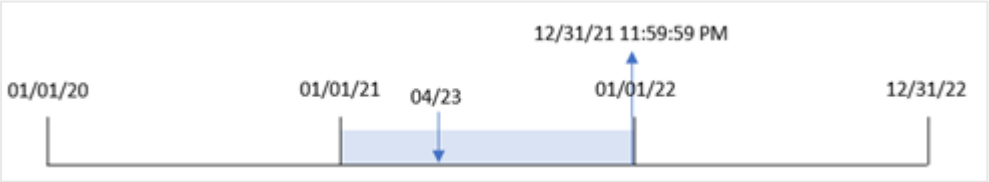
id	date	year_end	year_end_timestamp
8188	01/13/2020	12/31/2020	12/31/2020 11:59:59 PM
8189	02/26/2020	12/31/2020	12/31/2020 11:59:59 PM
8190	03/27/2020	12/31/2020	12/31/2020 11:59:59 PM
8191	04/16/2020	12/31/2020	12/31/2020 11:59:59 PM
8192	05/21/2020	12/31/2020	12/31/2020 11:59:59 PM
8193	08/14/2020	12/31/2020	12/31/2020 11:59:59 PM
8194	10/07/2020	12/31/2020	12/31/2020 11:59:59 PM

id	date	year_end	year_end_timestamp
8195	12/05/2020	12/31/2020	12/31/2020 11:59:59 PM
8196	01/22/2021	12/31/2021	12/31/2021 11:59:59 PM
8197	02/03/2021	12/31/2021	12/31/2021 11:59:59 PM
8198	03/17/2021	12/31/2021	12/31/2021 11:59:59 PM
8199	04/23/2021	12/31/2021	12/31/2021 11:59:59 PM
8200	05/04/2021	12/31/2021	12/31/2021 11:59:59 PM
8201	06/30/2021	12/31/2021	12/31/2021 11:59:59 PM
8202	07/26/2021	12/31/2021	12/31/2021 11:59:59 PM
8203	12/27/2021	12/31/2021	12/31/2021 11:59:59 PM
8204	06/06/2022	12/31/2022	12/31/2022 11:59:59 PM
8205	07/18/2022	12/31/2022	12/31/2022 11:59:59 PM
8206	11/14/2022	12/31/2022	12/31/2022 11:59:59 PM
8207	12/12/2022	12/31/2022	12/31/2022 11:59:59 PM

The ‘`year_end`’ field is created in the preceding load statement by using the `yearend()` function and passing the date field as the function’s argument.

The `yearend()` function initially identifies which year the date value falls into and returns a timestamp for the last millisecond of that year.

Diagram of `yearend()` function with transaction 8199 selected.



Transaction 8199 took place on April 23, 2021. The `yearend()` function returns the last millisecond of that year, which is December 31 at 11:59:59 PM.

Example 2 – period_no

Load script and results

Overview

The same dataset and scenario as the first example are used.

However, in this example, the task is to create a field, ‘`previous_year_end`’ , that returns the end date timestamp of the year prior to the year in which a transaction took place.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
       *,
       yearend(date,-1) as previous_year_end,
       timestamp(yearend(date,-1)) as previous_year_end_timestamp
       ;
Load
*
Inline
[
id,date,amount
8188,01/13/2020,37.23
8189,02/26/2020,17.17
8190,03/27/2020,88.27
8191,04/16/2020,57.42
8192,05/21/2020,53.80
8193,08/14/2020,82.06
8194,10/07/2020,40.39
8195,12/05/2020,87.21
8196,01/22/2021,95.93
8197,02/03/2021,45.89
8198,03/17/2021,36.23
8199,04/23/2021,25.66
8200,05/04/2021,82.77
8201,06/30/2021,69.98
8202,07/26/2021,76.11
8203,12/27/2021,25.12
8204,06/06/2022,46.23
8205,07/18/2022,84.21
8206,11/14/2022,96.24
8207,12/12/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `id`
- `date`
- `previous_year_end`
- `previous_ year_end_timestamp`

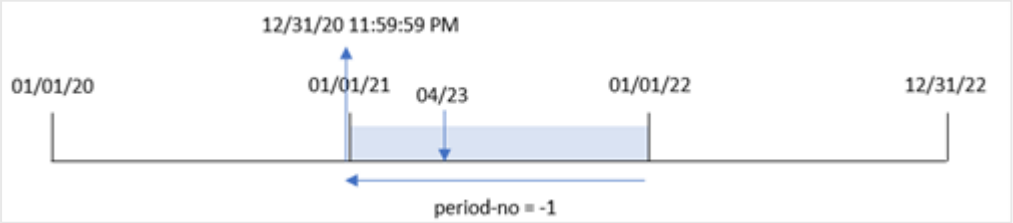
Results table

id	date	previous_year_end	previous_year_end_timestamp
8188	01/13/2020	12/31/2019	12/31/2019 11:59:59 PM

id	date	previous_year_end	previous_year_end_timestamp
8189	02/26/2020	12/31/2019	12/31/2019 11:59:59 PM
8190	03/27/2020	12/31/2019	12/31/2019 11:59:59 PM
8191	04/16/2020	12/31/2019	12/31/2019 11:59:59 PM
8192	05/21/2020	12/31/2019	12/31/2019 11:59:59 PM
8193	08/14/2020	12/31/2019	12/31/2019 11:59:59 PM
8194	10/07/2020	12/31/2019	12/31/2019 11:59:59 PM
8195	12/05/2020	12/31/2019	12/31/2019 11:59:59 PM
8196	01/22/2021	12/31/2020	12/31/2020 11:59:59 PM
8197	02/03/2021	12/31/2020	12/31/2020 11:59:59 PM
8198	03/17/2021	12/31/2020	12/31/2020 11:59:59 PM
8199	04/23/2021	12/31/2020	12/31/2020 11:59:59 PM
8200	05/04/2021	12/31/2020	12/31/2020 11:59:59 PM
8201	06/30/2021	12/31/2020	12/31/2020 11:59:59 PM
8202	07/26/2021	12/31/2020	12/31/2020 11:59:59 PM
8203	12/27/2021	12/31/2020	12/31/2020 11:59:59 PM
8204	06/06/2022	12/31/2021	12/31/2021 11:59:59 PM
8205	07/18/2022	12/31/2021	12/31/2021 11:59:59 PM
8206	11/14/2022	12/31/2021	12/31/2021 11:59:59 PM
8207	12/12/2022	12/31/2021	12/31/2021 11:59:59 PM

Because a `period_no` of `-1` was used as the offset argument in the `yearend()` function, the function first identifies the year that the transactions take place in. It then looks one year prior and identifies the last millisecond of that year.

Diagram of `yearend()` function with a period_no of -1.



Transaction 8199 takes place on April 23, 2021. The `yearend()` function returns the last millisecond of the prior year, December 31, 2020 at 11:59:59 PM, for the '`previous_year_end`' field.

Example 3 – first_month_of_year

Load script and results

Overview

The same dataset and scenario as the first example are used.

However, in this example, the company policy is for the year to begin from April 1.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
       *,
       yearend(date,0,4) as year_end,
       timestamp(yearend(date,0,4)) as year_end_timestamp
       ;
Load
*
Inline
[
id,date,amount
8188,01/13/2020,37.23
8189,02/26/2020,17.17
8190,03/27/2020,88.27
8191,04/16/2020,57.42
8192,05/21/2020,53.80
8193,08/14/2020,82.06
8194,10/07/2020,40.39
8195,12/05/2020,87.21
8196,01/22/2021,95.93
8197,02/03/2021,45.89
8198,03/17/2021,36.23
8199,04/23/2021,25.66
8200,05/04/2021,82.77
8201,06/30/2021,69.98
8202,07/26/2021,76.11
8203,12/27/2021,25.12
8204,06/06/2022,46.23
8205,07/18/2022,84.21
8206,11/14/2022,96.24
8207,12/12/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

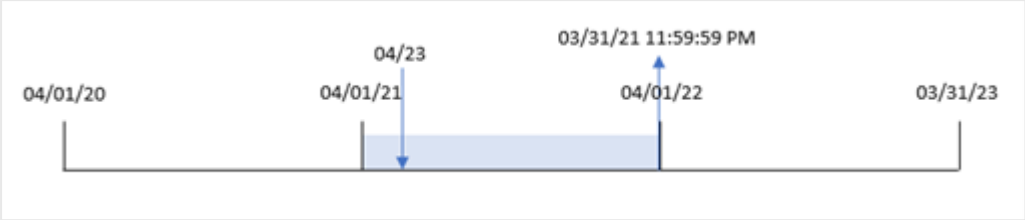
- `id`
- `date`
- `year_end`
- `year_end_timestamp`

Results table

id	date	year_end	year_end_timestamp
8188	01/13/2020	03/31/2020	3/31/2020 11:59:59 PM
8189	02/26/2020	03/31/2020	3/31/2020 11:59:59 PM
8190	03/27/2020	03/31/2020	3/31/2020 11:59:59 PM
8191	04/16/2020	03/31/2021	3/31/2021 11:59:59 PM
8192	05/21/2020	03/31/2021	3/31/2021 11:59:59 PM
8193	08/14/2020	03/31/2021	3/31/2021 11:59:59 PM
8194	10/07/2020	03/31/2021	3/31/2021 11:59:59 PM
8195	12/05/2020	03/31/2021	3/31/2021 11:59:59 PM
8196	01/22/2021	03/31/2021	3/31/2021 11:59:59 PM
8197	02/03/2021	03/31/2021	3/31/2021 11:59:59 PM
8198	03/17/2021	03/31/2021	3/31/2021 11:59:59 PM
8199	04/23/2021	03/31/2022	3/31/2022 11:59:59 PM
8200	05/04/2021	03/31/2022	3/31/2022 11:59:59 PM
8201	06/30/2021	03/31/2022	3/31/2022 11:59:59 PM
8202	07/26/2021	03/31/2022	3/31/2022 11:59:59 PM
8203	12/27/2021	03/31/2022	3/31/2022 11:59:59 PM
8204	06/06/2022	03/31/2023	3/31/2023 11:59:59 PM
8205	07/18/2022	03/31/2023	3/31/2023 11:59:59 PM
8206	11/14/2022	03/31/2023	3/31/2023 11:59:59 PM
8207	12/12/2022	03/31/2023	3/31/2023 11:59:59 PM

Because the `first_month_of_year` argument of `4` is used in the `yearend()` function, it sets the first day of the year to April 1, and the last day of the year to March 31.

Diagram of `yearend()` function with April as the first month of the year.



Transaction 8199 takes place on April 23, 2021. Because the `yearend()` function sets the start of the year to April 1, it returns March 31, 2022 as the '`year_end`' value for the transaction.

Example 4 – Chart object example

Load script and chart expression

Overview

The same dataset and scenario as the first example are used.

However, in this example, the dataset is unchanged and loaded into the application. The calculation that returns the end date timestamp of the year in which a transaction took place is created as a measure in a chart object of the application.

Load script

```
Transactions:
Load
*
Inline
[
id,date,amount
8188,01/13/2020,37.23
8189,02/26/2020,17.17
8190,03/27/2020,88.27
8191,04/16/2020,57.42
8192,05/21/2020,53.80
8193,08/14/2020,82.06
8194,10/07/2020,40.39
8195,12/05/2020,87.21
8196,01/22/2021,95.93
8197,02/03/2021,45.89
8198,03/17/2021,36.23
8199,04/23/2021,25.66
8200,05/04/2021,82.77
8201,06/30/2021,69.98
8202,07/26/2021,76.11
8203,12/27/2021,25.12
8204,06/06/2022,46.23
8205,07/18/2022,84.21
8206,11/14/2022,96.24
```

```
8207,12/12/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `id`
- `date`

To calculate in which year a transaction took place, create the following measures:

- `=yearend(date)`
- `=timestamp(yearend(date))`

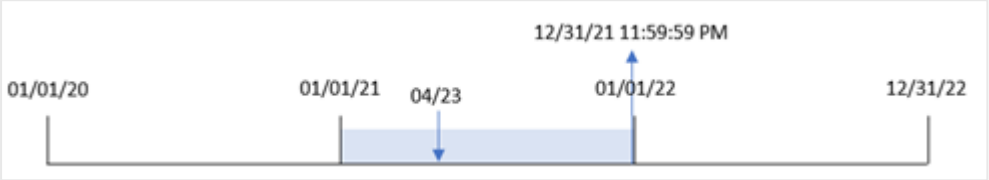
Results table

id	date	=yearend(date)	=timestamp(yearend(date))
8188	01/13/2020	12/31/2020	12/31/2020 11:59:59 PM
8189	02/26/2020	12/31/2020	12/31/2020 11:59:59 PM
8190	03/27/2020	12/31/2020	12/31/2020 11:59:59 PM
8191	04/16/2020	12/31/2020	12/31/2020 11:59:59 PM
8192	05/21/2020	12/31/2020	12/31/2020 11:59:59 PM
8193	08/14/2020	12/31/2020	12/31/2020 11:59:59 PM
8194	10/07/2020	12/31/2020	12/31/2020 11:59:59 PM
8195	12/05/2020	12/31/2020	12/31/2020 11:59:59 PM
8196	01/22/2021	12/31/2021	12/31/2021 11:59:59 PM
8197	02/03/2021	12/31/2021	12/31/2021 11:59:59 PM
8198	03/17/2021	12/31/2021	12/31/2021 11:59:59 PM
8199	04/23/2021	12/31/2021	12/31/2021 11:59:59 PM
8200	05/04/2021	12/31/2021	12/31/2021 11:59:59 PM
8201	06/30/2021	12/31/2021	12/31/2021 11:59:59 PM
8202	07/26/2021	12/31/2021	12/31/2021 11:59:59 PM
8203	12/27/2021	12/31/2021	12/31/2021 11:59:59 PM
8204	06/06/2022	12/31/2022	12/31/2022 11:59:59 PM
8205	07/18/2022	12/31/2022	12/31/2022 11:59:59 PM
8206	11/14/2022	12/31/2022	12/31/2022 11:59:59 PM
8207	12/12/2022	12/31/2022	12/31/2022 11:59:59 PM

The ‘end_of_year’ measure is created in the chart object by using the yearend() function and passing the date field as the function’s argument.

The yearend() function initially identifies which year the date value falls into returning a timestamp for the last millisecond of that year.

Diagram of yearend() function that shows Transaction 8199 took place in April.



Transaction 8199 takes place on April 23, 2021. The yearend() function returns the last millisecond of that year, which is December 31 at 11:59:59 PM.

Example 5 – Scenario

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset is loaded into a table called ‘Employee_Expenses’. The table contains the following fields:
  - employee IDs
  - employee name
  - average daily expense claims of each employee

The end user would like a chart object that displays, by employee id and employee name, the estimated expense claims still to be incurred for the remainder of the year. The financial year begins in January.

Load script

```
Employee_Expenses:
Load
*
Inline
[
employee_id,employee_name,avg_daily_claim
182,Mark, $15
183,Deryck, $12.5
184,Dexter, $12.5
185,Sydney,$27
186,Agatha,$18
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `employee_id`
- `employee_name`

To calculate the projected expense claims, create the following measure:

```
=(yearend(today(1))-today(1))*avg_daily_claim
```

Set the measure's **Number Formatting** to **Money**.

Results table

employee_id	employee_name	=(yearend(today(1))-today(1))*avg_daily_claim
182	Mark	$3240.00
183	Deryck	$2700.00
184	Dexter	$2700.00
185	Sydney	$5832.00
186	Agatha	$3888.00

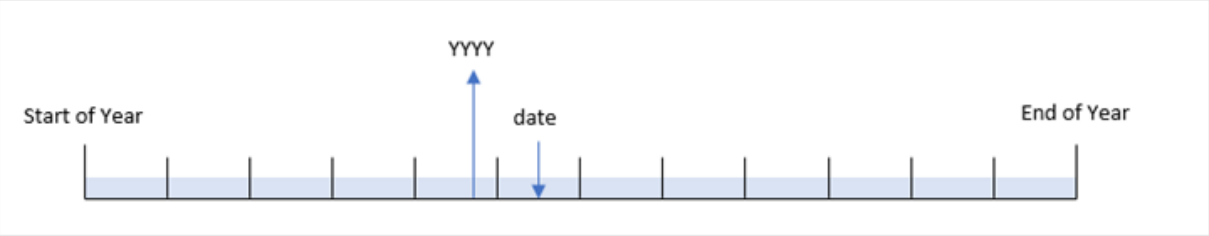
By using today's date as its only argument, the `yearend()` function returns the end date of the current year. Then, by subtracting today's date from the year end date, the expression returns the number of days remaining in this year.

This value is then multiplied by the average daily expense claim by each employee to calculate the estimated value of claims each employee is expected to make in the remaining year.

yearname

This function returns a four-digit year as display value with an underlying numeric value corresponding to a timestamp of the first millisecond of the first day of the year containing **date**.

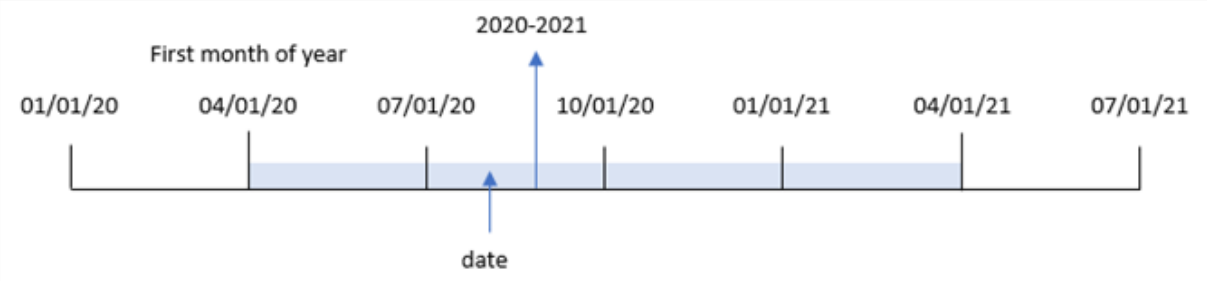
Diagram of range of time of the `yearname()` function.



The `yearname()` function is different to the `year()` function as it lets you offset the date you want evaluated and lets you set the first month of the year.

If the first month of the year is not January, the function will return the two four-digit years across the twelve month period that contain the date. For example, if the start of the year is April and the date being evaluated is 06/30/2020, the result returned would be 2020-2021.

Diagram of `yearname()` function with April set as the first month of the year.



Syntax:

```
YearName(date[, period_no[, first_month_of_year]] )
```

Return data type: dual

Argument	Description
date	The date to evaluate.
period_no	**period_no** is an integer, where the value 0 indicates the year which contains **date**. Negative values in **period_no** indicate preceding years and positive values indicate succeeding years.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**. The display value will then be a string showing two years.

You can use the following values to set the first month of year in the `first_month_of_year` argument:

first_month_of_year values

Month	Value
February	2
March	3
April	4
May	5
June	6
July	7
August	8
September	9
October	10

Month	Value
November	11
December	12

When to use it

The `yearname()` function is useful for comparing aggregations by year. For example, if you want to see the total sales of products by year.

These dimensions can be created in the load script by using the function to create a field in a Master Calendar table. They can also be created in a chart as calculated dimensions

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Function examples

Example	Result
`yearname('10/19/2001')`	Returns '2001.'
`yearname('10/19/2001',-1)`	Returns '2000.'
`yearname('10/19/2001',0,4)`	Returns '2001-2002.'

Related topics

Topic	Description
year (page 727)	This function returns an integer representing the year when the expression is interpreted as a date according to the standard number interpretation.

Example 1 – No additional arguments

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions between 2020 and 2022 is loaded into a table called ‘`Transactions`’.
- The `DateFormat` system variable which is set to ‘`MM/DD/YYYY`’.
- A preceding load that uses the `yearname()` and which is set as the `year_name` field.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
      *,
      yearname(date) as year_name
      ;
Load
*
Inline
[
id,date,amount
8188,'01/13/2020',37.23
8189,'02/26/2020',17.17
8190,'03/27/2020',88.27
8191,'04/16/2020',57.42
8192,'05/21/2020',53.80
8193,'08/14/2020',82.06
8194,'10/07/2020',40.39
8195,'12/05/2020',87.21
8196,'01/22/2021',95.93
8197,'02/03/2021',45.89
8198,'03/17/2021',36.23
8199,'04/23/2021',25.66
8200,'05/04/2021',82.77
8201,'06/30/2021',69.98
8202,'07/26/2021',76.11
8203,'12/27/2021',25.12
8204,'06/06/2022',46.23
8205,'07/18/2022',84.21
8206,'11/14/2022',96.24
8207,'12/12/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `date`
- `year_name`

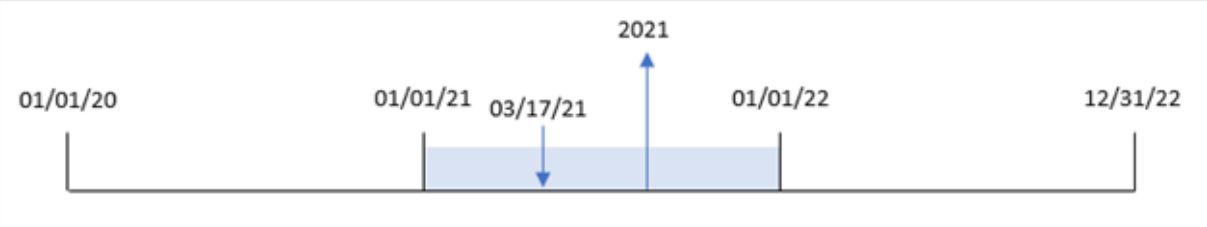
Results table

date	year_name
01/13/2020	2020
02/26/2020	2020
03/27/2020	2020
04/16/2020	2020
05/21/2020	2020
08/14/2020	2020
10/07/2020	2020
12/05/2020	2020
01/22/2021	2021
02/03/2021	2021
03/17/2021	2021
04/23/2021	2021
05/04/2021	2021
06/30/2021	2021
07/26/2021	2021
12/27/2021	2021
06/06/2022	2022
07/18/2022	2022
11/14/2022	2022
12/12/2022	2022

The '`year_name`' field is created in the preceding load statement by using the `yearname()` function and passing the date field as the function's argument.

The `yearname()` function identifies which year the date value falls into and returns this as a four-digit year value.

Diagram of `yearname()` function that shows 2021 as the year value.



Example 2 – period_no

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions between 2020 and 2022 is loaded into a table called 'Transactions'.
- The DateFormat system variable which is set to '`MM/DD/YYYY`'.
- A preceding load that uses the `yearname()` and which is set as the `year_name` field.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
      *,
      yearname(date,-1) as prior_year_name
      ;
Load
*
Inline
[
id,date,amount
8188,'01/13/2020',37.23
8189,'02/26/2020',17.17
8190,'03/27/2020',88.27
8191,'04/16/2020',57.42
8192,'05/21/2020',53.80
8193,'08/14/2020',82.06
8194,'10/07/2020',40.39
8195,'12/05/2020',87.21
8196,'01/22/2021',95.93
8197,'02/03/2021',45.89
8198,'03/17/2021',36.23
8199,'04/23/2021',25.66
8200,'05/04/2021',82.77
8201,'06/30/2021',69.98
```

```
8202,'07/26/2021',76.11
8203,'12/27/2021',25.12
8204,'06/06/2022',46.23
8205,'07/18/2022',84.21
8206,'11/14/2022',96.24
8207,'12/12/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

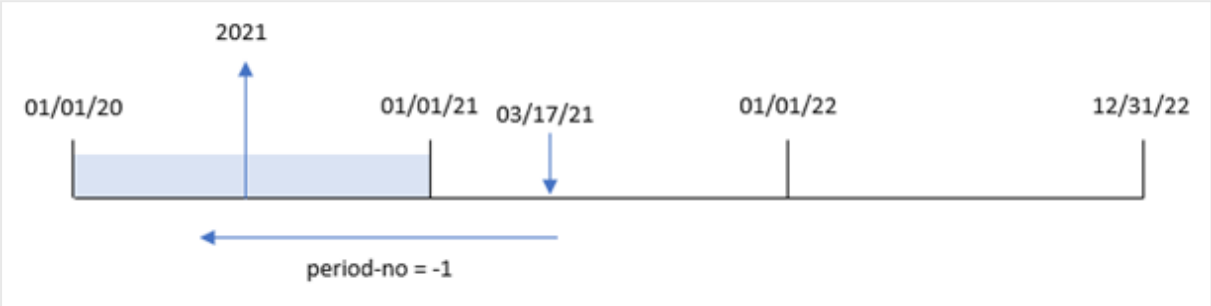
- `date`
- `prior_year_name`

Results table

date	prior_year_name
01/13/2020	2019
02/26/2020	2019
03/27/2020	2019
04/16/2020	2019
05/21/2020	2019
08/14/2020	2019
10/07/2020	2019
12/05/2020	2019
01/22/2021	2020
02/03/2021	2020
03/17/2021	2020
04/23/2021	2020
05/04/2021	2020
06/30/2021	2020
07/26/2021	2020
12/27/2021	2020
06/06/2022	2021
07/18/2022	2021
11/14/2022	2021
12/12/2022	2021

Because a `period_no` of `-1` is used as the offset argument in the `yearname()` function, the function first identifies the year that the transactions take place in. The function then shifts one year prior and returns the resulting year.

Diagram of `yearname()` function with the `period_no` set `-1`.



Example 3 – first_month_of_year

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset from the first example.
- The `DateFormat` system variable which is set to '`MM/DD/YYYY`'.
- A preceding load that uses the `yearname()` and which is set as the `year_name` field.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
      *,
      yearname(date,0,4) as year_name
      ;
Load
*
Inline
[
id,date,amount
8188,'01/13/2020',37.23
8189,'02/26/2020',17.17
8190,'03/27/2020',88.27
8191,'04/16/2020',57.42
8192,'05/21/2020',53.80
8193,'08/14/2020',82.06
8194,'10/07/2020',40.39
8195,'12/05/2020',87.21
```

```
8196,'01/22/2021',95.93
8197,'02/03/2021',45.89
8198,'03/17/2021',36.23
8199,'04/23/2021',25.66
8200,'05/04/2021',82.77
8201,'06/30/2021',69.98
8202,'07/26/2021',76.11
8203,'12/27/2021',25.12
8204,'06/06/2022',46.23
8205,'07/18/2022',84.21
8206,'11/14/2022',96.24
8207,'12/12/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `id`
- `date`
- `year_name`

Results table

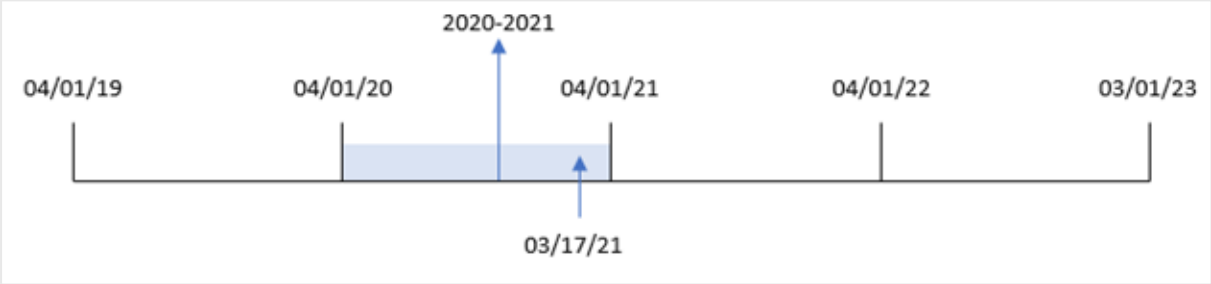
date	year_name
01/13/2020	2019-2020
02/26/2020	2019-2020
03/27/2020	2019-2020
04/16/2020	2020-2021
05/21/2020	2020-2021
08/14/2020	2020-2021
10/07/2020	2020-2021
12/05/2020	2020-2021
01/22/2021	2020-2021
02/03/2021	2020-2021
03/17/2021	2020-2021
04/23/2021	2021-2022
05/04/2021	2021-2022
06/30/2021	2021-2022
07/26/2021	2021-2022
12/27/2021	2021-2022

date	year_name
06/06/2022	2022-2023
07/18/2022	2022-2023
11/14/2022	2022-2023
12/12/2022	2022-2023

Because the `first_month_of_year` argument of 4 is used in the `yearname()` function, the start of the year moves from January 1 to April 1. Therefore, each twelve month period crosses two calendar years and the `yearname()` function returns the two four-digit years for dates evaluated.

Transaction 8198 takes place on March 17, 2021. The `yearname()` function sets the beginning of the year on April 1 and the ending on March 30. Therefore, transaction 8198 occurred in the year period from April 1, 2020 and March 30, 2021. As a result, the `yearname()` function returns the value 2020-2021.

Diagram of `yearname()` function with March set as the first month of the year.



Example 4 – Chart object example

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset from the first example.
- The `DateFormat` system variable which is set to '`MM/DD/YYYY`'.

However, the field that returns the year that the transaction took place in is created as a measure in a chart object.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
Load
*
```

```
Inline
[
id,date,amount
8188,'01/13/2020',37.23
8189,'02/26/2020',17.17
8190,'03/27/2020',88.27
8191,'04/16/2020',57.42
8192,'05/21/2020',53.80
8193,'08/14/2020',82.06
8194,'10/07/2020',40.39
8195,'12/05/2020',87.21
8196,'01/22/2021',95.93
8197,'02/03/2021',45.89
8198,'03/17/2021',36.23
8199,'04/23/2021',25.66
8200,'05/04/2021',82.77
8201,'06/30/2021',69.98
8202,'07/26/2021',76.11
8203,'12/27/2021',25.12
8204,'06/06/2022',46.23
8205,'07/18/2022',84.21
8206,'11/14/2022',96.24
8207,'12/12/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table and add this field as a dimension:

`date`

To calculate the ‘`year_name`’ field, create this measure:

`=yearname(date)`

Results table

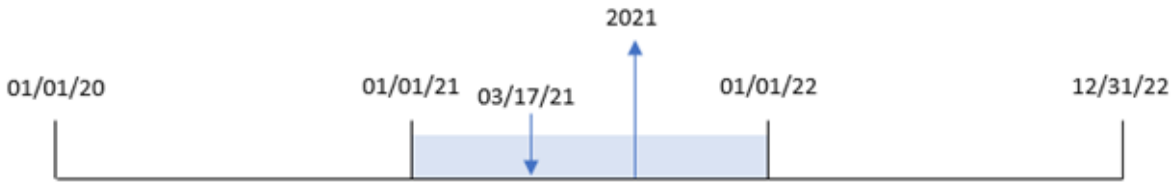
date	=yearname(date)
01/13/2020	2020
02/26/2020	2020
03/27/2020	2020
04/16/2020	2020
05/21/2020	2020
08/14/2020	2020
10/07/2020	2020
12/05/2020	2020
01/22/2021	2021
02/03/2021	2021

date	=yearname(date)
03/17/2021	2021
04/23/2021	2021
05/04/2021	2021
06/30/2021	2021
07/26/2021	2021
12/27/2021	2021
06/06/2022	2022
07/18/2022	2022
11/14/2022	2022
12/12/2022	2022

The ‘`year_name`’ measure is created in the chart object using the `yearname()` function and passing the date field as the function’s argument.

The `yearname()` function identifies which year the date value falls into and returns this as a four-digit year value.

Diagram of `yearname()` function with 2021 as the year value.



Example 5 – Scenario

Load script and chart expression

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- The same dataset from the first example.
- The `DateFormat` system variable which is set to ‘`MM/DD/YYYY`’.

The end user would like a chart that presents the total sales by quarter for the transactions. Use the `yearname()` function as a calculated dimension to create this chart when the `yearname()` dimension is not available in the data model.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
Load
*
Inline
[
id,date,amount
8188,'01/13/2020',37.23
8189,'02/26/2020',17.17
8190,'03/27/2020',88.27
8191,'04/16/2020',57.42
8192,'05/21/2020',53.80
8193,'08/14/2020',82.06
8194,'10/07/2020',40.39
8195,'12/05/2020',87.21
8196,'01/22/2021',95.93
8197,'02/03/2021',45.89
8198,'03/17/2021',36.23
8199,'04/23/2021',25.66
8200,'05/04/2021',82.77
8201,'06/30/2021',69.98
8202,'07/26/2021',76.11
8203,'12/27/2021',25.12
8204,'06/06/2022',46.23
8205,'07/18/2022',84.21
8206,'11/14/2022',96.24
8207,'12/12/2022',67.67
];
```

Results

Load the data and open a sheet. Create a new table.

To compare aggregations by year, create this calculated dimension:

```
=yearname(date)
```

Create this measure:

```
=sum(amount)
```

Set the measure's **Number Formatting** to **Money**.

Results table

yearname(date)	=sum(amount)
2020	$463.55
2021	$457.69
2022	$294.35

yearstart

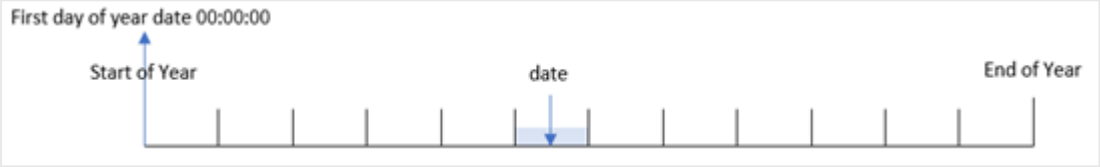
This function returns a timestamp corresponding to the start of the first day of the year containing **date**. The default output format will be the **DateFormat** set in the script.

Syntax:

```
YearStart(date[, period_no[, first_month_of_year]])
```

In other words, the `yearstart()` function determines which year the date falls into. It then returns a timestamp, in date format, for the first millisecond of that year. The first month of the year is, by default, January; however, you can change which month is set as first by using the `first_month_of_year` argument in the `yearstart()` function.

Diagram of `yearstart()` function that shows the range of time that the function can cover.



When to use it

The `yearstart()` function is used as part of an expression when you want the calculation to use the fraction of the year that has elapsed thus far. For example, if you want to calculate the interest that has accumulated in a year to date.

Return data type: dual

Arguments

Argument	Description
date	The date to evaluate.
period_no	**period_no** is an integer, where the value 0 indicates the year which contains **date**. Negative values in **period_no** indicate preceding years and positive values indicate succeeding years.
first_month_of_year	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in **first_month_of_year**.

The following months can be used in the `first_month_of_year argument`:

first_month_of_year values

Month	Value
February	2
March	3

Month	Value
April	4
May	5
June	6
July	7
August	8
September	9
October	10
November	11
December	12

Regional settings

Unless otherwise specified, the examples in this topic use the following date format: MM/DD/YYYY. The date format is specified in the `SET DateFormat` statement in your data load script. The default date formatting may be different in your system, due to your regional settings and other factors. You can change the formats in the examples below to suit your requirements. Or you can change the formats in your load script to match these examples.

Default regional settings in apps are based on the regional system settings of the computer or server where Qlik Sense is installed. If the Qlik Sense server you are accessing is set to Sweden, the Data load editor will use Swedish regional settings for dates, time, and currency. These regional format settings are not related to the language displayed in the Qlik Sense user interface. Qlik Sense will be displayed in the same language as the browser you are using.

Function examples

Example	Result
`yearstart('10/19/2001')`	Returns 01/01/2001 00:00:00.
`yearstart('10/19/2001',-1)`	Returns 01/01/2000 00:00:00.
`yearstart('10/19/2001',0,4)`	Returns 04/01/2001 00:00:00.

Example 1 – Basic example

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset containing a set of transactions between 2020 and 2022 is loaded into a table called 'Transactions'.
- The date field has been provided in the DateFormat system variable (MM/DD/YYYY) format.
- A preceding load statement which contains the following:
  - yearstart() function which is set as the year_start field.
  - Timestamp() function which is set as the year_start_timestamp field

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
       *,
       yearstart(date) as year_start,
       timestamp(yearstart(date)) as year_start_timestamp
       ;
Load
*
Inline
[
id,date,amount
8188,01/13/2020,37.23
8189,02/26/2020,17.17
8190,03/27/2020,88.27
8191,04/16/2020,57.42
8192,05/21/2020,53.80
8193,08/14/2020,82.06
8194,10/07/2020,40.39
8195,12/05/2020,87.21
8196,01/22/2021,95.93
8197,02/03/2021,45.89
8198,03/17/2021,36.23
8199,04/23/2021,25.66
8200,05/04/2021,82.77
8201,06/30/2021,69.98
8202,07/26/2021,76.11
8203,12/27/2021,25.12
8204,06/06/2022,46.23
8205,07/18/2022,84.21
8206,11/14/2022,96.24
8207,12/12/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- id
- date

- `year_start`
- `year_start_timestamp`

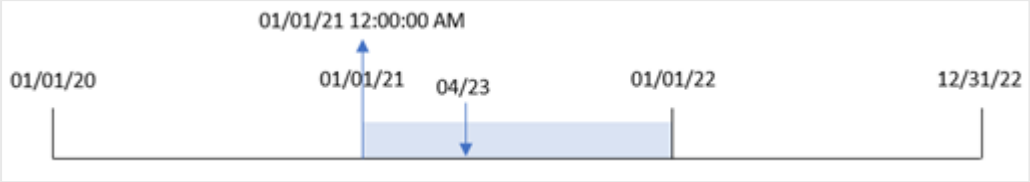
Results table

id	date	year_start	year_start_timestamp
8188	01/13/2020	01/01/2020	1/1/2020 12:00:00 AM
8189	02/26/2020	01/01/2020	1/1/2020 12:00:00 AM
8190	03/27/2020	01/01/2020	1/1/2020 12:00:00 AM
8191	04/16/2020	01/01/2020	1/1/2020 12:00:00 AM
8192	05/21/2020	01/01/2020	1/1/2020 12:00:00 AM
8193	08/14/2020	01/01/2020	1/1/2020 12:00:00 AM
8194	10/07/2020	01/01/2020	1/1/2020 12:00:00 AM
8195	12/05/2020	01/01/2020	1/1/2020 12:00:00 AM
8196	01/22/2021	01/01/2021	1/1/2021 12:00:00 AM
8197	02/03/2021	01/01/2021	1/1/2021 12:00:00 AM
8198	03/17/2021	01/01/2021	1/1/2021 12:00:00 AM
8199	04/23/2021	01/01/2021	1/1/2021 12:00:00 AM
8200	05/04/2021	01/01/2021	1/1/2021 12:00:00 AM
8201	06/30/2021	01/01/2021	1/1/2021 12:00:00 AM
8202	07/26/2021	01/01/2021	1/1/2021 12:00:00 AM
8203	12/27/2021	01/01/2021	1/1/2021 12:00:00 AM
8204	06/06/2022	01/01/2022	1/1/2022 12:00:00 AM
8205	07/18/2022	01/01/2022	1/1/2022 12:00:00 AM
8206	11/14/2022	01/01/2022	1/1/2022 12:00:00 AM
8207	12/12/2022	01/01/2022	1/1/2022 12:00:00 AM

The ‘`year_start`’ field is created in the preceding load statement by using the `yearstart()` function and passing the date field as the function’s argument.

The `yearstart()` function initially identifies which year the date value falls into and returns a timestamp for the first millisecond of that year.

Diagram of the `yearstart()` function and transaction 8199.



Transaction 8199 took place on April 23, 2021. The `yearstart()` function returns the first millisecond of that year, which is January 1 at 12:00:00 AM.

Example 2 – period_no

Load script and results

Overview

The same dataset and scenario as the first example are used.

However, in this example, the task is to create a field, '`previous_year_start`', that returns the start date timestamp of the year prior to the year in which a transaction took place.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
       *,
       yearstart(date,-1) as previous_year_start,
       timestamp(yearstart(date,-1)) as previous_year_start_timestamp
       ;
Load
*
Inline
[
id,date,amount
8188,01/13/2020,37.23
8189,02/26/2020,17.17
8190,03/27/2020,88.27
8191,04/16/2020,57.42
8192,05/21/2020,53.80
8193,08/14/2020,82.06
8194,10/07/2020,40.39
8195,12/05/2020,87.21
8196,01/22/2021,95.93
8197,02/03/2021,45.89
8198,03/17/2021,36.23
8199,04/23/2021,25.66
8200,05/04/2021,82.77
8201,06/30/2021,69.98
8202,07/26/2021,76.11
8203,12/27/2021,25.12
```

```
8204,06/06/2022,46.23
8205,07/18/2022,84.21
8206,11/14/2022,96.24
8207,12/12/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

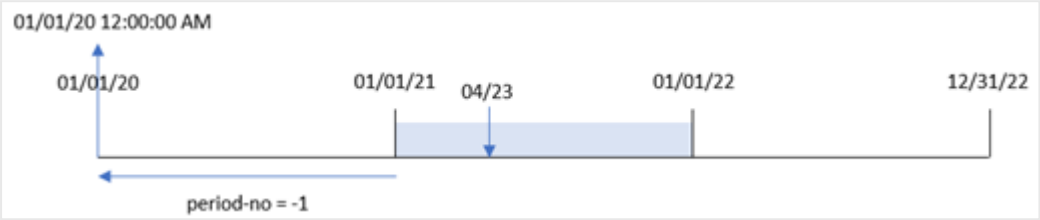
- `id`
- `date`
- `previous_year_start`
- `previous_ year_start_timestamp`

Results table

id	date	previous_year_start	previous_year_start_timestamp
8188	01/13/2020	01/01/2019	1/1/2019 12:00:00 AM
8189	02/26/2020	01/01/2019	1/1/2019 12:00:00 AM
8190	03/27/2020	01/01/2019	1/1/2019 12:00:00 AM
8191	04/16/2020	01/01/2019	1/1/2019 12:00:00 AM
8192	05/21/2020	01/01/2019	1/1/2019 12:00:00 AM
8193	08/14/2020	01/01/2019	1/1/2019 12:00:00 AM
8194	10/07/2020	01/01/2019	1/1/2019 12:00:00 AM
8195	12/05/2020	01/01/2019	1/1/2019 12:00:00 AM
8196	01/22/2021	01/01/2020	1/1/2020 12:00:00 AM
8197	02/03/2021	01/01/2020	1/1/2020 12:00:00 AM
8198	03/17/2021	01/01/2020	1/1/2020 12:00:00 AM
8199	04/23/2021	01/01/2020	1/1/2020 12:00:00 AM
8200	05/04/2021	01/01/2020	1/1/2020 12:00:00 AM
8201	06/30/2021	01/01/2020	1/1/2020 12:00:00 AM
8202	07/26/2021	01/01/2020	1/1/2020 12:00:00 AM
8203	12/27/2021	01/01/2020	1/1/2020 12:00:00 AM
8204	06/06/2022	01/01/2021	1/1/2021 12:00:00 AM
8205	07/18/2022	01/01/2021	1/1/2021 12:00:00 AM
8206	11/14/2022	01/01/2021	1/1/2021 12:00:00 AM
8207	12/12/2022	01/01/2021	1/1/2021 12:00:00 AM

In this instance, because a `period_no` of `-1` is used as the offset argument in the `yearstart()` function, the function first identifies the year that the transactions take place in. It then looks one year prior and identifies the first millisecond of that year.

Diagram of the `yearstart()` function with a `period_no` of -1.



Transaction 8199 took place on April 23, 2021. The `yearstart()` function returns the first millisecond of the prior year, January 1, 2020 at 12:00:00 AM, for the '`previous_year_start`' field.

Example 3 – first_month_of_year

Load script and results

Overview

The same dataset and scenario as the first example are used.

However, in this example, the company policy is for the year to begin from April 1.

Load script

```
SET DateFormat='MM/DD/YYYY';

Transactions:
    Load
       *,
       yearstart(date,0,4) as year_start,
       timestamp(yearstart(date,0,4)) as year_start_timestamp
       ;
Load
*
Inline
[
id,date,amount
8188,01/13/2020,37.23
8189,02/26/2020,17.17
8190,03/27/2020,88.27
8191,04/16/2020,57.42
8192,05/21/2020,53.80
8193,08/14/2020,82.06
8194,10/07/2020,40.39
8195,12/05/2020,87.21
8196,01/22/2021,95.93
8197,02/03/2021,45.89
8198,03/17/2021,36.23
8199,04/23/2021,25.66
```

```
8200,05/04/2021,82.77
8201,06/30/2021,69.98
8202,07/26/2021,76.11
8203,12/27/2021,25.12
8204,06/06/2022,46.23
8205,07/18/2022,84.21
8206,11/14/2022,96.24
8207,12/12/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `id`
- `date`
- `year_start`
- `year_start_timestamp`

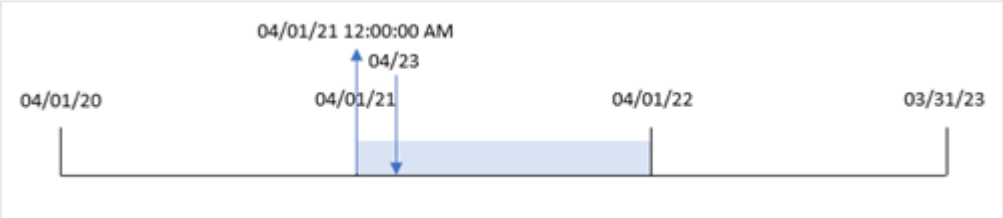
Results table

id	date	year_start	year_start_timestamp
8188	01/13/2020	04/01/2019	4/1/2019 12:00:00 AM
8189	02/26/2020	04/01/2019	4/1/2019 12:00:00 AM
8190	03/27/2020	04/01/2019	4/1/2019 12:00:00 AM
8191	04/16/2020	04/01/2020	4/1/2020 12:00:00 AM
8192	05/21/2020	04/01/2020	4/1/2020 12:00:00 AM
8193	08/14/2020	04/01/2020	4/1/2020 12:00:00 AM
8194	10/07/2020	04/01/2020	4/1/2020 12:00:00 AM
8195	12/05/2020	04/01/2020	4/1/2020 12:00:00 AM
8196	01/22/2021	04/01/2020	4/1/2020 12:00:00 AM
8197	02/03/2021	04/01/2020	4/1/2020 12:00:00 AM
8198	03/17/2021	04/01/2020	4/1/2020 12:00:00 AM
8199	04/23/2021	04/01/2021	4/1/2021 12:00:00 AM
8200	05/04/2021	04/01/2021	4/1/2021 12:00:00 AM
8201	06/30/2021	04/01/2021	4/1/2021 12:00:00 AM
8202	07/26/2021	04/01/2021	4/1/2021 12:00:00 AM
8203	12/27/2021	04/01/2021	4/1/2021 12:00:00 AM
8204	06/06/2022	04/01/2022	4/1/2022 12:00:00 AM

id	date	year_start	year_start_timestamp
8205	07/18/2022	04/01/2022	4/1/2022 12:00:00 AM
8206	11/14/2022	04/01/2022	4/1/2022 12:00:00 AM
8207	12/12/2022	04/01/2022	4/1/2022 12:00:00 AM

In this instance, because the `first_month_of_year` argument of 4 is used in the `yearstart()` function, it sets the first day of the year to April 1, and the last day of the year to March 31.

Diagram of the `yearstart()` function with the first month set as April.



Transaction 8199 took place on April 23, 2021. Because the `yearstart()` function sets the start of the year to April 1 and returns it as the '`year_start`' value for the transaction.

Example 4 – Chart object example

Load script and chart expression

Overview

The same dataset and scenario as the first example are used.

However, in this example, the dataset is unchanged and loaded into the application. The calculation that returns the start date timestamp of the year in which a transaction took place is created as a measure in a chart object of the application.

Load script

```
Transactions:
Load
*
Inline
[
id,date,amount
8188,01/13/2020,37.23
8189,02/26/2020,17.17
8190,03/27/2020,88.27
8191,04/16/2020,57.42
8192,05/21/2020,53.80
8193,08/14/2020,82.06
8194,10/07/2020,40.39
8195,12/05/2020,87.21
8196,01/22/2021,95.93
```

```
8197,02/03/2021,45.89
8198,03/17/2021,36.23
8199,04/23/2021,25.66
8200,05/04/2021,82.77
8201,06/30/2021,69.98
8202,07/26/2021,76.11
8203,12/27/2021,25.12
8204,06/06/2022,46.23
8205,07/18/2022,84.21
8206,11/14/2022,96.24
8207,12/12/2022,67.67
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `id`
- `date`

To calculate in which year a transaction took place, create the following measures:

- `=yearstart(date)`
- `=timestamp(yearstart(date))`

Results table

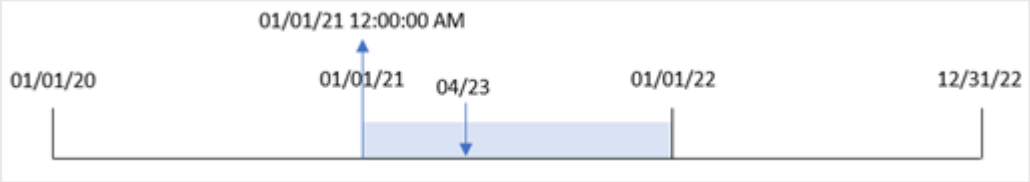
id	date	=yearstart(date)	=timestamp(yearstart(date))
8188	06/06/2022	01/01/2022	1/1/2022 12:00:00 AM
8189	07/18/2022	01/01/2022	1/1/2022 12:00:00 AM
8190	11/14/2022	01/01/2022	1/1/2022 12:00:00 AM
8191	12/12/2022	01/01/2022	1/1/2022 12:00:00 AM
8192	01/22/2021	01/01/2021	1/1/2021 12:00:00 AM
8193	02/03/2021	01/01/2021	1/1/2021 12:00:00 AM
8194	03/17/2021	01/01/2021	1/1/2021 12:00:00 AM
8195	04/23/2021	01/01/2021	1/1/2021 12:00:00 AM
8196	05/04/2021	01/01/2021	1/1/2021 12:00:00 AM
8197	06/30/2021	01/01/2021	1/1/2021 12:00:00 AM
8198	07/26/2021	01/01/2021	1/1/2021 12:00:00 AM
8199	12/27/2021	01/01/2021	1/1/2021 12:00:00 AM
8200	01/13/2020	01/01/2020	1/1/2020 12:00:00 AM
8201	02/26/2020	01/01/2020	1/1/2020 12:00:00 AM

id	date	=yearstart(date)	=timestamp(yearstart(date))
8202	03/27/2020	01/01/2020	1/1/2020 12:00:00 AM
8203	04/16/2020	01/01/2020	1/1/2020 12:00:00 AM
8204	05/21/2020	01/01/2020	1/1/2020 12:00:00 AM
8205	08/14/2020	01/01/2020	1/1/2020 12:00:00 AM
8206	10/07/2020	01/01/2020	1/1/2020 12:00:00 AM
8207	12/05/2020	01/01/2020	1/1/2020 12:00:00 AM

The ‘`start_of_year`’ measure is created in the chart object by using the `yearstart()` function and passing the date field as the function’s argument.

The `yearstart()` function initially identifies which year the date value falls into and returns a timestamp for the first millisecond of that year.

Diagram of the`yearstart()` function and transaction 8199.



Transaction 8199 took place on April 23, 2021. The `yearstart()` function returns the first millisecond of that year, which is January 1 at 12:00:00 AM.

Example 5 – Scenario

Load script and results

Overview

Open the Data load editor and add the load script below to a new tab.

The load script contains:

- A dataset is loaded into a table called ‘`Loans`’. The table contains the following fields:
  - Loan IDs.
  - The balance at the beginning of the year.
  - The simple interest rate charged on each loan per annum.

The end user would like a chart object that displays, by loan id, the current interest that has been accrued on each loan in the year to date.

Load script

```
Loans:
Load
*
Inline
[
loan_id,start_balance,rate
8188,$10000.00,0.024
8189,$15000.00,0.057
8190,$17500.00,0.024
8191,$21000.00,0.034
8192,$90000.00,0.084
];
```

Results

Load the data and open a sheet. Create a new table and add these fields as dimensions:

- `loan_id`
- `start_balance`

To calculate the accumulated interest, create the following measure:

`=start_balance*(rate*(today(1)-yearstart(today(1)))/365)`
Set the measure's **Number Formatting** to **Money**.

Results table

loan_id	start_balance	=start_balance*(rate*(today(1)-yearstart(today(1)))/365)
8188	$10000.00	$39.73
8189	$15000.00	$339.66
8190	$17500.00	$166.85
8191	$21000.00	$283.64
8192	$90000.00	$3003.29

The `yearstart()` function, using today's date as its only argument, returns the start date of the current year. By subtracting that result from the current date, the expression returns the number of days that have elapsed so far this year.

This value is then multiplied by the interest rate and divided by 365 to return the effective interest rate for the period. The effective interest rate for the period is then multiplied by the starting balance of the loan to return the interest that has been accrued so far this year.

yeartodate

This function finds if the input timestamp falls within the year of the date the script was last loaded, and returns True if it does, False if it does not.

Syntax:

```
YearToDate(timestamp[ , yearoffset [ , firstmonth [ , todaydate] ] ])
```

Return data type: Boolean

If none of the optional parameters are used, the year to date means any date within one calendar year from January 1 up to and including the date of the last script execution.

Arguments:

Arguments

Argument	Description
timestamp	The timestamp to evaluate, for example '2012-10-12'.
yearoffset	By specifying a **yearoffset**, **yeartodate** returns True for the same period in another year. A negative **yearoffset** indicates a previous year, a positive offset a future year. The most recent year-to-date is achieved by specifying yearoffset = -1. If omitted, 0 is assumed.
firstmonth	By specifying a **firstmonth** between 1 and 12 (1 if omitted), the beginning of the year may be moved forward to the first day of any month. For example, if you want to work with a fiscal year beginning on May 1, specify **firstmonth** = 5.
todaydate	By specifying a **todaydate** (timestamp of the last script execution if omitted) it is possible to move the day used as the upper boundary of the period.

Examples and results:
The following examples assume last reload time = 2011-11-18

Script examples

Example	Result
`yeartodate( '2010-11-18')`	returns False
`yeartodate( '2011-02-01')`	returns True
`yeartodate( '2011-11-18')`	returns True
`yeartodate( '2011-11-19')`	returns False
`yeartodate( '2011-11-19', 0, 1, '2011-12-31')`	returns True
`yeartodate( '2010-11-18', -1)`	returns True
`yeartodate( '2011-11-18', -1)`	returns False
`yeartodate( '2011-04-30', 0, 5)`	returns False
`yeartodate( '2011-05-01', 0, 5)`	returns True

5.8 Exponential and logarithmic functions

This section describes functions related to exponential and logarithmic calculations. All functions can be used in both the data load script and in chart expressions.

In the functions below, the parameters are expressions where **x** and **y** should be interpreted as real valued numbers.

exp

The natural exponential function, e^x, using the natural logarithm **e** as base. The result is a positive number.

```
exp(x )
```

Examples and results:

`exp(3)` returns 20.085.

log

The natural logarithm of **x**. The function is only defined if **x**> 0. The result is a number.

```
log(x )
```

Examples and results:

`log(3)` returns 1.0986

log10

The common logarithm (base 10) of **x**. The function is only defined if **x**> 0. The result is a number.

```
log10(x )
```

Examples and results:

`log10(3)` returns 0.4771

pow

Returns **x** to the power of **y**. The result is a number.

```
pow(x,y )
```

Examples and results:

`pow(3, 3)` returns 27

sqr

x squared (**x** to the power of 2). The result is a number.

```
sqr (x )
```

Examples and results:

`sqr(3)` returns 9

sqrt
Square root of **x**. The function is only defined if **x** >= 0. The result is a positive number.

```
sqrt(x )
```

Examples and results:

`sqrt(3)` returns 1.732

5.9 Field functions

These functions can only be used in chart expressions.

Field functions either return integers or strings identifying different aspects of field selections.

Count functions

GetAlternativeCount
GetAlternativeCount()is used to find the number of alternative (light gray) values in the identified field.

```
GetAlternativeCount - chart function (field_name)
```

GetExcludedCount
GetExcludedCount() finds the number of excluded distinct values in the identified field. Excluded values include alternative (light gray), excluded (dark gray), and selected excluded (dark gray with check mark) fields.

```
GetExcludedCount - chart function (page 770)(field_name)
```

GetNotSelectedCount
This chart function returns the number of not-selected values in the field named **fieldname**. The field must be in and-mode for this function to be relevant.

```
GetNotSelectedCount - chart function(fieldname [, includeexcluded=false])
```

GetPossibleCount
GetPossibleCount() is used to find the number of possible values in the identified field. If the identified field includes selections, then the selected (green) fields are counted. Otherwise associated (white) values are counted.

```
GetPossibleCount - chart function(field_name)
```

GetSelectedCount
GetSelectedCount() finds the number of selected (green) values in a field.

```
GetSelectedCount - chart function (field_name [, include_excluded])
```

Field and selection functions

GetCurrentSelections

GetCurrentSelections() returns a list of the current selections in the app. If the selections are instead made using a search string in a search box, **GetCurrentSelections()** returns the search string.

```
GetCurrentSelections - chart function([record_sep [,tag_sep [,value_sep
[,max_values]]]])
```

GetFieldSelections

GetFieldSelections() returns a **string** with the current selections in a field.

```
GetFieldSelections - chart function ( field_name [, value_sep [, max_
values]])
```

GetObjectDimension

GetObjectDimension() returns the name of the dimension. **Index** is an optional integer denoting the dimension that should be returned.

```
GetObjectDimension - chart function ([index])
```

GetObjectField

GetObjectField() returns the name of the dimension. **Index** is an optional integer denoting the dimension that should be returned.

```
GetObjectField - chart function ([index])
```

GetObjectMeasure

GetObjectMeasure() returns the name of the measure. **Index** is an optional integer denoting the measure that should be returned.

```
GetObjectMeasure - chart function ([index])
```

GetAlternativeCount - chart function

GetAlternativeCount()is used to find the number of alternative (light gray) values in the identified field.

Syntax:

```
GetAlternativeCount (field_name)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
field_name	The field containing the range of data to be measured.

Examples and results:

The following example uses the **First name** field loaded to a filter pane.

Examples and results

Examples	Results
Given that **John** is selected in **First name**. `GetAlternativeCount ([First name])`	4 as there are 4 unique and excluded (gray) values in **First name**.
Given that **John** and **Peter** are selected. `GetAlternativeCount ([First name])`	3 as there are 3 unique and excluded (gray) values in **First name**.
Given that no values are selected in **First name**. `GetAlternativeCount ([First name])`	0 as there are no selections.

Data used in example:

```
Names:
LOAD * inline [
First name|Last name|Initials|Has cellphone
John|Anderson|JA|Yes
Sue|Brown|SB|Yes
Mark|Carr|MC|No
Peter|Devonshire|PD|No
Jane|Elliot|JE|Yes
Peter|Franc|PF|Yes ] (delimiter is '|');
```

GetCurrentSelections - chart function

GetCurrentSelections() returns a list of the current selections in the app. If the selections are instead made using a search string in a search box, **GetCurrentSelections()** returns the search string.

If options are used, you will need to specify record_sep. To specify a new line, set **record_sep** to **chr(13)&chr(10)**.

If all but two, or all but one, values, are selected, the format 'NOT x,y' or 'NOT y' will be used respectively. If you select all values and the count of all values is greater than max_values, the text ALL will be returned.

Syntax:

```
GetCurrentSelections ([record_sep [, tag_sep [, value_sep [, max_values [,
state_name]]]]])
```

Return data type: string

Arguments:

Arguments

Arguments	Description
record_sep	Separator to be put between field records. The default is <CR><LF> meaning a new line.
tag_sep	Separator to be put between the field name tag and the field values. The default is ': '.
value_sep	The separator to be put between field values. The default is ', '.
max_values	The maximum number of field values to be individually listed. When a larger number of values is selected, the format 'x of y values' will be used instead. The default is 6.
state_name	The name of an alternate state that has been chosen for the specific visualization. If the **state_name** argument is used, only the selections associated with the specified state name are taken into account.

Examples and results:

The following example uses two fields loaded to different filter panes, one for **First name** name and one for **Initials**.

Examples and results

Examples	Results
Given that **John** is selected in **First name**. `GetCurrentSelections ()`	'First name: John'
Given that **John** and **Peter** are selected in **First name**. `GetCurrentSelections ()`	'First name: John, Peter'
Given that **John** and **Peter** are selected in **First name** and **JA** is selected in **Initials**. `GetCurrentSelections ()`	'First name: John, Peter Initials: JA'
Given that **John** is selected in **First name** and **JA** is selected in **Initials**. `GetCurrentSelections ( chr(13)&chr(10) , ' = ' )`	'First name = John Initials = JA'
Given that you have selected all names except Sue in **First name** and no selections in **Initials**. `GetCurrentSelections (chr(13)&chr(10), '=', ',' ,3)`	'First name=NOT Sue'

Data used in example:

```
Names:
LOAD * inline [
First name|Last name|Initials|Has cellphone
```

```
John|Anderson|JA|Yes
Sue|Brown|SB|Yes
Mark|Carr|MC|No
Peter|Devonshire|PD|No
Jane|Elliot|JE|Yes
Peter|Franc|PF|Yes ] (delimiter is '|');
```

GetExcludedCount - chart function

GetExcludedCount() finds the number of excluded distinct values in the identified field. Excluded values include alternative (light gray), excluded (dark gray), and selected excluded (dark gray with check mark) fields.

Syntax:

```
GetExcludedCount (field_name)
```

Return data type: string

Arguments:

Arguments

Arguments	Description
field_name	The field containing the range of data to be measured.

Examples and results:

The following example uses three fields loaded to different filter panes, one for **First name** , one for **Last name**, and one for **Initials**.

Examples and results

Examples	Results
If no values are selected in **First name**.	`GetExcludedCount (Initials) = 0` There are no selections.
If **John** is selected in **First name**.	`GetExcludedCount (Initials) = 5` There are 5 excluded values in **Initials** with dark gray color. The sixth cell (JA) will be white as it is associated with the selection John in **First name**.
If **John** and **Peter** are selected.	`GetExcludedCount (Initials) = 3` John is associated with 1 value and Peter is associated with 2 values, in **Initials**.
If **John** and **Peter** are selected in **First name**, and then **Franc** is selected in **Last name**.	`GetExcludedCount ([First name]) = 4` There are 4 excluded values in **First name** with dark gray color. **GetExcludedCount()** evaluates for fields with excluded values, including alternative and selected excluded fields.

Examples	Results
If **John** and **Peter** are selected in **First name**, and then **Franc** and **Anderson** are selected in **Last name**.	`GetExcludedCount (Initials) = 4` There are 4 excluded values in **Initials** with dark gray color. The other two cells (JA and PF) will be white as they associated with the selections John and Peter in **First name**.
If **John** and **Peter** are selected in **First name**, and then **Franc** and **Anderson** are selected in **Last name**.	`GetExcludedCount ([Last name]) = 4` There are 4 excluded values in **Initials**. Devonshire has light gray color while Brown, Carr, and Elliot have dark gray color.

Data used in example:

```
Names:
LOAD * inline [
First name|Last name|Initials|Has cellphone
John|Anderson|JA|Yes
Sue|Brown|SB|Yes
Mark|Carr|MC|No
Peter|Devonshire|PD|No
Jane|Elliot|JE|Yes
Peter|Franc|PF|Yes ] (delimiter is '|');
```

GetFieldSelections - chart function

GetFieldSelections() returns a **string** with the current selections in a field.

If all but two, or all but one of the values are selected, the format 'NOT x,y' or 'NOT y' will be used respectively. If you select all values and the count of all values is greater than max_values, the text ALL will be returned.

Syntax:

```
GetFieldSelections ( field_name [, value_sep [, max_values [, state_name]]])
```

Return data type: string

Return string formats

Format	Description
'a, b, c'	If the number of selected values is max_values or less, the string returned is a list of the selected values. The values are separated with value_sep as delimiter.
'NOT a, b, c'	If the number of non-selected values is max_values or less, the string returned is a list of the non-selected values with NOT as a prefix. The values are separated with value_sep as delimiter.

Format	Description
'x of y'	x = the number of selected values y = the total number of values This is returned when max_values < x < (y - max_values).
'ALL'	Returned if all values are selected.
'-'	Returned if no value is selected.
<search string>	If you have selected using search, the search string is returned.

Arguments:

Arguments

Arguments	Description
field_name	The field containing the range of data to be measured.
value_sep	The separator to be put between field values. The default is ', '.
max_values	The maximum number of field values to be individually listed. When a larger number of values is selected, the format 'x of y values' will be used instead. The default is 6.
state_name	The name of an alternate state that has been chosen for the specific visualization. If the **state_name** argument is used, only the selections associated with the specified state name are taken into account.

Examples and results:

The following example uses the **First name** field loaded to a filter pane.

Examples and results

Examples	Results
Given that **John** is selected in **First name**. `GetFieldSelections ([First name])`	'John'
Given that **John** and **Peter** are selected. `GetFieldSelections ([First name])`	'John,Peter'

Examples	Results
Given that **John** and **Peter** are selected. `GetFieldSelections ([First name],'; ')`	'John; Peter'
Given that **John**, **Sue**, **Mark** are selected in **First name**. `GetFieldSelections ([First name],';',2)`	'NOT Jane;Peter', because the value 2 is stated as the value of the max_ values argument. Otherwise, the result would have been John; Sue; Mark.

Data used in example:

```
Names:
LOAD * inline [
First name|Last name|Initials|Has cellphone
John|Anderson|JA|Yes
Sue|Brown|SB|Yes
Mark|Carr|MC|No
Peter|Devonshire|PD|No
Jane|Elliot|JE|Yes
Peter|Franc|PF|Yes ] (delimiter is '|');
```

GetNotSelectedCount - chart function

This chart function returns the number of not-selected values in the field named **fieldname**. The field must be in and-mode for this function to be relevant.

Syntax:

```
GetNotSelectedCount(fieldname [, includeexcluded=false])
```

Arguments:

Arguments

Argument	Description
fieldname	The name of the field to be evaluated.
includeexcluded	If **includeexcluded** is stated as True, the count will include selected values which are excluded by selections in another field.

Examples:

```
GetNotSelectedCount( Country )
GetNotSelectedCount( Country, true )
```

GetObjectDimension - chart function

GetObjectDimension() returns the name of the dimension. **Index** is an optional integer denoting the dimension that should be returned.

You cannot use this function in a chart in the following locations: title, subtitle, footer, reference line expression.

You cannot reference the name of a dimension or measure in another object using the Object ID.

Syntax:

```
GetObjectDimension ([index])
```

Example:

```
GetObjectDimension(1)
```

Example: Chart expression

Qlik Sense table showing examples of the GetObjectDimension function in a chart expression

transaction_date	customer_id	transaction_quantity	=GetObjectDimension ()	=GetObjectDimension (0)	=GetObjectDimension (1)
2018/08/30	049681	13	transaction_date	transaction_date	customer_id
2018/08/30	203521	6	transaction_date	transaction_date	customer_id
2018/08/30	203521	21	transaction_date	transaction_date	customer_id

If you want to return the name of a measure, use the **GetObjectMeasure** function instead.

GetObjectField - chart function

GetObjectField() returns the name of the dimension. **Index** is an optional integer denoting the dimension that should be returned.

You cannot use this function in a chart in the following locations: title, subtitle, footer, reference line expression.

You cannot reference the name of a dimension or measure in another object using the Object ID.

Syntax:

```
GetObjectField ([index])
```

Example:

```
GetObjectField(1)
```

Example: Chart expression

Qlik Sense table showing examples of the GetObjectField function in a chart expression.

transaction_date	customer_id	transaction_quantity	=GetObjectField ()	=GetObjectField (0)	=GetObjectField (1)
2018/08/30	049681	13	transaction_date	transaction_date	customer_id
2018/08/30	203521	6	transaction_date	transaction_date	customer_id
2018/08/30	203521	21	transaction_date	transaction_date	customer_id

If you want to return the name of a measure, use the **GetObjectMeasure** function instead.

GetObjectMeasure - chart function

GetObjectMeasure() returns the name of the measure. **Index** is an optional integer denoting the measure that should be returned.

You cannot use this function in a chart in the following locations: title, subtitle, footer, reference line expression.

You cannot reference the name of a dimension or measure in another object using the Object ID.

Syntax:

```
GetObjectMeasure ([index])
```

Example:

```
GetObjectMeasure(1)
```

Example: Chart expression

Qlik Sense table showing examples of the GetObjectMeasure function in a chart expression

customer_id	sum (transaction_quantity)	Avg (transaction_quantity)	=GetObjectMeasure ()	=GetObjectMeasure(0)	=GetObjectMeasure(1)
49681	13	13	sum(transaction_quantity)	sum(transaction_quantity)	Avg(transaction_quantity)
203521	27	13.5	sum(transaction_quantity)	sum(transaction_quantity)	Avg(transaction_quantity)

If you want to return the name of a dimension, use the **GetObjectField** function instead.

GetPossibleCount - chart function

GetPossibleCount() is used to find the number of possible values in the identified field. If the identified field includes selections, then the selected (green) fields are counted. Otherwise associated (white) values are counted. .

For fields with selections, **GetPossibleCount()** returns the number of selected (green) fields.

Return data type: integer

Syntax:

```
GetPossibleCount (field_name)
```

Arguments:

Arguments

Arguments	Description
field_name	The field containing the range of data to be measured.

Examples and results:

The following example uses two fields loaded to different filter panes, one for **First name** name and one for **Initials**.

Examples and results

Examples	Results
Given that **John** is selected in **First name**. `GetPossibleCount ([Initials])`	1 as there is 1 value in Initials associated with the selection, **John**, in **First name**.
Given that **John** is selected in **First name**. `GetPossibleCount ([First name])`	1 as there is 1 selection, **John**, in **First name**.
Given that **Peter**is selected in **First name**. `GetPossibleCount ([Initials])`	2 as Peter is associated with 2 values in **Initials**.
Given that no values are selected in **First name**. `GetPossibleCount ([First name])`	5 as there are no selections and there are 5 unique values in **First name**.
Given that no values are selected in **First name**. `GetPossibleCount ([Initials])`	6 as there are no selections and there are 6 unique values in **Initials**.

Data used in example:

```
Names:
LOAD * inline [
First name|Last name|Initials|Has cellphone
John|Anderson|JA|Yes
Sue|Brown|SB|Yes
Mark|Carr|MC|No
Peter|Devonshire|PD|No
Jane|Elliot|JE|Yes
Peter|Franc|PF|Yes ] (delimiter is '|');
```

GetSelectedCount - chart function

GetSelectedCount() finds the number of selected (green) values in a field.

Syntax:

```
GetSelectedCount (field_name [, include_excluded [, state_name]])
```

Return data type: integer

Arguments:

Arguments

Arguments	Description
field_name	The field containing the range of data to be measured.
include_ excluded	If set to **True()**, the count will include selected values, which are currently excluded by selections in other fields. If False or omitted, these values will not be included.
state_name	The name of an alternate state that has been chosen for the specific visualization. If the **state_name** argument is used, only the selections associated with the specified state name are taken into account.

Examples and results:

The following example uses three fields loaded to different filter panes, one for **First name** name, one for **Initials** and one for **Has cellphone**.

Examples and results

Examples	Results
Given that **John** is selected in **First name**. `GetSelectedCount ([First name])`	1 as one value is selected in **First name**.
Given that **John** is selected in **First name**. `GetSelectedCount ([Initials])`	0 as no values are selected in **Initials**.

Examples	Results
With no selections in .**First name**, select all values in **Initials** and after that select the value **Yes** in **Has cellphone**. `GetSelectedCount ([Initials], True())`	6. Although selections with **Initials**MC and PD have **Has cellphone** set to **No**, the result is still 6, because the argument include_excluded is set to True().

Data used in example:

```
Names:
LOAD * inline [
First name|Last name|Initials|Has cellphone
John|Anderson|JA|Yes
Sue|Brown|SB|Yes
Mark|Carr|MC|No
Peter|Devonshire|PD|No
Jane|Elliot|JE|Yes
Peter|Franc|PF|Yes ] (delimiter is '|');
```

5.10 File functions

The file functions (only available in script expressions) return information about the table file which is currently being read. These functions will return NULL for all data sources except table files (exception: **ConnectString( )**).

File functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Attribute

This script function returns the value of the meta tags of different media files as text. The following file formats are supported: MP3, WMA, WMV, PNG and JPG. If the file **filename** does not exist, is not a supported file format or does not contain a meta tag named **attributename**, NULL will be returned.

```
Attribute (filename, attributename)
```

ConnectString

The **ConnectString()** function returns the name of the active data connection for ODBC or OLE DB connections. The function returns an empty string if no **connect** statement has been executed, or after a **disconnect** statement.

```
ConnectString ()
```

FileBaseName

The **FileBaseName** function returns a string containing the name of the table file currently being read, without path or extension.

```
FileBaseName ()
```

FileDir

The **FileDir** function returns a string containing the path to the directory of the table file currently being read.

```
FileDir ()
```

FileExtension

The **FileExtension** function returns a string containing the extension of the table file currently being read.

```
FileExtension ()
```

FileName

The **FileName** function returns a string containing the name of the table file currently being read, without path but including the extension.

```
FileName ()
```

FilePath

The **FilePath** function returns a string containing the full path to the table file currently being read.

```
FilePath ()
```

FileSize

The **FileSize** function returns an integer containing the size in bytes of the file filename or, if no filename is specified, of the table file currently being read.

```
FileSize ()
```

FileTime

The **FileTime** function returns a timestamp in UTC format of the last modification of a specified file. If a file is not specified, the function returns a timestamp in UTC of the last modification of the currently read table file.

```
FileTime ([ filename ])
```

GetFolderPath

The **GetFolderPath** function returns the value of the Microsoft Windows *SHGetFolderPath* function. This function takes as input the name of a Microsoft Windows folder and returns the full path of the folder.

```
GetFolderPath ()
```

QvdCreateTime

This script function returns the XML-header time stamp from a QVD file, if any is present, otherwise it returns NULL. In the timestamp, time is provided in UTC.

```
QvdCreateTime (filename)
```

QvdFieldName

This script function returns the name of field number **fieldno** in a QVD file. If the field does not exist NULL is returned.

```
QvdFieldName (filename , fieldno)
```

QvdNoOfFields

This script function returns the number of fields in a QVD file.

```
QvdNoOfFields (filename)
```

QvdNoOfRecords

This script function returns the number of records currently in a QVD file.

```
QvdNoOfRecords (filename)
```

QvdTableName

This script function returns the name of the table stored in a QVD file.

```
QvdTableName (filename)
```

Attribute

This script function returns the value of the meta tags of different media files as text. The following file formats are supported: MP3, WMA, WMV, PNG and JPG. If the file **filename** does not exist, is not a supported file format or does not contain a meta tag named **attributename**, NULL will be returned.

Syntax:

```
Attribute(filename, attributename)
```

A large number of meta tags can be read. The examples in this topic show which tags can be read for the respective supported file types.

> *You can only read meta tags saved in the file according to the relevant specification, for example ID2v3 for MP3 files or EXIF for JPG files, not meta information saved in the **Windows File Explorer**.*

Arguments:

Arguments

Argument	Description
filename	The name of a media file including path, if needed, as a folder data connection. **Example:** ***'lib://Table Files/'*** In legacy scripting mode, the following path formats are also supported: • absolute **Example:** ***c:\data*** • relative to the Qlik Sense app working directory. **Example:** ***data***
attributename	The name of a meta tag.

The examples use the **GetFolderPath** function to find the paths to media files. As **GetFolderPath** is only supported in legacy mode, you need to replace the references to **GetFolderPath** with a lib:// data connection path when you use this function in standard mode or in Qlik Sense SaaS.

File system access restriction (page 1036)

Example 1: MP3 files

This script reads all possible MP3 meta tags in folder *MyMusic*.

```
// Script to read MP3 meta tags
for each vExt in 'mp3'
for each vFoundFile in filelist( GetFolderPath('MyMusic') & '\*.'& vExt )
FileList:
LOAD FileLongName,
    subfield(FileLongName,'\',-1) as FileShortName,
    num(FileSize(FileLongName),'# ### ### ###',',',' ') as FileSize,
    FileTime(FileLongName) as FileTime,
    // ID3v1.0 and ID3v1.1 tags
    Attribute(FileLongName, 'Title') as Title,
    Attribute(FileLongName, 'Artist') as Artist,
    Attribute(FileLongName, 'Album') as Album,
    Attribute(FileLongName, 'Year') as Year,
    Attribute(FileLongName, 'Comment') as Comment,
    Attribute(FileLongName, 'Track') as Track,
    Attribute(FileLongName, 'Genre') as Genre,
    // ID3v2.3 tags
    Attribute(FileLongName, 'AENC') as AENC, // Audio encryption
    Attribute(FileLongName, 'APIC') as APIC, // Attached picture
    Attribute(FileLongName, 'COMM') as COMM, // Comments
    Attribute(FileLongName, 'COMR') as COMR, // Commercial frame
    Attribute(FileLongName, 'ENCR') as ENCR, // Encryption method registration
```

```
    Attribute(FileLongName, 'EQUA') as EQUA, // Equalization
    Attribute(FileLongName, 'ETCO') as ETCO, // Event timing codes
    Attribute(FileLongName, 'GEOB') as GEOB, // General encapsulated object
    Attribute(FileLongName, 'GRID') as GRID, // Group identification registration
    Attribute(FileLongName, 'IPLS') as IPLS, // Involved people list
    Attribute(FileLongName, 'LINK') as LINK, // Linked information
    Attribute(FileLongName, 'MCDI') as MCDI, // Music CD identifier
    Attribute(FileLongName, 'MLLT') as MLLT, // MPEG location lookup table
    Attribute(FileLongName, 'OWNE') as OWNE, // Ownership frame
    Attribute(FileLongName, 'PRIV') as PRIV, // Private frame
    Attribute(FileLongName, 'PCNT') as PCNT, // Play counter
    Attribute(FileLongName, 'POPM') as POPM, // Popularimeter
    Attribute(FileLongName, 'POSS') as POSS, // Position synchronisation frame
    Attribute(FileLongName, 'RBUF') as RBUF, // Recommended buffer size
    Attribute(FileLongName, 'RVAD') as RVAD, // Relative volume adjustment
    Attribute(FileLongName, 'RVRB') as RVRB, // Reverb
    Attribute(FileLongName, 'SYLT') as SYLT, // Synchronized lyric/text
    Attribute(FileLongName, 'SYTC') as SYTC, // Synchronized tempo codes
    Attribute(FileLongName, 'TALB') as TALB, // Album/Movie/Show title
    Attribute(FileLongName, 'TBPM') as TBPM, // BPM (beats per minute)
    Attribute(FileLongName, 'TCOM') as TCOM, // Composer
    Attribute(FileLongName, 'TCON') as TCON, // Content type
    Attribute(FileLongName, 'TCOP') as TCOP, // Copyright message
    Attribute(FileLongName, 'TDAT') as TDAT, // Date
    Attribute(FileLongName, 'TDLY') as TDLY, // Playlist delay
    Attribute(FileLongName, 'TENC') as TENC, // Encoded by
    Attribute(FileLongName, 'TEXT') as TEXT, // Lyricist/Text writer
    Attribute(FileLongName, 'TFLT') as TFLT, // File type
    Attribute(FileLongName, 'TIME') as TIME, // Time
    Attribute(FileLongName, 'TIT1') as TIT1, // Content group description
    Attribute(FileLongName, 'TIT2') as TIT2, // Title/songname/content description
    Attribute(FileLongName, 'TIT3') as TIT3, // Subtitle/Description refinement
    Attribute(FileLongName, 'TKEY') as TKEY, // Initial key
    Attribute(FileLongName, 'TLAN') as TLAN, // Language(s)
    Attribute(FileLongName, 'TLEN') as TLEN, // Length
    Attribute(FileLongName, 'TMED') as TMED, // Media type
    Attribute(FileLongName, 'TOAL') as TOAL, // Original album/movie/show title
    Attribute(FileLongName, 'TOFN') as TOFN, // Original filename
    Attribute(FileLongName, 'TOLY') as TOLY, // Original lyricist(s)/text writer(s)
    Attribute(FileLongName, 'TOPE') as TOPE, // Original artist(s)/performer(s)
    Attribute(FileLongName, 'TORY') as TORY, // Original release year
    Attribute(FileLongName, 'TOWN') as TOWN, // File owner/licensee
    Attribute(FileLongName, 'TPE1') as TPE1, // Lead performer(s)/Soloist(s)
    Attribute(FileLongName, 'TPE2') as TPE2, // Band/orchestra/accompaniment
    Attribute(FileLongName, 'TPE3') as TPE3, // Conductor/performer refinement
    Attribute(FileLongName, 'TPE4') as TPE4, // Interpreted, remixed, or otherwise modified by
    Attribute(FileLongName, 'TPOS') as TPOS, // Part of a set
    Attribute(FileLongName, 'TPUB') as TPUB, // Publisher
    Attribute(FileLongName, 'TRCK') as TRCK, // Track number/Position in set
    Attribute(FileLongName, 'TRDA') as TRDA, // Recording dates
    Attribute(FileLongName, 'TRSN') as TRSN, // Internet radio station name
    Attribute(FileLongName, 'TRSO') as TRSO, // Internet radio station owner
    Attribute(FileLongName, 'TSIZ') as TSIZ, // Size
    Attribute(FileLongName, 'TSRC') as TSRC, // ISRC (international standard recording code)
    Attribute(FileLongName, 'TSSE') as TSSE, // Software/Hardware and settings used for
encoding
```

```
    Attribute(FileLongName, 'TYER') as TYER, // Year
    Attribute(FileLongName, 'TXXX') as TXXX, // User defined text information frame
    Attribute(FileLongName, 'UFID') as UFID, // Unique file identifier
    Attribute(FileLongName, 'USER') as USER, // Terms of use
    Attribute(FileLongName, 'USLT') as USLT, // Unsychronized lyric/text transcription
    Attribute(FileLongName, 'WCOM') as WCOM, // Commercial information
    Attribute(FileLongName, 'WCOP') as WCOP, // Copyright/Legal information
    Attribute(FileLongName, 'WOAF') as WOAF, // Official audio file webpage
    Attribute(FileLongName, 'WOAR') as WOAR, // Official artist/performer webpage
    Attribute(FileLongName, 'WOAS') as WOAS, // Official audio source webpage
    Attribute(FileLongName, 'WORS') as WORS, // Official internet radio station homepage
    Attribute(FileLongName, 'WPAY') as WPAY, // Payment
    Attribute(FileLongName, 'WPUB') as WPUB, // Publishers official webpage
    Attribute(FileLongName, 'WXXX') as WXXX; // User defined URL link frame
LOAD @1:n as FileLongName Inline "$(vFoundFile)" (fix, no labels);
Next vFoundFile
Next vExt
```

Example 2: JPEG

This script reads all possible EXIF meta tags from JPG files in folder *MyPictures*.

```
// Script to read Jpeg Exif meta tags
for each vExt in 'jpg', 'jpeg', 'jpe', 'jfif', 'jif', 'jfi'
for each vFoundFile in filelist( GetFolderPath('MyPictures') & '\*.'& vExt )
FileList:
LOAD FileLongName,
    subfield(FileLongName,'\',-1) as FileShortName,
    num(FileSize(FileLongName),'# ### ### ###',',',' ') as FileSize,
    FileTime(FileLongName) as FileTime,
    // ************   Exif Main (IFD0) Attributes   ************
    Attribute(FileLongName, 'ImageWidth') as ImageWidth,
    Attribute(FileLongName, 'ImageLength') as ImageLength,
    Attribute(FileLongName, 'BitsPerSample') as BitsPerSample,
    Attribute(FileLongName, 'Compression') as Compression,
    //  examples: 1=uncompressed, 2=CCITT, 3=CCITT 3, 4=CCITT 4,
    //5=LZW, 6=JPEG (old style), 7=JPEG, 8=Deflate, 32773=PackBits RLE,
    Attribute(FileLongName, 'PhotometricInterpretation') as PhotometricInterpretation,
    //  examples: 0=WhiteIsZero, 1=BlackIsZero, 2=RGB, 3=Palette, 5=CMYK, 6=YCbCr,
    Attribute(FileLongName, 'ImageDescription') as ImageDescription,
    Attribute(FileLongName, 'Make') as Make,
    Attribute(FileLongName, 'Model') as Model,
    Attribute(FileLongName, 'StripOffsets') as StripOffsets,
    Attribute(FileLongName, 'Orientation') as Orientation,
    //  examples: 1=TopLeft, 2=TopRight, 3=BottomRight, 4=BottomLeft,
    // 5=LeftTop, 6=RightTop, 7=RightBottom, 8=LeftBottom,
    Attribute(FileLongName, 'SamplesPerPixel') as SamplesPerPixel,
    Attribute(FileLongName, 'RowsPerStrip') as RowsPerStrip,
    Attribute(FileLongName, 'StripByteCounts') as StripByteCounts,
    Attribute(FileLongName, 'XResolution') as XResolution,
    Attribute(FileLongName, 'YResolution') as YResolution,
    Attribute(FileLongName, 'PlanarConfiguration') as PlanarConfiguration,
    //  examples: 1=chunky format, 2=planar format,
    Attribute(FileLongName, 'ResolutionUnit') as ResolutionUnit,
    //  examples: 1=none, 2=inches, 3=centimeters,
    Attribute(FileLongName, 'TransferFunction') as TransferFunction,
```

```
    Attribute(FileLongName, 'Software') as Software,
    Attribute(FileLongName, 'DateTime') as DateTime,
    Attribute(FileLongName, 'Artist') as Artist,
    Attribute(FileLongName, 'HostComputer') as HostComputer,
    Attribute(FileLongName, 'WhitePoint') as WhitePoint,
    Attribute(FileLongName, 'PrimaryChromaticities') as PrimaryChromaticities,
    Attribute(FileLongName, 'YCbCrCoefficients') as YCbCrCoefficients,
    Attribute(FileLongName, 'YCbCrSubSampling') as YCbCrSubSampling,
    Attribute(FileLongName, 'YCbCrPositioning') as YCbCrPositioning,
    //  examples: 1=centered, 2=co-sited,
    Attribute(FileLongName, 'ReferenceBlackWhite') as ReferenceBlackWhite,
    Attribute(FileLongName, 'Rating') as Rating,
    Attribute(FileLongName, 'RatingPercent') as RatingPercent,
    Attribute(FileLongName, 'ThumbnailFormat') as ThumbnailFormat,
    //  examples: 0=Raw Rgb, 1=Jpeg,
    Attribute(FileLongName, 'Copyright') as Copyright,
    Attribute(FileLongName, 'ExposureTime') as ExposureTime,
    Attribute(FileLongName, 'FNumber') as FNumber,
    Attribute(FileLongName, 'ExposureProgram') as ExposureProgram,
    //  examples: 0=Not defined, 1=Manual, 2=Normal program, 3=Aperture priority, 4=Shutter
priority,
    // 5=Creative program, 6=Action program, 7=Portrait mode, 8=Landscape mode, 9=Bulb,
    Attribute(FileLongName, 'ISOSpeedRatings') as ISOSpeedRatings,
    Attribute(FileLongName, 'TimeZoneOffset') as TimeZoneOffset,
    Attribute(FileLongName, 'SensitivityType') as SensitivityType,
    //  examples: 0=Unknown, 1=Standard output sensitivity (SOS), 2=Recommended exposure index
(REI),
    // 3=ISO speed, 4=Standard output sensitivity (SOS) and Recommended exposure index (REI),
    //5=Standard output sensitivity (SOS) and ISO Speed, 6=Recommended exposure index (REI)
and ISO Speed,
    // 7=Standard output sensitivity (SOS) and Recommended exposure index (REI) and ISO speed,
    Attribute(FileLongName, 'ExifVersion') as ExifVersion,
    Attribute(FileLongName, 'DateTimeOriginal') as DateTimeOriginal,
    Attribute(FileLongName, 'DateTimeDigitized') as DateTimeDigitized,
    Attribute(FileLongName, 'ComponentsConfiguration') as ComponentsConfiguration,
    //  examples: 1=Y, 2=Cb, 3=Cr, 4=R, 5=G, 6=B,
    Attribute(FileLongName, 'CompressedBitsPerPixel') as CompressedBitsPerPixel,
    Attribute(FileLongName, 'ShutterSpeedValue') as ShutterSpeedValue,
    Attribute(FileLongName, 'ApertureValue') as ApertureValue,
    Attribute(FileLongName, 'BrightnessValue') as BrightnessValue, //  examples: -1=Unknown,
    Attribute(FileLongName, 'ExposureBiasValue') as ExposureBiasValue,
    Attribute(FileLongName, 'MaxApertureValue') as MaxApertureValue,
    Attribute(FileLongName, 'SubjectDistance') as SubjectDistance,
    //  examples: 0=Unknown, -1=Infinity,
    Attribute(FileLongName, 'MeteringMode') as MeteringMode,
    //  examples: 0=Unknown, 1=Average, 2=CenterWeightedAverage, 3=Spot,
    // 4=MultiSpot, 5=Pattern, 6=Partial, 255=Other,
    Attribute(FileLongName, 'LightSource') as LightSource,
    //  examples: 0=Unknown, 1=Daylight, 2=Fluorescent, 3=Tungsten, 4=Flash, 9=Fine weather,
    // 10=Cloudy weather, 11=Shade, 12=Daylight fluorescent,
    // 13=Day white fluorescent, 14=Cool white fluorescent,
    // 15=White fluorescent, 17=Standard light A, 18=Standard light B, 19=Standard light C,
    // 20=D55, 21=D65, 22=D75, 23=D50, 24=ISO studio tungsten, 255=other light source,
    Attribute(FileLongName, 'Flash') as Flash,
    Attribute(FileLongName, 'FocalLength') as FocalLength,
    Attribute(FileLongName, 'SubjectArea') as SubjectArea,
    Attribute(FileLongName, 'MakerNote') as MakerNote,
```

```
    Attribute(FileLongName, 'UserComment') as UserComment,
    Attribute(FileLongName, 'SubSecTime') as SubSecTime,
    Attribute(FileLongName, 'SubsecTimeOriginal') as SubsecTimeOriginal,
    Attribute(FileLongName, 'SubsecTimeDigitized') as SubsecTimeDigitized,
    Attribute(FileLongName, 'XPTitle') as XPTitle,
    Attribute(FileLongName, 'XPComment') as XPComment,
    Attribute(FileLongName, 'XPAuthor') as XPAuthor,
    Attribute(FileLongName, 'XPKeywords') as XPKeywords,
    Attribute(FileLongName, 'XPSubject') as XPSubject,
    Attribute(FileLongName, 'FlashpixVersion') as FlashpixVersion,
    Attribute(FileLongName, 'ColorSpace') as ColorSpace, //  examples: 1=sRGB,
65535=Uncalibrated,
    Attribute(FileLongName, 'PixelXDimension') as PixelXDimension,
    Attribute(FileLongName, 'PixelYDimension') as PixelYDimension,
    Attribute(FileLongName, 'RelatedSoundFile') as RelatedSoundFile,
    Attribute(FileLongName, 'FocalPlaneXResolution') as FocalPlaneXResolution,
    Attribute(FileLongName, 'FocalPlaneYResolution') as FocalPlaneYResolution,
    Attribute(FileLongName, 'FocalPlaneResolutionUnit') as FocalPlaneResolutionUnit,
    //  examples: 1=None, 2=Inch, 3=Centimeter,
    Attribute(FileLongName, 'ExposureIndex') as ExposureIndex,
    Attribute(FileLongName, 'SensingMethod') as SensingMethod,
    //  examples: 1=Not defined, 2=One-chip color area sensor, 3=Two-chip color area sensor,
    // 4=Three-chip color area sensor, 5=Color sequential area sensor,
    // 7=Trilinear sensor, 8=Color sequential linear sensor,
    Attribute(FileLongName, 'FileSource') as FileSource,
    //  examples: 0=Other, 1=Scanner of transparent type,
    // 2=Scanner of reflex type, 3=Digital still camera,
    Attribute(FileLongName, 'SceneType') as SceneType,
    //  examples: 1=A directly photographed image,
    Attribute(FileLongName, 'CFAPattern') as CFAPattern,
    Attribute(FileLongName, 'CustomRendered') as CustomRendered,
    //  examples: 0=Normal process, 1=Custom process,
    Attribute(FileLongName, 'ExposureMode') as ExposureMode,
    //  examples: 0=Auto exposure, 1=Manual exposure, 2=Auto bracket,
    Attribute(FileLongName, 'WhiteBalance') as WhiteBalance,
    //  examples: 0=Auto white balance, 1=Manual white balance,
    Attribute(FileLongName, 'DigitalZoomRatio') as DigitalZoomRatio,
    Attribute(FileLongName, 'FocalLengthIn35mmFilm') as FocalLengthIn35mmFilm,
    Attribute(FileLongName, 'SceneCaptureType') as SceneCaptureType,
    //  examples: 0=Standard, 1=Landscape, 2=Portrait, 3=Night scene,
    Attribute(FileLongName, 'GainControl') as GainControl,
    //  examples: 0=None, 1=Low gain up, 2=High gain up, 3=Low gain down, 4=High gain down,
    Attribute(FileLongName, 'Contrast') as Contrast,
    //  examples: 0=Normal, 1=Soft, 2=Hard,
    Attribute(FileLongName, 'Saturation') as Saturation,
    //  examples: 0=Normal, 1=Low saturation, 2=High saturation,
    Attribute(FileLongName, 'Sharpness') as Sharpness,
    //  examples: 0=Normal, 1=Soft, 2=Hard,
    Attribute(FileLongName, 'SubjectDistanceRange') as SubjectDistanceRange,
    //  examples: 0=Unknown, 1=Macro, 2=Close view, 3=Distant view,
    Attribute(FileLongName, 'ImageUniqueID') as ImageUniqueID,
    Attribute(FileLongName, 'BodySerialNumber') as BodySerialNumber,
    Attribute(FileLongName, 'CMNT_GAMMA') as CMNT_GAMMA,
    Attribute(FileLongName, 'PrintImageMatching') as PrintImageMatching,
    Attribute(FileLongName, 'OffsetSchema') as OffsetSchema,
    // ************   Interoperability Attributes   ************
    Attribute(FileLongName, 'InteroperabilityIndex') as InteroperabilityIndex,
```

```
    Attribute(FileLongName, 'InteroperabilityVersion') as InteroperabilityVersion,
    Attribute(FileLongName, 'InteroperabilityRelatedImageFileFormat') as
InteroperabilityRelatedImageFileFormat,
    Attribute(FileLongName, 'InteroperabilityRelatedImageWidth') as
InteroperabilityRelatedImageWidth,
    Attribute(FileLongName, 'InteroperabilityRelatedImageLength') as
InteroperabilityRelatedImageLength,
    Attribute(FileLongName, 'InteroperabilityColorSpace') as InteroperabilityColorSpace,
    //  examples: 1=sRGB, 65535=Uncalibrated,
    Attribute(FileLongName, 'InteroperabilityPrintImageMatching') as
InteroperabilityPrintImageMatching,
    // ************   GPS Attributes   ************
    Attribute(FileLongName, 'GPSVersionID') as GPSVersionID,
    Attribute(FileLongName, 'GPSLatitudeRef') as GPSLatitudeRef,
    Attribute(FileLongName, 'GPSLatitude') as GPSLatitude,
    Attribute(FileLongName, 'GPSLongitudeRef') as GPSLongitudeRef,
    Attribute(FileLongName, 'GPSLongitude') as GPSLongitude,
    Attribute(FileLongName, 'GPSAltitudeRef') as GPSAltitudeRef,
    //  examples: 0=Above sea level, 1=Below sea level,
    Attribute(FileLongName, 'GPSAltitude') as GPSAltitude,
    Attribute(FileLongName, 'GPSTimeStamp') as GPSTimeStamp,
    Attribute(FileLongName, 'GPSSatellites') as GPSSatellites,
    Attribute(FileLongName, 'GPSStatus') as GPSStatus,
    Attribute(FileLongName, 'GPSMeasureMode') as GPSMeasureMode,
    Attribute(FileLongName, 'GPSDOP') as GPSDOP,
    Attribute(FileLongName, 'GPSSpeedRef') as GPSSpeedRef,
    Attribute(FileLongName, 'GPSSpeed') as GPSSpeed,
    Attribute(FileLongName, 'GPSTrackRef') as GPSTrackRef,
    Attribute(FileLongName, 'GPSTrack') as GPSTrack,
    Attribute(FileLongName, 'GPSImgDirectionRef') as GPSImgDirectionRef,
    Attribute(FileLongName, 'GPSImgDirection') as GPSImgDirection,
    Attribute(FileLongName, 'GPSMapDatum') as GPSMapDatum,
    Attribute(FileLongName, 'GPSDestLatitudeRef') as GPSDestLatitudeRef,
    Attribute(FileLongName, 'GPSDestLatitude') as GPSDestLatitude,
    Attribute(FileLongName, 'GPSDestLongitudeRef') as GPSDestLongitudeRef,
    Attribute(FileLongName, 'GPSDestLongitude') as GPSDestLongitude,
    Attribute(FileLongName, 'GPSDestBearingRef') as GPSDestBearingRef,
    Attribute(FileLongName, 'GPSDestBearing') as GPSDestBearing,
    Attribute(FileLongName, 'GPSDestDistanceRef') as GPSDestDistanceRef,
    Attribute(FileLongName, 'GPSDestDistance') as GPSDestDistance,
    Attribute(FileLongName, 'GPSProcessingMethod') as GPSProcessingMethod,
    Attribute(FileLongName, 'GPSAreaInformation') as GPSAreaInformation,
    Attribute(FileLongName, 'GPSDateStamp') as GPSDateStamp,
    Attribute(FileLongName, 'GPSDifferential') as GPSDifferential;
    //  examples: 0=No correction, 1=Differential correction,
LOAD @1:n as FileLongName Inline "$(vFoundFile)" (fix, no labels);
Next vFoundFile
Next vExt
```

Example 3: Windows media files

This script reads all possible WMA/WMV ASF meta tags in folder *MyMusic*.

```
/ Script to read WMA/WMV ASF meta tags
for each vExt in 'asf', 'wma', 'wmv'
for each vFoundFile in filelist( GetFolderPath('MyMusic') & '\*.'& vExt )
```

```
FileList:
LOAD FileLongName,
    subfield(FileLongName,'\',-1) as FileShortName,
    num(FileSize(FileLongName),'# ### ### ###',',',' ') as FileSize,
    FileTime(FileLongName) as FileTime,
    Attribute(FileLongName, 'Title') as Title,
    Attribute(FileLongName, 'Author') as Author,
    Attribute(FileLongName, 'Copyright') as Copyright,
    Attribute(FileLongName, 'Description') as Description,
    Attribute(FileLongName, 'Rating') as Rating,
    Attribute(FileLongName, 'PlayDuration') as PlayDuration,
    Attribute(FileLongName, 'MaximumBitrate') as MaximumBitrate,
    Attribute(FileLongName, 'WMFSDKVersion') as WMFSDKVersion,
    Attribute(FileLongName, 'WMFSDKNeeded') as WMFSDKNeeded,
    Attribute(FileLongName, 'IsVBR') as IsVBR,
    Attribute(FileLongName, 'ASFLeakyBucketPairs') as ASFLeakyBucketPairs,
    Attribute(FileLongName, 'PeakValue') as PeakValue,
    Attribute(FileLongName, 'AverageLevel') as AverageLevel;
LOAD @1:n as FileLongName Inline "$(vFoundFile)" (fix, no labels);
Next vFoundFile
Next vExt
```

Example 4: PNG

This script reads all possible PNG meta tags in folder *MyPictures*.

```
// Script to read PNG meta tags
for each vExt in 'png'
for each vFoundFile in filelist( GetFolderPath('MyPictures') & '\*.'& vExt )
FileList:
LOAD FileLongName,
    subfield(FileLongName,'\',-1) as FileShortName,
    num(FileSize(FileLongName),'# ### ### ###',',',' ') as FileSize,
    FileTime(FileLongName) as FileTime,
    Attribute(FileLongName, 'Comment') as Comment,
    Attribute(FileLongName, 'Creation Time') as Creation_Time,
    Attribute(FileLongName, 'Source') as Source,
    Attribute(FileLongName, 'Title') as Title,
    Attribute(FileLongName, 'Software') as Software,
    Attribute(FileLongName, 'Author') as Author,
    Attribute(FileLongName, 'Description') as Description,
    Attribute(FileLongName, 'Copyright') as Copyright;
LOAD @1:n as FileLongName Inline "$(vFoundFile)" (fix, no labels);
Next vFoundFile
Next vExt
```

ConnectString

The **ConnectString()** function returns the name of the active data connection for ODBC or OLE DB connections. The function returns an empty string if no **connect** statement has been executed, or after a **disconnect** statement.

Syntax:

```
ConnectString()
```

Examples and results:

Scripting examples

Example	Result

`LIB CONNECT TO 'Tutorial ODBC';`
`ConnectString:`
`Load ConnectString() as ConnectString AutoGenerate 1;`
	Returns 'Tutorial ODBC' in field ConnectString.
	
	This examples assumes that you have an available data connection called Tutorial ODBC.

FileBaseName

The **FileBaseName** function returns a string containing the name of the table file currently being read, without path or extension.

Syntax:

```
FileBaseName()
```

Examples and results:

Scripting examples

Example	Result
`LOAD *, filebasename( ) as X from C:\UserFiles\abc.txt`	Will return 'abc' in field X in each record read.

FileDir

The **FileDir** function returns a string containing the path to the directory of the table file currently being read.

Syntax:

```
FileDir()
```

This function supports only folder data connections in standard mode.

Examples and results:

Scripting examples

Example	Result
`Load *, filedir( ) as X from C:\UserFiles\abc.txt`	Will return 'C:\UserFiles' in field X in each record read.

FileExtension

The **FileExtension** function returns a string containing the extension of the table file currently being read.

Syntax:

```
FileExtension()
```

Examples and results:

Scripting examples

Example	Result
`LOAD *, FileExtension( ) as X from C:\UserFiles\abc.txt`	Will return 'txt' in field X in each record read.

FileName

The **FileName** function returns a string containing the name of the table file currently being read, without path but including the extension.

Syntax:

```
FileName()
```

Examples and results:

Scripting examples

Example	Result
`LOAD *, FileName( ) as X from C:\UserFiles\abc.txt`	Will return 'abc.txt' in field X in each record read.

FilePath

The **FilePath** function returns a string containing the full path to the table file currently being read.

Syntax:

```
FilePath()
```

This function supports only folder data connections in standard mode.

Examples and results:

Scripting examples

Example	Result
`Load *, FilePath( ) as X from C:\UserFiles\abc.txt`	Will return 'C:\UserFiles\abc.txt' in field X in each record read.

FileSize

The **FileSize** function returns an integer containing the size in bytes of the file filename or, if no filename is specified, of the table file currently being read.

Syntax:

```
FileSize([filename])
```

Arguments:

Arguments

Argument	Description
filename	The name of a file, if necessary including path, as a folder or web file data connection. If you don't specify a file name, the table file currently being read is used. **Example:** ***'lib://Table Files/'*** In legacy scripting mode, the following path formats are also supported: • absolute **Example:** ***c:\data*** • relative to the Qlik Sense app working directory. **Example:** ***data*** • URL address (HTTP or FTP), pointing to a location on the Internet or an intranet. **Example:** ***http://www.qlik.com***

Examples and results:

Scripting examples

Example	Result
`LOAD *, FileSize( ) as X from abc.txt;`	Will return the size of the specified file (abc.txt) as an integer in field X in each record read.
`FileSize( 'lib://DataFiles/xyz.xls' )`	Will return the size of the file xyz.xls.

FileTime

The **FileTime** function returns a timestamp in UTC format of the last modification of a specified file. If a file is not specified, the function returns a timestamp in UTC of the last modification of the currently read table file.

Syntax:

```
FileTime([ filename ])
```

Arguments:

Arguments

Argument	Description
filename	The name of a file, if necessary including path, as a folder or web file data connection. **Example:** ***'lib://Table Files/'*** In legacy scripting mode, the following path formats are also supported: • absolute **Example:** ***c:\data*** • relative to the Qlik Sense app working directory. **Example:** ***data*** • URL address (HTTP or FTP), pointing to a location on the Internet or an intranet. **Example:** ***http://www.qlik.com***

Examples and results:

Script examples

Example	Result
`LOAD *, FileTime( ) as X from abc.txt;`	Will return the timestamp of the last modification of the file (abc.txt) in field X in each record read.
`FileTime( 'xyz.xls' )`	Will return the timestamp of the last modification of the file xyz.xls.

GetFolderPath

The **GetFolderPath** function returns the value of the Microsoft Windows *SHGetFolderPath* function. This function takes as input the name of a Microsoft Windows folder and returns the full path of the folder.

This function is not supported in standard mode.

Syntax:

```
GetFolderPath(foldername)
```

Arguments:

Arguments

Argument	Description
foldername	Name of the Microsoft Windows folder. The folder name should not contain any space. Any space in the folder name seen in Windows Explorer should be removed from the folder name. Examples: *MyMusic* *MyDocuments*

Examples and results:

The goal of this example is to get the paths of the following Microsoft Windows folders: *MyMusic*, *MyPictures* and *Windows*. Add the example script to your app and reload it.

```
LOAD
 GetFolderPath('MyMusic') as MyMusic,
 GetFolderPath('MyPictures') as MyPictures,
 GetFolderPath('Windows') as Windows
AutoGenerate 1;
```

Once the app is reloaded, the fields *MyMusic*, *MyPictures* and *Windows* are added to the data model. Each field contains the path to the folder defined in input. For example:

- *C:\Users\smu\Music* for the folder *MyMusic*
- *C:\Users\smu\Pictures* for the folder *MyPictures*
- *C:\Windows* for the folder *Windows*

QvdCreateTime

This script function returns the XML-header time stamp from a QVD file, if any is present, otherwise it returns NULL. In the timestamp, time is provided in UTC.

Syntax:

```
QvdCreateTime(filename)
```

Arguments:

Arguments

Argument	Description
filename	The name of a QVD file, if necessary including path, as a folder or web data connection. **Example:** ***'lib://Table Files/'*** In legacy scripting mode, the following path formats are also supported: • absolute **Example:** ***c:\data*** • relative to the Qlik Sense app working directory. **Example:** ***data*** • URL address (HTTP or FTP), pointing to a location on the Internet or an intranet. **Example:** ***http://www.qlik.com***

Example:

```
QvdCreateTime('MyFile.qvd')
QvdCreateTime('C:\MyDir\MyFile.qvd')
QvdCreateTime('lib://DataFiles/MyFile.qvd')
```

QvdFieldName

This script function returns the name of field number **fieldno** in a QVD file. If the field does not exist NULL is returned.

Syntax:

```
QvdFieldName(filename , fieldno)
```

Arguments:

Arguments

Argument	Description
filename	The name of a QVD file, if necessary including path, as a folder or web data connection. **Example:** ***'lib://Table Files/'*** In legacy scripting mode, the following path formats are also supported: • absolute **Example:** ***c:\data*** • relative to the Qlik Sense app working directory. **Example:** ***data*** • URL address (HTTP or FTP), pointing to a location on the Internet or an intranet. **Example:** ***http://www.qlik.com***
fieldno	The number of the field within the table contained in the QVD file.

Examples:

```
QvdFieldName ('MyFile.qvd', 5)
QvdFieldName ('C:\MyDir\MyFile.qvd', 5)
QvdFieldName ('lib://DataFiles/MyFile.qvd', 5)
```

All three examples return the name of the fifth field of the table contained in the QVD file.

QvdNoOfFields

This script function returns the number of fields in a QVD file.

Syntax:

```
QvdNoOfFields(filename)
```

Arguments:

Arguments

Argument	Description
filename	The name of a QVD file, if necessary including path, as a folder or web data connection. **Example:** ***'lib://Table Files/'*** In legacy scripting mode, the following path formats are also supported: • absolute **Example:** ***c:\data*** • relative to the Qlik Sense app working directory. **Example:** ***data*** • URL address (HTTP or FTP), pointing to a location on the Internet or an intranet. **Example:** ***http://www.qlik.com***

Examples:

```
QvdNoOfFields ('MyFile.qvd')
QvdNoOfFields ('C:\MyDir\MyFile.qvd')
QvdNoOfFields ('lib://DataFiles/MyFile.qvd')
```

QvdNoOfRecords

Example: This script function returns the number of records currently in a QVD file.

Syntax:

```
QvdNoOfRecords(filename)
```

Arguments:

Arguments

Argument	Description
filename	The name of a QVD file, if necessary including path, as a folder or web data connection. **Example:** ***'lib://Table Files/'*** In legacy scripting mode, the following path formats are also supported: • absolute **Example:** ***c:\data*** • relative to the Qlik Sense app working directory. **Example:** ***data*** • URL address (HTTP or FTP), pointing to a location on the Internet or an intranet. **Example:** ***http://www.qlik.com***

Examples:

```
QvdNoOfRecords ('MyFile.qvd')
QvdNoOfRecords ('C:\MyDir\MyFile.qvd')
QvdNoOfRecords ('lib://DataFiles/MyFile.qvd')
```

QvdTableName

This script function returns the name of the table stored in a QVD file.

Syntax:

```
QvdTableName(filename)
```

Arguments:

Arguments

Argument	Description
filename	The name of a QVD file, if necessary including path, as a folder or web data connection. **Example:** ***'lib://Table Files/'*** In legacy scripting mode, the following path formats are also supported: • absolute **Example:** ***c:\data*** • relative to the Qlik Sense app working directory. **Example:** ***data*** • URL address (HTTP or FTP), pointing to a location on the Internet or an intranet. **Example:** ***http://www.qlik.com***

Examples:

```
QvdTableName ('MyFile.qvd')
QvdTableName ('C:\MyDir\MyFile.qvd')
QvdTableName ('lib://data\MyFile.qvd')
```

5.11 Financial functions

Financial functions can be used in the data load script and in chart expressions to calculate payments and interest rates.
For all the arguments, cash that is paid out is represented by negative numbers. Cash received is represented by positive numbers.
Listed here are the arguments that are used in the financial functions (excepting the ones beginning with **range**-).

> *For all financial functions it is vital that you are consistent when specifying units for **rate** and **nper**. If monthly payments are made on a five-year loan at 6% annual interest, use 0.005 (6%/12) for **rate** and 60 (5*12) for **nper**. If annual payments are made on the same loan, use 6% for **rate** and 5 for **nper**.*

Financial functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

FV

This function returns the future value of an investment based on periodic, constant payments and a simple annual interest.

```
FV (rate, nper, pmt [ ,pv [ , type ] ])
```

nPer

This function returns the number of periods for an investment based on periodic, constant payments and a constant interest rate.

```
nPer (rate, pmt, pv [ ,fv [ , type ] ])
```

Pmt

This function returns the payment for a loan based on periodic, constant payments and a constant interest rate. It cannot change over the life of the annuity. A payment is stated as a negative number, for example, -20.

```
Pmt (rate, nper, pv [ ,fv [ , type ] ] )
```

PV

This function returns the present value of an investment.

```
PV (rate, nper, pmt [ ,fv [ , type ] ])
```

Rate

This function returns the interest rate per period on annuity. The result has a default number format of **Fix** two decimals and %.

```
Rate (nper, pmt , pv [ ,fv [ , type ] ])
```

BlackAndSchole

The Black and Scholes model is a mathematical model for financial market derivative instruments. The formula calculates the theoretical value of an option. In Qlik Sense, the **BlackAndSchole** function returns the value according to the Black and Scholes unmodified formula (European style options).

```
BlackAndSchole(strike , time_left , underlying_price , vol , risk_free_rate ,
type)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
strike	The future purchase price of the stock.
time_left	The number of time periods remaining.
underlying_price	The current value of the stock.

Argument	Description
vol	Volatility (of the stock price) expressed as a percentage in decimal form, per time period.
risk_free_rate	The risk-free rate expressed as a percentage in decimal form, per time period.
call_or_put	The type of option: 'c', 'call' or any non-zero numeric value for call options 'p', 'put' or 0 for put options.

Limitations:

The value of strike, time_left, and underlying_price must be >0.

The value of vol and risk_free_rate must be: <0 or >0.

Examples and results:

Scripting examples

Example	Result
`BlackAndSchole(130, 4, 68.5, 0.4, 0.04, 'call')` This calculates the theoretical price of an option to buy a share that is worth 68.5 today, at a value of 130 in 4 years. The formula uses a volatility of 0.4 (40%) per year and a risk-free interest rate of 0.04 (4%).	Returns 11.245

FV

This function returns the future value of an investment based on periodic, constant payments and a simple annual interest.

Syntax:

```
FV(rate, nper, pmt [ ,pv [ , type ] ])
```

Return data type: numeric. The result has a default number format of money..

Arguments:

Arguments

Argument	Description
rate	The interest rate per period.
nper	The total number of payment periods in an annuity.
pmt	The payment made each period. It cannot change over the life of the annuity. A payment is stated as a negative number, for example, -20.

Argument	Description
pv	The present value, or lump-sum amount, that a series of future payments is worth right now. If **pv** is omitted, it is assumed to be 0 (zero).
type	Should be 0 if payments are due at the end of the period and 1 if payments are due at the beginning of the period. If **type** is omitted, it is assumed to be 0.

Examples and results:

Scripting example

Example	Result
You are paying a new household appliance by 36 monthly installments of $20. The interest rate is 6% per annum. The bill comes at the end of every month. What is the total invested, when the last bill has been paid? `FV(0.005,36,-20)`	Returns $786.72

nPer

This function returns the number of periods for an investment based on periodic, constant payments and a constant interest rate.

Syntax:

```
nPer(rate, pmt, pv [ ,fv [ , type ] ])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
rate	The interest rate per period.
nper	The total number of payment periods in an annuity.
pmt	The payment made each period. It cannot change over the life of the annuity. A payment is stated as a negative number, for example, -20.
pv	The present value, or lump-sum amount, that a series of future payments is worth right now. If **pv** is omitted, it is assumed to be 0 (zero).
fv	The future value, or cash balance, you want to attain after the last payment is made. If **fv** is omitted, it is assumed to be 0.
type	Should be 0 if payments are due at the end of the period and 1 if payments are due at the beginning of the period. If **type** is omitted, it is assumed to be 0.

Examples and results:

Scripting example

Example	Result
You want to sell a household appliance by monthly installments of $20. The interest rate is 6% per annum. The bill comes at the end of every month. How many periods are required if the value of the money received after the last bill has been paid should equal $800? `nPer(0.005,-20,0,800)`	Returns 36.56

Pmt

This function returns the payment for a loan based on periodic, constant payments and a constant interest rate. It cannot change over the life of the annuity. A payment is stated as a negative number, for example, -20.

```
Pmt(rate, nper, pv [ ,fv [ , type ] ] )
```

Return data type: numeric. The result has a default number format of money..

To find the total amount paid over the duration of the loan, multiply the returned **pmt** value by **nper**.

Arguments:

Arguments

Argument	Description
rate	The interest rate per period.
nper	The total number of payment periods in an annuity.
pv	The present value, or lump-sum amount, that a series of future payments is worth right now. If **pv** is omitted, it is assumed to be 0 (zero).
fv	The future value, or cash balance, you want to attain after the last payment is made. If **fv** is omitted, it is assumed to be 0.
type	Should be 0 if payments are due at the end of the period and 1 if payments are due at the beginning of the period. If **type** is omitted, it is assumed to be 0.

Examples and results:

Scripting examples

Example	Result
The following formula returns the monthly payment on a $20,000 loan at an annual rate of 10 percent, that must be paid off in 8 months: `Pmt(0.1/12,8,20000)`	Returns -$2,594.66

Example	Result
For the same loan, if payment is due at the beginning of the period, the payment is: `Pmt(0.1/12,8,20000,0,1)`	Returns -$2,573.21

PV

This function returns the present value of an investment.

```
PV(rate, nper, pmt [ ,fv [ , type ] ])
```

Return data type: numeric. The result has a default number format of money..

The present value is the total amount that a series of future payments is worth right now. For example, when borrowing money, the loan amount is the present value to the lender.

Arguments:

Arguments

Argument	Description
rate	The interest rate per period.
nper	The total number of payment periods in an annuity.
pmt	The payment made each period. It cannot change over the life of the annuity. A payment is stated as a negative number, for example, -20.
fv	The future value, or cash balance, you want to attain after the last payment is made. If **fv** is omitted, it is assumed to be 0.
type	Should be 0 if payments are due at the end of the period and 1 if payments are due at the beginning of the period. If **type** is omitted, it is assumed to be 0.

Examples and results:

Scripting example

Example	Result
What is the present value of a debt, when you have to pay $100 at the end of each month during a five-year period, given an interest rate of 7%? `PV(0.07/12,12*5,-100,0,0)`	Returns $5,050.20

Rate

This function returns the interest rate per period on annuity. The result has a default number format of **Fix** two decimals and %.

Syntax:

```
Rate(nper, pmt , pv [ ,fv [ , type ] ])
```

Return data type: numeric.

The **rate** is calculated by iteration and can have zero or more solutions. If the successive results of **rate** do not converge, a NULL value will be returned.

Arguments:

Arguments

Argument	Description
nper	The total number of payment periods in an annuity.
pmt	The payment made each period. It cannot change over the life of the annuity. A payment is stated as a negative number, for example, -20.
pv	The present value, or lump-sum amount, that a series of future payments is worth right now. If **pv** is omitted, it is assumed to be 0 (zero).
fv	The future value, or cash balance, you want to attain after the last payment is made. If **fv** is omitted, it is assumed to be 0.
type	Should be 0 if payments are due at the end of the period and 1 if payments are due at the beginning of the period. If **type** is omitted, it is assumed to be 0.

Examples and results:

Scripting example

Example	Result
What is the interest rate of a five-year $10,000 annuity loan with monthly payments of $300? `Rate(60,-300,10000)`	Returns 2.00%

5.12 Formatting functions

The formatting functions impose the display format on the input numeric fields or expressions, Depending on data type, you can specify the characters for the decimal separator, thousands separator, and so on.

The functions all return a dual value with both the string and the number value, but can be thought of as performing a number-to-string conversion. **Dual()** is a special case, but the other formatting functions take the numeric value of the input expression and generate a string representing the number.

In contrast, the interpretation functions do the opposite: they take string expressions and evaluate them as numbers, specifying the format of the resulting number.

The functions can be used both in data load scripts and chart expressions.

All number representations are given with a decimal point as the decimal separator.

Formatting functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

ApplyCodepage

ApplyCodepage() applies a different code page character set to the field or text stated in the expression. The **codepage** argument must be in number format.

```
ApplyCodepage (text, codepage)
```

Date

Date() formats an expression as a date using the format set in the system variables in the data load script, or the operating system, or a format string, if supplied.

```
Date (number[, format])
```

Dual

Dual() combines a number and a string into a single record, such that the number representation of the record can be used for sorting and calculation purposes, while the string value can be used for display purposes.

```
Dual (text, number)
```

Interval

Interval() formats a number as a time interval using the format in the system variables in the data load script, or the operating system, or a format string, if supplied.

```
Interval (number[, format])
```

Money

Money() formats an expression numerically as a money value, in the format set in the system variables set in the data load script, or in the operating system, unless a format string is supplied, and optional decimal and thousands separators.

```
Money (number[, format[, dec_sep [, thou_sep]]])
```

Num

Num() formats a number, that is it converts the numeric value of the input to display text using the format specified in the second parameter. If the second parameter is omitted, it uses the decimal and thousand separators set in the data load script. Custom decimal and thousand separator symbols are optional parameters.

```
Num (number[, format[, dec_sep [, thou_sep]]])
```

Time

Time() formats an expression as a time value, in the time format set in the system variables in the data load script, or in the operating system, unless a format string is supplied.

```
Time (number[, format])
```

Timestamp

TimeStamp() formats an expression as a date and time value, in the timestamp format set in the system variables in the data load script, or in the operating system, unless a format string is supplied.

```
Timestamp (number[, format])
```

See also:

Interpretation functions (page 837)

ApplyCodepage

ApplyCodepage() applies a different code page character set to the field or text stated in the expression. The **codepage** argument must be in number format.

Although ApplyCodepage can be used in chart expressions, it is more commonly used as a script function in the data load editor. For example, as you load files that might have been saved in different character sets out of your control, you can apply the code page that represents the character set you require.

Syntax:

```
ApplyCodepage(text, codepage)
```

Return data type: string

Arguments:

Arguments

Argument	Description
text	Field or text to which you want to apply a different code page, given by the argument **codepage**.
codepage	Number representing the code page to be applied to the field or expression given by **text**.

Examples and results:

Scripting examples

Example	Result
`LOAD ApplyCodepage(ROWX,1253) as GreekProduct, ApplyCodepage (ROWY, 1255) as HebrewProduct, ApplyCodepage (ROWZ, 65001) as EnglishProduct; SQL SELECT ROWX, ROWY, ROWZ From Products;`	When loading from SQL the source might have a mixture of different character sets: Cyrillic, Hebrew, and so on, from the UTF-8 format. These would be required to be loaded row by row, applying a different code page for each row. The **codepage** value 1253 represents Windows Greek character set, the value 1255 represents Hebrew, and the value 65001 represents standard Latin UTF-8 characters.

See also: *Character set (page 129)*

Date

Date() formats an expression as a date using the format set in the system variables in the data load script, or the operating system, or a format string, if supplied.

Syntax:

```
Date(number[, format])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
number	The number to be formatted.
format	String describing the format of the resulting string. If no format string is supplied, the date format set in the system variables in the data load script, or the operating system is used.

Examples and results:
The examples below assume the following default settings:

- Date setting 1: YY-MM-DD
- Date setting 2: M/D/YY

Example:

`Date( A )`
where A=35648

Results table

Results	Setting 1	Setting 2
String:	97-08-06	8/6/97
Number:	35648	35648

Example:

```
Date( A, 'YY.MM.DD' )
```
where A=35648

Results table

Results	Setting 1	Setting 2
String:	97.08.06	97.08.06
Number:	35648	35648

Example:

```
Date( A, 'DD.MM.YYYY' )
```
where A=35648.375

Results table

Results	Setting 1	Setting 2
String:	06.08.1997	06.08.1997
Number:	35648.375	35648.375

Example:

```
Date( A, 'YY.MM.DD' )
```
where A=8/6/97

Results table

Results	Setting 1	Setting 2
String:	NULL (nothing)	97.08.06
Number:	NULL	35648

Dual

Dual() combines a number and a string into a single record, such that the number representation of the record can be used for sorting and calculation purposes, while the string value can be used for display purposes.

Syntax:

```
Dual(text, number)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
text	The string value to be used in combination with the number argument.
number	The number to be used in combination with the string in the string argument.

In Qlik Sense, all field values are potentially dual values. This means that the field values can have both a numeric value and a textual value. An example is a date that could have a numeric value of 40908 and the textual representation '2011-12-31'.

When several data items read into one field have different string representations but the same valid number representation, they will all share the first string representation encountered.

*The **dual** function is typically used early in the script, before other data is read into the field concerned, in order to create that first string representation, which will be shown in filter panes.*

Examples and results:

Scripting examples

Example	Description
Add the following examples to your script and run it. `Load dual ( NameDay,NumDay ) as DayOfWeek inline [ NameDay,NumDay Monday,0 Tuesday,1 Wednesday,2 Thursday,3 Friday,4 Saturday,5 Sunday,6 ];`	The field DayOfWeek can be used in a visualization, as a dimension, for example.In a table with the week days are automatically sorted into their correct number sequence, instead of alphabetical order.
`Load Dual('Q' & Ceil (Month(Now())/3), Ceil (Month(Now())/3)) as Quarter AutoGenerate 1;`	This example finds the current quarter. It is displayed as Q1 when the **Now()** function is run in the first three months of the year, Q2 for the second three months, and so on. However, when used in sorting, the field Quarter will behave as its numerical value: 1 to 4.

Example	Description
`Dual('Q' & Ceil(Month (Date)/3), Ceil(Month (Date)/3)) as Quarter`	As in the previous example, the field Quarter is created with the text values 'Q1' to 'Q4', and assigned the numeric values 1 to 4. In order to use this in the script the values for Date must be loaded.
`Dual(WeekYear(Date) & '-W' & Week(Date), WeekStart(Date)) as YearWeek`	This example create sa field YearWeek with text values of the form '2012-W22' and at the same time, assigns a numeric value corresponding to the date number of the first day of the week, for example: 41057. In order to use this in the script the values for Date must be loaded.

Interval

Interval() formats a number as a time interval using the format in the system variables in the data load script, or the operating system, or a format string, if supplied.

Intervals may be formatted as a time, as days or as a combination of days, hours, minutes, seconds and fractions of seconds.

Syntax:

```
Interval(number[, format])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
number	The number to be formatted.
format	String describing how the resulting interval string is to be formatted. If omitted, the short date format, time format, and decimal separator set in the operating system are used.

Examples and results:
The examples below assume the following default settings:

- Date format setting 1: YY-MM-DD
- Date format setting 2: hh:mm:ss
- Number decimal separator: .

Results table

Example	String	Number
`Interval( A )` where A=0.375	09:00:00	0.375

Example	String	Number
`Interval( A )` where A=1.375	33:00:00	1.375
`Interval( A, 'D hh:mm' )` where A=1.375	1 09:00	1.375
`Interval( A-B, 'D hh:mm' )` where A=97-08-06 09:00:00 and B=96-08-06 00:00:00	365 09:00	365.375

Money

Money() formats an expression numerically as a money value, in the format set in the system variables set in the data load script, or in the operating system, unless a format string is supplied, and optional decimal and thousands separators.

Syntax:

```
Money(number[, format[, dec_sep[, thou_sep]]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
number	The number to be formatted.
format	String describing how the resulting money string is to be formatted.
dec_sep	String specifying the decimal number separator.
thou_sep	String specifying the thousands number separator.

If arguments 2-4 are omitted, the currency format set in the operating system is used.

Examples and results:
The examples below assume the following default settings:

- MoneyFormat setting 1: kr ##0,00, MoneyThousandSep' '
- MoneyFormat setting 2: $ #,##0.00, MoneyThousandSep','

Example:

`Money( A )`
where A=35648

Results table

Results	Setting 1	Setting 2
String:	kr 35 648,00	$ 35,648.00
Number:	35648.00	35648.00

Example:

```
Money( A, '#,##0 ¥', '.' , ',' )
```
where A=3564800

Results table

Results	Setting 1	Setting 2
String:	3,564,800 ¥	3,564,800 ¥
Number:	3564800	3564800

Num

Num() formats a number, that is it converts the numeric value of the input to display text using the format specified in the second parameter. If the second parameter is omitted, it uses the decimal and thousand separators set in the data load script. Custom decimal and thousand separator symbols are optional parameters.

Syntax:

```
Num(number[, format[, dec_sep [, thou_sep]]])
```

Return data type: dual

The Num function returns a dual value with both the string and the numeric value. The function takes the numeric value of the input expression and generates a string representing the number.

Arguments:

Arguments

Argument	Description
number	The number to be formatted.
format	String specifying how the resulting string is to be formatted. If omitted, the decimal and thousand separators that are set in the data load script are used.
dec_sep	String specifying the decimal number separator. If omitted, the value of the variable DecimalSep that is set in the data load script is used.
thou_sep	String specifying the thousands number separator. If omitted, the value of the variable ThousandSep that is set in the data load script is used.

Example: Chart expression

Example:

The following table shows the results when field A equals 35648.312.

Results

A	Result
Num(A)	35648.312 (depends on environment variables in script)
Num(A, '0.0', '.')	35648.3
Num(A, '0,00', ',')	35648,31
Num(A, '#,##0.0', '.', ',')	35,648.3
Num(A, '# ##0', ',', ' ')	35 648

Example: Load script

Load script

Num can be used in load script to format a number, even if the thousand and decimal separators are already set in the script. The load script below includes specific thousand and decimal separators but then uses *Num* to format data in different ways.

In the **Data load editor**, create a new section, and then add the example script and run it. Then add, at least, the fields listed in the results column to a sheet in your app to see the result.

```
SET ThousandSep=',';
SET DecimalSep='.';
Transactions:
Load
*,
Num(transaction_amount) as [No formatting],
Num(transaction_amount,'0') as [0],
Num(transaction_amount,'#,##0') as [#,##0],
Num(transaction_amount,'# ###,00') as [# ###,00],
Num(transaction_amount,'# ###,00',',',' ') as [# ###,00 , ',' , ' '],
Num(transaction_amount,'#,###.00','.',',') as [#,###.00 , '.' , ','],
Num(transaction_amount,'$#,###.00') as [$#,###.00],
;
Load * Inline [
transaction_id, transaction_date, transaction_amount, transaction_quantity, discount,
customer_id, size, color_code
3750, 20180830, 12423.56, 23, 0,2038593, L, Red
3751, 20180907, 5356.31, 6, 0.1, 203521, m, orange
3752, 20180916, 15.75, 1, 0.22, 5646471, S, blue
3753, 20180922, 1251, 7, 0, 3036491, l, Black
3754, 20180922, 21484.21, 1356, 75, 049681, xs, Red
3756, 20180922, -59.18, 2, 0.333333333333333, 2038593, M, Blue
3757, 20180923, 3177.4, 21, .14, 203521, XL, Black
];
```

Qlik Sense table showing the results from different uses of the *Num* function in the load script. The fourth column of the table contains incorrect formatting use, for example purposes.

No formatting	0	#,##0	# ###,00	# ###,00 , ',' , ' '	#,###.00 , '.' , ','	$#,###.00
-59.18	-59	-59	-59###,00	-59,18	-59.18	$-59,18
15.75	16	16	16###,00	15,75	15.75	$15,75
1251	1251	1,251	1251###,00	1 251,00	1,251.00	$1,251.00
3177.4	3177	3,177	3177###,00	3 177,40	3,177.40	$3,177.40
5356.31	5356	5,356	5356###,00	5 356,31	5,356.31	$5,356.31
12423.56	12424	12,424	12424###,00	12 423,56	12,423.56	$12,423.56
21484.21	21484	21,484	21484###,00	21 484,21	21,484.21	$21,484.21

Example: Load script

Load script

Num can be used in a load script to format a number as a percentage.

In the **Data load editor**, create a new section, and then add the example script and run it. Then add, at least, the fields listed in the results column to a sheet in your app to see the result.

```
SET ThousandSep=',';
SET DecimalSep='.';
Transactions:
Load
*,
Num(discount,'#,##0%') as [Discount #,##0%]
;
Load * Inline [
transaction_id, transaction_date, transaction_amount, transaction_quantity, discount,
customer_id, size, color_code
3750, 20180830, 12423.56, 23, 0,2038593, L, Red
3751, 20180907, 5356.31, 6, 0.1, 203521, m, orange
3752, 20180916, 15.75, 1, 0.22, 5646471, S, blue
3753, 20180922, 1251, 7, 0, 3036491, l, Black
3754, 20180922, 21484.21, 1356, 75, 049681, xs, Red
3756, 20180922, -59.18, 2, 0.333333333333333, 2038593, M, Blue
3757, 20180923, 3177.4, 21, .14, 203521, XL, Black
];
```

Qlik Sense table showing the results of the *Num* function being used in the load script to format percentages.

Discount	Discount #,##0%
0.333333333333333	33%

Discount	Discount #,##0%
0.22	22%
0	0%
.14	14%
0.1	10%
0	0%
75	7,500%

Time

Time() formats an expression as a time value, in the time format set in the system variables in the data load script, or in the operating system, unless a format string is supplied.

Syntax:

```
Time(number[, format])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
number	The number to be formatted.
format	String describing how the resulting time string is to be formatted. If omitted, the short date format, time format, and decimal separator set in the operating system is used.

Examples and results:
The examples below assume the following default settings:

- Time format setting 1: hh:mm:ss
- Time format setting 2: hh.mm.ss

Example:

```
Time( A )
```
where A=0.375

Results table

Results	Setting 1	Setting 2
String:	09:00:00	09.00.00
Number:	0.375	0.375

Example:

```
Time( A )
```
where A=35648.375

Results table

Results	Setting 1	Setting 2
String:	09:00:00	09.00.00
Number:	35648.375	35648.375

Example:

```
Time( A, 'hh-mm' )
```
where A=0.99999

Results table

Results	Setting 1	Setting 2
String:	23-59	23-59
Number:	0.99999	0.99999

Timestamp

TimeStamp() formats an expression as a date and time value, in the timestamp format set in the system variables in the data load script, or in the operating system, unless a format string is supplied.

Syntax:

```
Timestamp(number[, format])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
number	The number to be formatted.
format	String describing how the resulting timestamp string is to be formatted. If omitted, the short date format, time format, and decimal separator set in the operating system is used.

Examples and results:
The examples below assume the following default settings:

- TimeStampFormat setting 1: YY-MM-DD hh:mm:ss
- TimeStampFormat setting 2: M/D/YY hh:mm:ss

Example:

```
Timestamp( A )
```
where A=35648.375

Results table

Results	Setting 1	Setting 2
String:	97-08-06 09:00:00	8/6/97 09:00:00
Number:	35648.375	35648.375

Example:

```
Timestamp( A,'YYYY-MM-DD hh.mm')
```
where A=35648

Results table

Results	Setting 1	Setting 2
String:	1997-08-06 00.00	1997-08-06 00.00
Number:	35648	35648

5.13 General numeric functions

In these general numeric functions, the arguments are expressions where **x** should be interpreted as a real valued number. All functions can be used in both data load scripts and chart expressions.

General numeric functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

bitcount

BitCount() returns how many bits in the binary equivalent of a decimal number are set to 1. That is, the function returns the number of set bits in **integer_number**, where **integer_number** is interpreted as a signed 32-bit integer.

```
BitCount(integer_number)
```

div

Div() returns the integer part of the arithmetic division of the first argument by the second argument. Both parameters are interpreted as real numbers, that is, they do not have to be integers.

```
Div(integer_number1, integer_number2)
```

fabs

Fabs() returns the absolute value of **x**. The result is a positive number.

```
Fabs(x)
```

fact

Fact() returns the factorial of a positive integer **x**.

```
Fact(x)
```

frac

Frac() returns the fraction part of **x**.

```
Frac(x)
```

sign

Sign() returns 1, 0 or -1 depending on whether **x** is a positive number, 0, or a negative number.

```
Sign(x)
```

Combination and permutation functions

combin

Combin() returns the number of combinations of **q** elements that can be picked from a set of **p** items. As represented by the formula: `Combin(p,q) = p! / q!(p-q)!` The order in which the items are selected is insignificant.

```
Combin(p, q)
```

permut

Permut() returns the number of permutations of **q** elements that can be selected from a set of **p** items. As represented by the formula: `Permut(p,q) = (p)! / (p - q)!` The order in which the items are selected is significant.

```
Permut(p, q)
```

Modulo functions

fmod

fmod() is a generalized modulo function that returns the remainder part of the integer division of the first argument (the dividend) by the second argument (the divisor). The result is a real number. Both arguments are interpreted as real numbers, that is, they do not have to be integers.

```
Fmod(a, b)
```

mod

Mod() is a mathematical modulo function that returns the non-negative remainder of an integer division. The first argument is the dividend, the second argument is the divisor, Both arguments must be integer values.

```
Mod(integer_number1, integer_number2)
```

Parity functions

even

Even() returns True (-1), if **integer_number** is an even integer or zero. It returns False (0), if **integer_number** is an odd integer, and NULL if **integer_number** is not an integer.

```
Even(integer_number)
```

odd

Odd() returns True (-1), if **integer_number** is an odd integer or zero. It returns False (0), if **integer_number** is an even integer, and NULL if **integer_number** is not an integer.

```
Odd(integer_number)
```

Rounding functions

ceil

Ceil() rounds up a number to the nearest multiple of the **step** shifted by the **offset** number.

```
Ceil(x[, step[, offset]])
```

floor

Floor() rounds down a number to the nearest multiple of the **step** shifted by the **offset** number.

```
Floor(x[, step[, offset]])
```

round

Round() returns the result of rounding a number up or down to the nearest multiple of **step** shifted by the **offset** number.

```
Round( x [ , step [ , offset ]])
```

BitCount

BitCount() returns how many bits in the binary equivalent of a decimal number are set to 1. That is, the function returns the number of set bits in **integer_number**, where **integer_number** is interpreted as a signed 32-bit integer.

Syntax:

```
BitCount(integer_number)
```

Return data type: integer

Examples and results:

Examples and results

Examples	Results
`BitCount ( 3 )`	3 is binary 11, therefore this returns 2
`BitCount ( -1 )`	-1 is 64 ones in binary, therefore this returns 64

Ceil

Ceil() rounds up a number to the nearest multiple of the **step** shifted by the **offset** number.

Compare with the **floor** function, which rounds input numbers down.

Syntax:

```
Ceil(x[, step[, offset]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
x	Input number.
step	Interval increment. The default value is 1.
offset	Defines the base of the step interval. The default value is 0.

Examples and results:

Examples and results

Examples	Results
`Ceil(2.4 )`	Returns 3 In this example, the size of the step is 1 and the base of the step interval is 0. The intervals are ...0 < x <=1, 1 < x <= 2, **2< x <=3**, 3< x <=4...
`Ceil(4.2 )`	Returns 5
`Ceil(3.88 ,0.1)`	Returns 3.9 In this example, the size of the interval is 0.1 and the base of the interval is 0. The intervals are ... 3.7 < x <= 3.8, **3.8 < x <= 3.9**, 3.9 < x <= 4.0...
`Ceil(3.88 ,5)`	Returns 5
`Ceil(1.1 ,1)`	Returns 2

Examples	Results
`Ceil(1.1 ,1,0.5)`	Returns 1.5 In this example, the size of the step is 1 and the offset is 0.5. It means that the base of the step interval is 0.5 and not 0. The intervals are ...**0.5 < x <=1.5**, 1.5 < x <= 2.5, 2.5< x <=3.5, 3.5< x <=4.5...
`Ceil(1.1 ,1,-0.01)`	Returns 1.99 The intervals are ...-0.01< x <= 0.99, **0.99< x <= 1.99**, 1.99 < x <=2.99...

Combin

Combin() returns the number of combinations of **q** elements that can be picked from a set of **p** items. As represented by the formula: `Combin(p,q) = p! / q!(p-q)!` The order in which the items are selected is insignificant.

Syntax:

```
Combin(p, q)
```

Return data type: integer

Limitations:

Non-integer items will be truncated.

Examples and results:

Examples and results

Examples	Results
How many combinations of 7 numbers can be picked from a total of 35 lottery numbers? `Combin( 35,7 )`	Returns 6,724,520

Div

Div() returns the integer part of the arithmetic division of the first argument by the second argument. Both parameters are interpreted as real numbers, that is, they do not have to be integers.

Syntax:

```
Div(integer_number1, integer_number2)
```

Return data type: integer

Examples and results:

Examples and results

Examples	Results
`Div( 7,2 )`	Returns 3
`Div( 7.1,2.3 )`	Returns 3
`Div( 9,3 )`	Returns 3
`Div( -4,3 )`	Returns -1
`Div( 4,-3 )`	Returns -1
`Div( -4,-3 )`	Returns 1

Even

Even() returns True (-1), if **integer_number** is an even integer or zero. It returns False (0), if **integer_number** is an odd integer, and NULL if **integer_number** is not an integer.

Syntax:

```
Even(integer_number)
```

Return data type: Boolean

Examples and results:

Examples and results

Examples	Results
`Even( 3 )`	Returns 0, False
`Even( 2 * 10 )`	Returns -1, True
`Even( 3.14 )`	Returns NULL

Fabs

Fabs() returns the absolute value of **x**. The result is a positive number.

Syntax:

```
fabs(x)
```

Return data type: numeric

Examples and results:

Examples and results

Examples	Results
`fabs( 2.4 )`	Returns 2.4
`fabs( -3.8 )`	Returns 3.8

Fact

Fact() returns the factorial of a positive integer **x**.

Syntax:

```
Fact(x)
```

Return data type: integer

Limitations:

If the number **x** is not an integer, it will be truncated. Non-positive numbers will return NULL.

Examples and results:

Examples and results

Examples	Results
`Fact( 1 )`	Returns 1
`Fact( 5 )`	Returns 120 (1 * 2 * 3 * 4 * 5 = 120)
`Fact( -5 )`	Returns NULL

Floor

Floor() rounds down a number to the nearest multiple of the **step** shifted by the **offset** number.

Compare with the **ceil** function, which rounds input numbers up.

Syntax:

```
Floor(x[, step[, offset]])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
x	Input number.
step	Interval increment. The default value is 1.
offset	Defines the base of the step interval. The default value is 0.

Examples and results:

Examples and results

Examples	Results
`Floor(2.4)`	Returns 2 In this example, the size of the step is 1 and the base of the step interval is 0. The intervals are ...0 <= x <1, 1 <= x < 2, **2<= x <3**, 3<= x <4....
`Floor(4.2)`	Returns 4
`Floor(3.88 ,0.1)`	Returns 3.8 In this example, the size of the interval is 0.1 and the base of the interval is 0. The intervals are ... 3.7 <= x < 3.8, **3.8 <= x < 3.9**, 3.9 <= x < 4.0...
`Floor(3.88 ,5)`	Returns 0
`Floor(1.1 ,1)`	Returns 1
`Floor(1.1 ,1,0.5)`	Returns 0.5 In this example, the size of the step is 1 and the offset is 0.5. It means that the base of the step interval is 0.5 and not 0. The intervals are ...**0.5 <= x <1.5**, 1.5 <= x < 2.5, 2.5<= x <3.5,...

Fmod

fmod() is a generalized modulo function that returns the remainder part of the integer division of the first argument (the dividend) by the second argument (the divisor). The result is a real number. Both arguments are interpreted as real numbers, that is, they do not have to be integers.

Syntax:

```
fmod(a, b)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
a	Dividend
b	Divisor

Examples and results:

Examples and results

Examples	Results
`fmod( 7,2 )`	Returns 1
`fmod( 7.5,2 )`	Returns 1.5
`fmod( 9,3 )`	Returns 0
`fmod( -4,3 )`	Returns -1
`fmod( 4,-3 )`	Returns 1
`fmod( -4,-3 )`	Returns -1

Frac

Frac() returns the fraction part of **x**.

The fraction is defined in such a way that `Frac(x ) + Floor(x ) = x`. In simple terms, this means that the fractional part of a positive number is the difference between the number (x) and the integer that precedes the fractional part.

For example: The fractional part of 11.43 = 11.43 - 11 = 0.43

For a negative number, say -1.4, Floor(-1.4) = -2, which produces the following result:

The fractional part of -1.4 = -1.4 - (-2) = -1.4 + 2 = 0.6

Syntax:

```
Frac(x)
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
x	Number to return fraction for.

Examples and results:

Examples and results

Examples	Results
`Frac( 11.43 )`	Returns 0.43
`Frac( -1.4 )`	Returns 0.6
Extract the time component from the numeric representation of a timestamp, thus omitting the date. `Time(Frac(44518.663888889))`	Returns 3:56:00 PM

Mod

Mod() is a mathematical modulo function that returns the non-negative remainder of an integer division. The first argument is the dividend, the second argument is the divisor, Both arguments must be integer values.

Syntax:

```
Mod(integer_number1, integer_number2)
```

Return data type: integer

Limitations:

integer_number2 must be greater than 0.

Examples and results:

Examples and results

Examples	Results
`Mod( 7,2 )`	Returns 1
`Mod( 7.5,2 )`	Returns NULL
`Mod( 9,3 )`	Returns 0

Examples	Results
`Mod( -4,3 )`	Returns 2
`Mod( 4,-3 )`	Returns NULL
`Mod( -4,-3 )`	Returns NULL

Odd

Odd() returns True (-1), if **integer_number** is an odd integer or zero. It returns False (0), if **integer_number** is an even integer, and NULL if **integer_number** is not an integer.

Syntax:

```
Odd(integer_number)
```

Return data type: Boolean

Examples and results:

Examples and results

Examples	Results
`Odd( 3 )`	Returns -1, True
`Odd( 2 * 10 )`	Returns 0, False
`Odd( 3.14 )`	Returns NULL

Permut

Permut() returns the number of permutations of **q** elements that can be selected from a set of **p** items. As represented by the formula: `Permut(p,q) = (p)! / (p - q)!` The order in which the items are selected is significant.

Syntax:

```
Permut(p, q)
```

Return data type: integer

Limitations:

Non-integer arguments will be truncated.

Examples and results:

Examples and results

Examples	Results
In how many ways could the gold, silver and bronze medals be distributed after a 100 m final with 8 participants? `Permut( 8,3 )`	Returns 336

Round

Round() returns the result of rounding a number up or down to the nearest multiple of **step** shifted by the **offset** number.

If the number to round is exactly in the middle of an interval, it is rounded upwards.

Syntax:

```
Round(x[, step[, offset]])
```

Return data type: numeric

> *If you are rounding a floating point number you may observe erroneous results. These rounding errors occur because floating point numbers are represented by a finite number of binary digits. Therefore, results are calculated using a number that is already rounded. If these rounding errors will affect your work, multiply the numbers to convert them to integers before rounding.*

Arguments:

Arguments

Argument	Description
x	Input number.
step	Interval increment. The default value is 1.
offset	Defines the base of the step interval. The default value is 0.

Examples and results:

Examples and results

Examples	Results
`Round(3.8 )`	Returns 4 In this example, the size of the step is 1 and the base of the step interval is 0. The intervals are ...0 <= x <1, 1 <= x < 2, 2<= x <3, **3<= x <4**...
`Round(3.8,4 )`	Returns 4
`Round(2.5 )`	Returns 3. In this example, the size of the step is 1 and the base of the step interval is 0. The intervals are ...0 <= x <1, 1 <= x <2, **2<= x <3**...
`Round(2,4 )`	Returns 4. Rounded up because 2 is exactly half of the step interval of 4. In this example, the size of the step is 4 and the base of the step interval is 0. The intervals are ...**0 <= x <4**, 4 <= x <8, 8<= x <12...
`Round(2,6 )`	Returns 0. Rounded down because 2 is less than half of the step interval of 6. In this example, the size of the step is 6 and the base of the step interval is 0. The intervals are ...**0 <= x <6**, 6 <= x <12, 12<= x <18...
`Round(3.88 ,0.1)`	Returns 3.9 In this example, the size of the step is 0.1 and the base of the step interval is 0. The intervals are ... 3.7 <= x <3.8, **3.8 <= x <3.9**, 3.9 <= x < 4.0...
`Round (3.88875,1/1000)`	Returns 3.889 In this example, the size of the step is 0.001, which rounds the number up and limits it to three decimal places.
`Round(3.88 ,5)`	Returns 5
`Round(1.1 ,1,0.5)`	Returns 1.5 In this example, the size of the step is 1 and the base of the step interval is 0.5. The intervals are ...**0.5 <= x <1.5**, 1.5 <= x <2.5, 2.5<= x <3.5...

Sign

Sign() returns 1, 0 or -1 depending on whether **x** is a positive number, 0, or a negative number.

Syntax:

```
Sign(x)
```

Return data type: numeric

Limitations:

If no numeric value is found, NULL is returned.

Examples and results:

Examples and results

Examples	Results
`Sign( 66 )`	Returns 1
`Sign( 0 )`	Returns 0
`Sign( - 234 )`	Returns -1

5.14 Geospatial functions

These functions are used to handle geospatial data in map visualizations. Qlik Sense follows GeoJSON specifications for geospatial data and supports the following:

- Point
- Linestring
- Polygon
- Multipolygon

For more information on GeoJSON specifications, see:
GeoJSON.org

Geospatial functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

There are two categories of geospatial functions: aggregation and non-aggregation.

Aggregation functions take a geometry set (points or areas) as input, and return a single geometry. For example, multiple areas can be merged together, and a single boundary for the aggregation can be drawn on the map.

Non-aggregation function take a sinlge geometry and return one geometry. For example, for the function GeoGetPolygonCenter(), if the boundary geometry of one area is set as input, the point geometry (longitude and latitude) for the center of that area is returned.

The following are aggregation functions:

GeoAggrGeometry

GeoAggrGeometry() is used to aggregate a number of areas into a larger area, for example aggregating a number of sub-regions to a region.

```
GeoAggrGeometry (field_name)
```

GeoBoundingBox

GeoBoundingBox() is used to aggregate a geometry into an area and calculate the smallest bounding box that contains all coordinates.

```
GeoBoundingBox (field_name)
```

GeoCountVertex

GeoCountVertex() is used to find the number of vertices a polygon geometry contains.

```
GeoCountVertex(field_name)
```

GeoInvProjectGeometry

GeoInvProjectGeometry() is used to aggregate a geometry into an area and apply the inverse of a projection.

```
GeoInvProjectGeometry(type, field_name)
```

GeoProjectGeometry

GeoProjectGeometry() is used to aggregate a geometry into an area and apply a projection.

```
GeoProjectGeometry(type, field_name)
```

GeoReduceGeometry

GeoReduceGeometry() is used to reduce the number of vertices of a geometry, and to aggregate a number of areas into one area, but still displaying the boundary lines from the individual areas.

```
GeoReduceGeometry (geometry)
```

The following are non-aggregation functions:

GeoGetBoundingBox

GeoGetBoundingBox() is used in scripts and chart expressions to calculate the smallest geospatial bounding box that contains all coordinates of a geometry.

```
GeoGetBoundingBox (geometry)
```

GeoGetPolygonCenter

GeoGetPolygonCenter() is used in scripts and chart expressions to calculate and return the center point of a geometry.

```
GeoGetPolygonCenter (geometry)
```

GeoMakePoint

GeoMakePoint() is used in scripts and chart expressions to create and tag a point with latitude and longitude.

```
GeoMakePoint (lat_field_name, lon_field_name)
```

GeoProject

GeoProject() is used in scripts and chart expressions to apply a projection to a geometry.

```
GeoProject (type, field_name)
```

GeoAggrGeometry

GeoAggrGeometry() is used to aggregate a number of areas into a larger area, for example aggregating a number of sub-regions to a region.

Syntax:

```
GeoAggrGeometry(field_name)
```

Return data type: string

Arguments:

Arguments

Argument	Description
field_name	A field or expression referring to a field containing the geometry to be represented. This could be either a point (or set of points) giving longitude and latitude, or an area.

Typically, **GeoAggrGeometry()** can be used to combine geospatial boundary data. For example, you might have postcode areas for suburbs in a city and sales revenues for each area. If a sales person's territory covers several postcode areas, it might be useful to present total sales by sales territory, rather than individual areas, and show the results on a color-filled map.

GeoAggrGeometry() can calculate the aggregation of the individual suburb geometries and generate the merged territory geometry in the data model. If then, the sales territory boundaries are adjusted, when the data is reloaded the new merged boundaries and revenues are reflected in the map.

As **GeoAggrGeometry()** is an aggregating function, if you use it in the script a **LOAD** statement with a **Group by** clause is required.

> *The boundary lines of maps created using **GeoAggrGeometry()** are those of the merged areas. If you want to display the individual boundary lines of the pre-aggregated areas, use **GeoReduceGeometry()**.*

Examples:

This example loads a KML file with area data, and then loads a table with the aggregated area data.

```
[MapSource]:
LOAD [world.Name],
       [world.Point],
       [world.Area]
 FROM [lib://Downloads/world.kml]
(kml, Table is [World.shp/Features]);
```

```
Map:
LOAD world.Name,
     GeoAggrGeometry(world.Area) as [AggrArea]
resident MapSource Group By world.Name;

Drop Table MapSource;
```

GeoBoundingBox

GeoBoundingBox() is used to aggregate a geometry into an area and calculate the smallest bounding box that contains all coordinates.

A GeoBoundingBox is represented as a list of four values: left, right, top, bottom.

Syntax:

```
GeoBoundingBox(field_name)
```

Return data type: string

Arguments:

Arguments

Argument	Description
field_name	A field or expression referring to a field containing the geometry to be represented. This could be either a point (or set of points) giving longitude and latitude, or an area.

GeoBoundingBox() aggregates a set of geometries and returns four coordinates for the smallest rectangle that contains all the coordinates of that aggregated geometry.

To visualize the result on a map, transfer the resulting string of four coordinates into a polygon format, tag the transferred field with a geopolygon format, and drag and drop that field into the map object. The rectangular boxes .will then be displayed in the map visualization.

GeoCountVertex

GeoCountVertex() is used to find the number of vertices a polygon geometry contains.

Syntax:

```
GeoCountVertex(field_name)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
field_name	A field or expression referring to a field containing the geometry to be represented. This could be either a point (or set of points) giving longitude and latitude, or an area.

GeoGetBoundingBox

GeoGetBoundingBox() is used in scripts and chart expressions to calculate the smallest geospatial bounding box that contains all coordinates of a geometry.

A geospatial bounding box, created by the function GeoBoundingBox() is represented as a list of four values: left, right, top, bottom.

Syntax:

```
GeoGetBoundingBox(field_name)
```

Return data type: string

Arguments:

Arguments

Argument	Description
field_name	A field or expression referring to a field containing the geometry to be represented. This could be either a point (or set of points) giving longitude and latitude, or an area.

*Do not use the **Group by** clause in the data load editor with this and other non-aggregating geospatial functions, because this will cause an error on load.*

GeoGetPolygonCenter

GeoGetPolygonCenter() is used in scripts and chart expressions to calculate and return the center point of a geometry.

In some cases, the requirement is to plot a dot instead of color fill on a map. If the existing geospatial data is only available in the form of area geometry (for example, a boundary), use **GeoGetPolygonCenter()** to retrieve a pair of longitude and latitude for the center of area.

Syntax:

```
GeoGetPolygonCenter(field_name)
```

Return data type: string

Arguments:

Arguments

Argument	Description
field_name	A field or expression referring to a field containing the geometry to be represented. This could be either a point (or set of points) giving longitude and latitude, or an area.

*Do not use the **Group by** clause in the data load editor with this and other non-aggregating geospatial functions, because this will cause an error on load.*

GeoInvProjectGeometry

GeoInvProjectGeometry() is used to aggregate a geometry into an area and apply the inverse of a projection.

Syntax:

```
GeoInvProjectGeometry(type, field_name)
```

Return data type: string

Arguments:

Arguments

Argument	Description
type	Projection type used in transforming the geometry of the map. This can take one of two values: 'unit', (default), which results in a 1:1 projection, or 'mercator', which uses the standard Mercator projection.
field_name	A field or expression referring to a field containing the geometry to be represented. This could be either a point (or set of points) giving longitude and latitude, or an area.

Example:

Scripting example

Example	Result
In a `Load` statement: `GeoInvProjectGeometry ('mercator',AreaPolygon) as InvProjectGeometry`	The geometry loaded as **AreaPolygon** is transformed using the inverse transformation of the Mercator projection and stored as **InvProjectGeometry** for use in visualizations.

GeoMakePoint

GeoMakePoint() is used in scripts and chart expressions to create and tag a point with latitude and longitude. GeoMakePoint returns points in the order of longitude and latitude.

Syntax:

```
GeoMakePoint(lat_field_name, lon_field_name)
```

Return data type: string, formatted [longitude, latitude]

Arguments:

Arguments

Argument	Description
lat_field_name	A field or expression referring to a field representing the latitude of the point.
lon_field_name	A field or expression referring to a field representing the longitude of the point.

*Do not use the **Group by** clause in the data load editor with this and other non-aggregating geospatial functions, because this will cause an error on load.*

GeoProject

GeoProject() is used in scripts and chart expressions to apply a projection to a geometry.

Syntax:

```
GeoProject(type, field_name)
```

Return data type: string

Arguments:

Arguments

Argument	Description
type	Projection type used in transforming the geometry of the map. This can take one of two values: 'unit', (default), which results in a 1:1 projection, or 'mercator', which uses the web Mercator projection.
field_name	A field or expression referring to a field containing the geometry to be represented. This could be either a point (or set of points) giving longitude and latitude, or an area.

*Do not use the **Group by** clause in the data load editor with this and other non-aggregating geospatial functions, because this will cause an error on load.*

Example:

Script examples

Example	Result
In a `Load` statement: `GeoProject('mercator',Area) as GetProject`	The Mercator projection is applied to the geometry loaded as **Area**, and the result is stored as **GetProject**.

GeoProjectGeometry

GeoProjectGeometry() is used to aggregate a geometry into an area and apply a projection.

Syntax:

```
GeoProjectGeometry(type, field_name)
```

Return data type: string

Arguments:

Arguments

Argument	Description
type	Projection type used in transforming the geometry of the map. This can take one of two values: 'unit', (default), which results in a 1:1 projection, or 'mercator', which uses the web Mercator projection.
field_name	A field or expression referring to a field containing the geometry to be represented. This could be either a point (or set of points) giving longitude and latitude, or an area.

Example:

Example	Result
In a `Load` statement: `GeoProjectGeometry ('mercator',AreaPolygon) as ProjectGeometry`	The geometry loaded as **AreaPolygon** is transformed using the Mercator projection and stored as **ProjectGeometry** for use in visualizations.

GeoReduceGeometry

GeoReduceGeometry() is used to reduce the number of vertices of a geometry, and to aggregate a number of areas into one area, but still displaying the boundary lines from the individual areas.

Syntax:

```
GeoReduceGeometry(field_name[, value])
```

Return data type: string

Arguments:

Arguments

Argument	Description
field_name	A field or expression referring to a field containing the geometry to be represented. This could be either a point (or set of points) giving longitude and latitude, or an area.

Argument	Description
value	The amount of reduction to apply to the geometry. The range is from 0 to 1, with 0 representing no reduction and 1 representing maximal reduction of vertices. *Using a value of 0.9 or higher with a complex data set can reduce the number of vertices to a level where the visual representation is inaccurate.*

GeoReduceGeometry() also performs a similar function to, **GeoAggrGeometry()** in that it aggregates a number of areas into one area. The difference being that individual boundary lines from the pre-aggregation data are displayed on the map if you use **GeoReduceGeometry()**.

As **GeoReduceGeometry()** is an aggregating function, if you use it in the script a **LOAD** statement with a **Group by** clause is required.

Examples:
This example loads a KML file with area data, and then loads a table with the reduced and aggregated area data.

```
[MapSource]:
LOAD [world.Name],
       [world.Point],
       [world.Area]
 FROM [lib://Downloads/world.kml]
(kml, Table is [World.shp/Features]);

Map:
LOAD world.Name,
     GeoReduceGeometry(world.Area,0.5) as [ReducedArea]
resident MapSource Group By world.Name;

Drop Table MapSource;
```

5.15 Interpretation functions

The interpretation functions evaluate the contents of input text fields or expressions, and impose a specified data format on the resulting numeric value. With these functions, you can specify the format of the number, in accordance with its data type, including attributes such as: decimal separator, thousands separator,and date format.

The interpretation functions all return a dual value with both the string and the number value, but can be thought of as performing a string-to-number conversion. The functions take the text value of the input expression and generate a number representing the string.

In contrast, the formatting functions do the opposite: they take numeric expressions and evaluate them as strings, specifying the display format of the resulting text.

If no interpretation functions are used, Qlik Sense interprets the data as a mix of numbers, dates, times, time stamps and strings, using the default settings for number format, date format, and time format, defined by script variables and by the operating system.

All interpretation functions can be used in both data load scripts and chart expressions.

> *All number representations are given with a decimal point as the decimal separator.*

Interpretation functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Date#

Date# evaluates an expression as a date in the format specified in the second argument, if supplied. If the format code is omitted, the default date format set in the operating system is used.

```
Date# (page 839)(text[, format])
```

Interval#

Interval#() evaluates a text expression as a time interval in the format set in the operating system, by default, or in the format specified in the second argument, if supplied.

```
Interval# (page 840)(text[, format])
```

Money#

Money#() converts a text string to a money value, in the format set in the load script or the operating system, unless a format string is supplied. Custom decimal and thousand separator symbols are optional parameters.

```
Money# (page 840)(text[, format[, dec_sep[, thou_sep ] ] ])
```

Num#

Num#() interprets a text string as a numerical value, that is it converts the input string to a number using the format specified in the second parameter. If the second parameter is omitted, it uses the decimal and thousand separators set in the data load script. Custom decimal and thousand separator symbols are optional parameters.

```
Num# (page 842)(text[ , format[, dec_sep[ , thou_sep]]])
```

Text

Text() forces the expression to be treated as text, even if a numeric interpretation is possible.

```
Text(expr)
```

Time#

Time#() evaluates an expression as a time value, in the time format set in the data load script or the operating system, unless a format string is supplied..

```
Time# (page 843)(text[, format])
```

Timestamp#

Timestamp#() evaluates an expression as a date and time value, in the timestamp format set in the data load script or the operating system, unless a format string is supplied.

```
Timestamp# (page 844)(text[, format])
```

See also:

Formatting functions (page 803)

Date#

Date# evaluates an expression as a date in the format specified in the second argument, if supplied.

Syntax:

```
Date#(text[, format])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
text	The text string to be evaluated.
format	String describing the format of the text string to be evaluated. If omitted, the date format set in the system variables in the data load script, or the operating system is used.

Examples and results:

The following example uses the date format **M/D/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of the data load script.

Add this example script to your app and run it.

```
Load *,
Num(Date#(StringDate)) as Date;
LOAD * INLINE [
StringDate
8/7/97
8/6/1997
]
```

If you create a table with **StringDate** and **Date** as dimensions, the results are as follows:

Results

StringDate	Date
8/7/97	35649
8/6/1997	35648

Interval#

Interval#() evaluates a text expression as a time interval in the format set in the operating system, by default, or in the format specified in the second argument, if supplied.

Syntax:

```
Interval#(text[, format])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
text	The text string to be evaluated.
format	String describing the expected input format to use when converting the string to a numeric interval. If omitted, the short date format, time format, and decimal separator set in the operating system are used.

The **interval#** function converts a text time interval to a numeric equivalent.

Examples and results:
The examples below assume the following operating system settings:

- Short date format: YY-MM-DD
- Time format: M/D/YY
- Number decimal separator: .

Results

Example	Result
`Interval#( A, 'D hh:mm' )` where A='1 09:00'	1.375

Money#

Money#() converts a text string to a money value, in the format set in the load script or the operating system, unless a format string is supplied. Custom decimal and thousand separator symbols are optional parameters.

Syntax:

```
Money#(text[, format[, dec_sep [, thou_sep ] ] ])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
text	The text string to be evaluated.
format	String describing the expected input format to use when converting the string to a numeric interval. If omitted, the money format set in the operating system is used.
dec_sep	String specifying the decimal number separator. If omitted, the MoneyDecimalSep value set in the data load script is used.
thou_sep	String specifying the thousands number separator. If omitted, the MoneyThousandSep value set in the data load script is used.

The **money#** function generally behaves just like the **num#** function but takes its default values for decimal and thousand separator from the script variables for money format or the system settings for currency.

Examples and results:
The examples below assume the two following operating system settings:

- Money format default setting 1: kr # ##0,00
- Money format default setting 2: $ #,##0.00

```
Money#(A , '# ##0,00 kr' )
```
where A=35 648,37 kr

Results

Results	Setting 1	Setting 2
String	35 648.37 kr	35 648.37 kr
Number	35648.37	3564837

```
Money#( A, ' $#', '.', ',' )
```
where A= $35,648.37

Results

Results	Setting 1	Setting 2
String	$35,648.37	$35,648.37
Number	35648.37	35648.37

Num#

Num#() interprets a text string as a numerical value, that is it converts the input string to a number using the format specified in the second parameter. If the second parameter is omitted, it uses the decimal and thousand separators set in the data load script. Custom decimal and thousand separator symbols are optional parameters.

Syntax:

```
Num#(text[, format[, dec_sep [, thou_sep ] ] ])
```

Return data type: dual

The **Num#()** function returns a dual value with both the string and the numeric value. The function takes the textual representation of the input expression and generates a number. It does not change the format of the number: the output is formatted in the same way as the input.

Arguments:

Arguments

Argument	Description
text	The text string to be evaluated.
format	String specifying the number format used in the first parameter. If omitted, the decimal and thousand separators that are set in the data load script are used.
dec_sep	String specifying the decimal number separator. If omitted, the value of the variable DecimalSep that is set in the data load script is used.
thou_sep	String specifying the thousands number separator. If omitted, the value of the variable ThousandSep that is set in the data load script is used.

Examples and results:
The following table shows the result of *Num#(A, '#', '.' , ',')* for different values of A.

Results

A	String representation	Numeric value (here displayed with decimal point)
35,648.31	35,648.31	35648.31
35 648.312	35 648.312	35648.312
35.648,3123	35.648,3123	-
35 648,31234	35 648,31234	-

Text

Text() forces the expression to be treated as text, even if a numeric interpretation is possible.

Syntax:

```
Text (expr)
```

Return data type: dual

Example:

```
Text( A )
```
where A=1234

Results

String	Number
1234	-

Example:

```
Text( pi( ) )
```

Results

String	Number
3.1415926535898	-

Time#

Time#() evaluates an expression as a time value, in the time format set in the data load script or the operating system, unless a format string is supplied..

Syntax:

```
time#(text[, format])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
text	The text string to be evaluated.
format	String describing the format of the text string to be evaluated. If omitted, the short date format, time format, and decimal separator set in the operating system is used.

Example:

- Time format default setting 1: hh:mm:ss
- Time format default setting 2: hh.mm.ss

```
time#( A )
```
where A=09:00:00

Results

Results	Setting 1	Setting 2
String:	09:00:00	09:00:00
Number:	0.375	-

Example:

- Time format default setting 1: hh:mm:ss
- Time format default setting 2: hh.mm.ss

```
time#( A, 'hh.mm' )
```
where A=09.00

Results

Results	Setting 1	Setting 2
String:	09.00	09.00
Number:	0.375	0.375

Timestamp#

Timestamp#() evaluates an expression as a date and time value, in the timestamp format set in the data load script or the operating system, unless a format string is supplied.

Syntax:

```
timestamp#(text[, format])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
text	The text string to be evaluated.
format	String describing the format of the text string to be evaluated. If omitted, the short date format, time format, and decimal separator set in the operating system is used. ISO 8601 is supported for timestamps.

Example:

The following example uses the date format **M/D/YYYY**. The date format is specified in the **SET DateFormat** statement at the top of the data load script.

Add this example script to your app and run it.

```
Load *,
Timestamp(Timestamp#(String)) as TS;
LOAD * INLINE [
String
2015-09-15T12:13:14
1952-10-16T13:14:00+0200
1109-03-01T14:15
];
```

If you create a table with **String** and **TS** as dimensions, the results are as follows:

Results

String	TS
2015-09-15T12:13:14	9/15/2015 12:13:14 PM
1952-10-16T13:14:00+0200	10/16/1952 11:14:00 AM
1109-03-01T14:15	3/1/1109 2:15:00 PM

5.16 Inter-record functions

Inter-record functions are used:

- In the data load script, when a value from previously loaded records of data is needed for the evaluation of the current record.
- In a chart expression, when another value from the data set of a visualization is needed.

Sorting on y-values in charts or sorting by expression columns in straight tables is not allowed when chart inter-record functions are used in any of the chart's expressions. These sort alternatives are therefore automatically disabled.

Self-referencing expression definitions can only reliably be made in tables with fewer than 100 rows, but this may vary depending on the hardware that the Qlik engine is running on.

Row functions

These functions can only be used in chart expressions.

Above

Above() evaluates an expression at a row above the current row within a column segment in a table. The row for which it is calculated depends on the value of **offset**, if present, the default being the row directly above. For charts other than tables, **Above()** evaluates for the row above the current row in the chart's straight table equivalent.

```
Above - chart function([TOTAL [<fld{,fld}>]] expr [ , offset [,count]])
```

Below

Below() evaluates an expression at a row below the current row within a column segment in a table. The row for which it is calculated depends on the value of **offset**, if present, the default being the row directly below. For charts other than tables, **Below()** evaluates for the row below the current column in the chart's straight table equivalent.

```
Below - chart function([TOTAL[<fld{,fld}>]] expression [ , offset [,count ]])
```

Bottom

Bottom() evaluates an expression at the last (bottom) row of a column segment in a table. The row for which it is calculated depends on the value of **offset**, if present, the default being the bottom row. For charts other than tables, the evaluation is made on the last row of the current column in the chart's straight table equivalent.

```
Bottom - chart function([TOTAL[<fld{,fld}>]] expr [ , offset [,count ]])
```

Top

Top() evaluates an expression at the first (top) row of a column segment in a table. The row for which it is calculated depends on the value of **offset**, if present, the default being the top row. For charts other than tables, the**Top()** evaluation is made on the first row of the current column in the chart's straight table equivalent.

```
Top - chart function([TOTAL [<fld{,fld}>]] expr [ , offset [,count ]])
```

NoOfRows

NoOfRows() returns the number of rows in the current column segment in a table. For bitmap charts, **NoOfRows()** returns the number of rows in the chart's straight table equivalent.

```
NoOfRows - chart function([TOTAL])
```

Column functions

These functions can only be used in chart expressions.

Column

Column() returns the value found in the column corresponding to **ColumnNo** in a straight table, disregarding dimensions. For example **Column(2)** returns the value of the second measure column.

```
Column - chart function(ColumnNo)
```

Dimensionality

Dimensionality() returns the number of dimensions for the current row. In the case of pivot tables, the function returns the total number of dimension columns that have non-aggregation content, that is, do not contain partial sums or collapsed aggregates.

```
Dimensionality - chart function ( )
```

Secondarydimensionality

SecondaryDimensionality() returns the number of dimension pivot table rows that have non-aggregation content, that is, do not contain partial sums or collapsed aggregates. This function is the equivalent of the **dimensionality()** function for horizontal pivot table dimensions.

```
SecondaryDimensionality - chart function ( )
```

Field functions

FieldIndex

FieldIndex() returns the position of the field value **value** in the field **field_name** (by load order).

```
FieldIndex(field_name , value)
```

FieldValue

FieldValue() returns the value found in position **elem_no** of the field **field_name** (by load order).

```
FieldValue(field_name , elem_no)
```

FieldValueCount

FieldValueCount() is an **integer** function that returns the number of distinct values in a field.

```
FieldValueCount(field_name)
```

Pivot table functions

These functions can only be used in chart expressions.

After

After() returns the value of an expression evaluated with a pivot table's dimension values as they appear in the column after the current column within a row segment in the pivot table.

```
After - chart function([TOTAL] expression [ , offset [,n]])
```

Before

Before() returns the value of an expression evaluated with a pivot table's dimension values as they appear in the column before the current column within a row segment in the pivot table.

```
Before - chart function([TOTAL] expression [ , offset [,n]])
```

First

First() returns the value of an expression evaluated with a pivot table's dimension values as they appear in the first column of the current row segment in the pivot table. This function returns NULL in all chart types except pivot tables.

```
First - chart function([TOTAL] expression [ , offset [,n]])
```

Last

Last() returns the value of an expression evaluated with a pivot table's dimension values as they appear in the last column of the current row segment in the pivot table. This function returns NULL in all chart types except pivot tables.

```
Last - chart function([TOTAL] expression [ , offset [,n]])
```

ColumnNo

ColumnNo() returns the number of the current column within the current row segment in a pivot table. The first column is number 1.

```
ColumnNo - chart function([TOTAL])
```

NoOfColumns

NoOfColumns() returns the number of columns in the current row segment in a pivot table.

```
NoOfColumns - chart function([TOTAL])
```

Inter-record functions in the data load script

Exists

Exists() determines whether a specific field value has already been loaded into the field in the data load script. The function returns TRUE or FALSE, so can be used in the **where** clause of a **LOAD** statement or an **IF** statement.

```
Exists (field_name [, expr])
```

LookUp

Lookup() looks into a table that is already loaded and returns the value of **field_name** corresponding to the first occurrence of the value **match_field_value** in the field **match_field_name**. The table can be the current table or another table previously loaded.

```
LookUp (field_name, match_field_name, match_field_value [, table_name])
```

Peek

Peek() returns the value of a field in a table for a row that has already been loaded. The row number can be specified, as can the table. If no row number is specified, the last previously loaded record will be used.

```
Peek (field_name[, row_no[, table_name ] ])
```

Previous

Previous() finds the value of the **expr** expression using data from the previous input record that has not been discarded because of a **where** clause. In the first record of an internal table, the function will return NULL.

```
Previous (page 882)(expr)
```

See also:

Range functions (page 901)

Above - chart function

Above() evaluates an expression at a row above the current row within a column segment in a table. The row for which it is calculated depends on the value of **offset**, if present, the default being the row directly above. For charts other than tables, **Above()** evaluates for the row above the current row in the chart's straight table equivalent.

Syntax:

```
Above([TOTAL] expr [ , offset [,count]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
offset	Specifying an offsetn, greater than 0, moves the evaluation of the expression n rows further up from the current row. Specifying an offset of 0 will evaluate the expression on the current row. Specifying a negative offset number makes the Above function work like the Below function with the corresponding positive offset number.
count	By specifying a third argument **count** greater than 1, the function will return a range of **count** values, one for each of **count** table rows counting upwards from the original cell. In this form, the function can be used as an argument to any of the special range functions. *Range functions (page 901)*
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

On the first row of a column segment, a NULL value is returned, as there is no row above it.

A column segment is defined as a consecutive subset of cells having the same values for the dimensions in the current sort order. Inter-record chart functions are computed in the column segment excluding the right-most dimension in the equivalent straight table chart. If there is only one dimension in the chart, or if the TOTAL qualifier is specified, the expression evaluates across full table.

If the table or table equivalent has multiple vertical dimensions, the current column segment will include only rows with the same values as the current row in all dimension columns, except for the column showing the last dimension in the inter-field sort order.

Limitations:

Recursive calls will return NULL.

Examples and results:

Example 1:

Table visualization for Example 1

Customer	Sum([Sales])	Above(Sum(Sales))	Sum(Sales)+Above(Sum(Sales))	Above offset 3	Higher?
	2566	-	-	-	-
Astrida	587	-	-	-	-
Betacab	539	587	1126	-	-
Canutility	683	539	1222	-	Higher
Divadip	757	683	1440	1344	Higher

In the screenshot of the table shown in this example, the table visualization is created from the dimension **Customer** and the measures: `Sum(Sales)` and `Above(Sum(Sales))`.

The column Above(Sum(Sales)) returns NULL for the **Customer** row containing **Astrida**, because there is no row above it. The result for the row **Betacab** shows the value of Sum(Sales) for **Astrida**, the result for **Canutility** shows the value for **Sum(Sales)** for **Betacab**, and so on.

For the column labeled Sum(Sales)+Above(Sum(Sales)), the row for **Betacab** shows the result of the addition of the **Sum(Sales)** values for the rows **Betacab** + **Astrida** (539+587). The result for the row **Canutility** shows the result of the addition of **Sum(Sales)** values for **Canutility** + **Betacab** (683+539).

The measure labeled Above offset 3 created using the expression `Sum(Sales)+Above(Sum(Sales), 3)` has the argument **offset**, set to 3, and has the effect of taking the value in the row three rows above the current row. It adds the **Sum(Sales)** value for the current **Customer** to the value for the **Customer**three rows above. The values returned for the first three **Customer** rows are null.

The table also shows more complex measures: one created from `Sum(Sales)+Above(Sum(Sales))` and one labeled **Higher?**, which is created from `IF(Sum(Sales)>Above(Sum(Sales)), 'Higher')`.

This function can also be used in charts other than tables, for example bar charts.

For other chart types, convert the chart to the straight table equivalent so you can easily interpret which row the function relates to.

Example 2:

In the screenshots of tables shown in this example, more dimensions have been added to the visualizations: **Month** and **Product**. For charts with more than one dimension, the results of expressions containing the **Above**, **Below**, **Top**, and **Bottom** functions depend on the order in which the column dimensions are sorted by Qlik Sense. Qlik Sense evaluates the functions based on the column segments that result from the dimension that is sorted last. The column sort order is controlled in the properties panel under **Sorting** and is not necessarily the order in which the columns appear in a table.

In the following screenshot of table visualization for Example 2, the last-sorted dimension is **Month**, so the **Above** function evaluates based on months. There is a series of results for each **Product** value for each month (**Jan** to **Aug**) - a column segment. This is followed by a series for the next column segment: for each **Month** for the next **Product**. There will be a column segment for each **Customer** value for each **Product**.

Table visualization for Example 2

Customer	Product	Month	Sum([Sales])	Above(Sum(Sales))
			2566	-
Astrida	AA	Jan	46	-
Astrida	AA	Feb	60	46
Astrida	AA	Mar	70	60
Astrida	AA	Apr	13	70
Astrida	AA	May	78	13
Astrida	AA	Jun	20	78
Astrida	AA	Jul	45	20
Astrida	AA	Aug	65	45

Example 3:

In the screenshot of table visualization for Example 3, the last sorted dimension is **Product**. This is done by moving the dimension Product to position 3 in the Sorting tab in the properties panel. The **Above** function is evaluated for each **Product**, and because there are only two products, **AA** and **BB**, there is only one non-null result in each series. In row **BB** for the month **Jan**, the value for **Above(Sum(Sales))**, is 46. For row **AA**, the value is null. The value in each row **AA** for any month will always be null, as there is no value of **Product** above AA. The second series is evaluated on **AA** and **BB** for the month **Feb**, for the **Customer** value, **Astrida**. When all the months have been evaluated for **Astrida**, the sequence is repeated for the second **Customer**Betacab, and so on.

Table visualization for Example 3

Customer	Product	Month	Sum([Sales])	Above(Sum(Sales))
			2566	-
Astrida	AA	Jan	46	-
Astrida	BB	Jan	46	46
Astrida	AA	Feb	60	-
Astrida	BB	Feb	60	60
Astrida	AA	Mar	70	-
Astrida	BB	Mar	70	70
Astrida	AA	Apr	13	-
Astrida	BB	Apr	13	13

Example 4

Example 4:	Result
The Above function can be used as input to the range functions. For example: `RangeAvg (Above(Sum(Sales),1,3))`.	In the arguments for the Above() function, offset is set to 1 and count is set to 3. The function finds the results of the expressionSum(Sales) on the three rows immediately above the current row in the column segment (where there is a row). These three values are used as input to the RangeAvg() function, which finds the average of the values in the supplied range of numbers. A table with Customer as dimension gives the following results for the RangeAvg() expression. Astrida - Betacab 587 Canutility 563 Divadip: 603

Data used in examples:

```
Monthnames:
LOAD *, Dual(MonthText,MonthNumber) as Month INLINE [
MonthText, MonthNumber
Jan, 1
Feb, 2
Mar, 3
Apr, 4
May, 5
Jun, 6
Jul, 7
Aug, 8
Sep, 9
Oct, 10
Nov, 11
Dec, 12
];

Sales2013:
Crosstable (MonthText, Sales) LOAD * inline [
Customer|Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec
Astrida|46|60|70|13|78|20|45|65|78|12|78|22
Betacab|65|56|22|79|12|56|45|24|32|78|55|15
Canutility|77|68|34|91|24|68|57|36|44|90|67|27
Divadip|57|36|44|90|67|27|57|68|47|90|80|94
] (delimiter is '|');
```

See also:

Below - chart function (page 853)
Bottom - chart function (page 856)
Top - chart function (page 884)
RangeAvg (page 904)

Below - chart function

Below() evaluates an expression at a row below the current row within a column segment in a table. The row for which it is calculated depends on the value of **offset**, if present, the default being the row directly below. For charts other than tables, **Below()** evaluates for the row below the current column in the chart's straight table equivalent.

Syntax:

```
Below([TOTAL] expr [ , offset [,count ]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
offset	Specifying an **offset**n, greater than 1 moves the evaluation of the expression n rows further down from the current row. Specifying an offset of 0 will evaluate the expression on the current row. Specifying a negative offset number makes the **Below** function work like the **Above** function with the corresponding positive offset number.
count	By specifying a third parameter **count** greater than 1, the function will return a range of **count** values, one for each of **count** table rows counting downwards from the original cell. In this form, the function can be used as an argument to any of the special range functions. *Range functions (page 901)*
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

On the last row of a column segment, a NULL value is returned, as there is no row below it.

A column segment is defined as a consecutive subset of cells having the same values for the dimensions in the current sort order. Inter-record chart functions are computed in the column segment excluding the right-most dimension in the equivalent straight table chart. If there is only one dimension in the chart, or if the TOTAL qualifier is specified, the expression evaluates across full table.

If the table or table equivalent has multiple vertical dimensions, the current column segment will include only rows with the same values as the current row in all dimension columns, except for the column showing the last dimension in the inter-field sort order.

Limitations:

Recursive calls will return NULL.

Examples and results:

Example 1:

Table visualization for Example 1

Customer	Sum([Sales])	Below(Sum(Sales))	Sum(Sales)+Below(Sum(Sales))	Below + Offset 3	Higher
	2566	-	-	-	-
Astrida	587	539	1126	1344	Higher
Betacab	539	683	1222	-	-
Canutility	683	757	1440	-	-
Divadip	757	-	-	-	-

In the table shown in screenshot for Example 1, the table visualization is created from the dimension **Customer** and the measures: `Sum(Sales)` and `Below(Sum(Sales))`.

The column **Below(Sum(Sales))** returns NULL for the **Customer** row containing **Divadip**, because there is no row below it. The result for the row **Canutility** shows the value of Sum(Sales) for **Divadip**, the result for **Betacab** shows the value for **Sum(Sales)** for **Canutility**, and so on.

The table also shows more complex measures, which you can see in the columns labeled: `Sum(Sales)+Below(Sum(Sales))`, **Below +Offset 3**, and **Higher?**. These expressions work as described in the following paragraphs.

For the column labeled **Sum(Sales)+Below(Sum(Sales))**, the row for **Astrida** shows the result of the addition of the **Sum(Sales)** values for the rows **Betacab** + **Astrida** (539+587). The result for the row **Betacab** shows the result of the addition of **Sum(Sales)** values for **Canutility** + **Betacab** (539+683).

The measure labeled **Below +Offset 3** created using the expression `Sum(Sales)+Below(Sum(Sales), 3)` has the argument **offset**, set to 3, and has the effect of taking the value in the row three rows below the current row. It adds the **Sum(Sales)** value for the current **Customer** to the value from the **Customer** three rows below. The values for the lowest three **Customer** rows are null.

The measure labeled **Higher?** is created from the expression:`IF(Sum(Sales)>Below(Sum(Sales)), 'Higher')`. This compares the values of the current row in the measure **Sum(Sales)** with the row below it. If the current row is a greater value, the text "Higher" is output.

This function can also be used in charts other than tables, for example bar charts.

For other chart types, convert the chart to the straight table equivalent so you can easily interpret which row the function relates to.

For charts with more than one dimension, the results of expressions containing the **Above**, **Below**, **Top**, and **Bottom** functions depend on the order in which the column dimensions are sorted by Qlik Sense. Qlik Sense evaluates the functions based on the column segments that result from the dimension that is sorted last. The column sort order is controlled in the properties panel under **Sorting** and is not necessarily the order in which the columns appear in a table.Please refer to Example: 2 in the **Above** function for further details.

Example 2

Example 2:	**Result**
The **Below** function can be used as input to the range functions. For example: `RangeAvg (Below (Sum(Sales),1,3))`.	In the arguments for the **Below()** function, offset is set to 1 and count is set to 3. The function finds the results of the expression**Sum(Sales)** on the three rows immediately below the current row in the column segment (where there is a row). These three values are used as input to the RangeAvg() function, which finds the average of the values in the supplied range of numbers. A table with **Customer** as dimension gives the following results for the RangeAvg() expression.
	Astrida 659.67 Betacab 720 Canutility 757 Divadip: -

Data used in examples:

```
Monthnames:
LOAD *, Dual(MonthText,MonthNumber) as Month INLINE [
MonthText, MonthNumber
Jan, 1
```

```
Feb, 2
Mar, 3
Apr, 4
May, 5
Jun, 6
Jul, 7
Aug, 8
Sep, 9
Oct, 10
Nov, 11
Dec, 12
];

Sales2013:
Crosstable (MonthText, Sales) LOAD * inline [
Customer|Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec
Astrida|46|60|70|13|78|20|45|65|78|12|78|22
Betacab|65|56|22|79|12|56|45|24|32|78|55|15
Canutility|77|68|34|91|24|68|57|36|44|90|67|27
Divadip|57|36|44|90|67|27|57|68|47|90|80|94
] (delimiter is '|');
```

See also:

- *Above - chart function (page 848)*
- *Bottom - chart function (page 856)*
- *Top - chart function (page 884)*
- *RangeAvg (page 904)*

Bottom - chart function

Bottom() evaluates an expression at the last (bottom) row of a column segment in a table. The row for which it is calculated depends on the value of **offset**, if present, the default being the bottom row. For charts other than tables, the evaluation is made on the last row of the current column in the chart's straight table equivalent.

Syntax:

```
Bottom([TOTAL] expr [ , offset [,count ]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.

Argument	Description
offset	Specifying an **offset**n greater than 1 moves the evaluation of the expression up n rows above the bottom row. Specifying a negative offset number makes the **Bottom** function work like the **Top** function with the corresponding positive offset number.
count	By specifying a third parameter **count** greater than 1, the function will return not one but a range of **count** values, one for each of the last **count** rows of the current column segment. In this form, the function can be used as an argument to any of the special range functions. *Range functions (page 901)*
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

A column segment is defined as a consecutive subset of cells having the same values for the dimensions in the current sort order. Inter-record chart functions are computed in the column segment excluding the right-most dimension in the equivalent straight table chart. If there is only one dimension in the chart, or if the TOTAL qualifier is specified, the expression evaluates across full table.

If the table or table equivalent has multiple vertical dimensions, the current column segment will include only rows with the same values as the current row in all dimension columns, except for the column showing the last dimension in the inter-field sort order.

Limitations:

Recursive calls will return NULL.

Examples and results:

Table visualization for Example 1

Customer	Sum([Sales])	Bottom(Sum(Sales))	Sum(Sales)+Bottom(Sum(Sales))	Bottom offset 3
	2566	**757**	**3323**	**3105**
Astrida	587	757	1344	1126
Betacab	539	757	1296	1078
Canutility	683	757	1440	1222
Divadip	757	757	1514	1296

In the screenshot of the table shown in this example, the table visualization is created from the dimension **Customer** and the measures: `Sum(Sales)` and `Bottom(Sum(Sales))`.

The column **Bottom(Sum(Sales))** returns 757 for all rows because this is the value of the bottom row: **Divadip**.

The table also shows more complex measures: one created from `Sum(Sales)+Bottom(Sum(Sales))` and one labeled **Bottom offset 3**, which is created using the expression `Sum(Sales)+Bottom(Sum(Sales), 3)` and has the argument **offset** set to 3. It adds the **Sum(Sales)** value for the current row to the value from the third row from the bottom row, that is, the current row plus the value for **Betacab**.

Example: 2

In the screenshots of tables shown in this example, more dimensions have been added to the visualizations: **Month** and **Product**. For charts with more than one dimension, the results of expressions containing the **Above**, **Below**, **Top**, and **Bottom** functions depend on the order in which the column dimensions are sorted by Qlik Sense. Qlik Sense evaluates the functions based on the column segments that result from the dimension that is sorted last. The column sort order is controlled in the properties panel under **Sorting** and is not necessarily the order in which the columns appear in a table.

In the first table, the expression is evaluated based on **Month**, and in the second table it is evaluated based on **Product**. The measure **End value** contains the expression `Bottom(Sum(Sales))`. The bottom row for **Month** is Dec, and the value for Dec both the values of **Product** shown in the screenshot is 22. (Some rows have been edited out of the screenshot to save space.)

First table for Example 2. The value of Bottom for the End value measure based on Month (Dec).

Customer	Product	Month	Sum(Sales)	End value
			2566	-
Astrida	AA	Jan	46	22
Astrida	AA	Feb	60	22
Astrida	AA	Mar	70	22
Astrida	AA	Sep	78	22
Astrida	AA	Oct	12	22
Astrida	AA	Nov	78	22
Astrida	AA	Dec	22	22
Astrida	BB	Jan	46	22

Second table for Example 2. The value of Bottom for the End value measure based on Product (BB for Astrida).

Customer	Product	Month	Sum(Sales)	End value
			2566	-
Astrida	AA	Jan	46	46
Astrida	BB	Jan	46	46
Astrida	AA	Feb	60	60
Astrida	BB	Feb	60	60
Astrida	AA	Mar	70	70
Astrida	BB	Mar	70	70
Astrida	AA	Apr	13	13
Astrida	BB	Apr	13	13

Please refer to Example: 2 in the **Above** function for further details.

Example 3

Example: 3	**Result**
The **Bottom** function can be used as input to the range functions. For example: `RangeAvg (Bottom (Sum(Sales),1,3))`.	In the arguments for the **Bottom()** function, offset is set to 1 and count is set to 3. The function finds the results of the expression**Sum(Sales)** on the three rows starting with the row above the bottom row in the column segment (because offset=1), and the two rows above that (where there is a row). These three values are used as input to the RangeAvg() function, which finds the average of the values in the supplied range of numbers. A table with **Customer** as dimension gives the following results for the RangeAvg() expression.
	Astrida 659.67 Betacab 659.67 Canutility 659.67 Divadip: 659.67

```
Monthnames:
LOAD *, Dual(MonthText,MonthNumber) as Month INLINE [
MonthText, MonthNumber
Jan, 1
Feb, 2
Mar, 3
Apr, 4
May, 5
Jun, 6
Jul, 7
Aug, 8
```

```
Sep, 9
Oct, 10
Nov, 11
Dec, 12
];

Sales2013:
Crosstable (MonthText, Sales) LOAD * inline [
Customer|Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec
Astrida|46|60|70|13|78|20|45|65|78|12|78|22
Betacab|65|56|22|79|12|56|45|24|32|78|55|15
Canutility|77|68|34|91|24|68|57|36|44|90|67|27
Divadip|57|36|44|90|67|27|57|68|47|90|80|94
] (delimiter is '|');
```

See also:

Top - chart function (page 884)

Column - chart function

Column() returns the value found in the column corresponding to **ColumnNo** in a straight table, disregarding dimensions. For example **Column(2)** returns the value of the second measure column.

Syntax:

```
Column(ColumnNo)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
ColumnNo	Column number of a column in the table containing a measure. *The Column() function disregards dimension columns.*

Limitations:

If **ColumnNo** references a column for which there is no measure, a NULL value is returned.

Recursive calls will return NULL.

Examples and results:

Example: Percentage total sales

Customer	Product	UnitPrice	UnitSales	Order Value	Total Sales Value	% Sales
A	AA	15	10	150	505	29.70
A	AA	16	4	64	505	12.67
A	BB	9	9	81	505	16.04
B	BB	10	5	50	505	9.90
B	CC	20	2	40	505	7.92
B	DD	25	-	0	505	0.00
C	AA	15	8	120	505	23.76
C	CC	19	-	0	505	0.00

Example: Percentage of sales for selected customer

Customer	Product	UnitPrice	UnitSales	Order Value	Total Sales Value	% Sales
A	AA	15	10	150	295	50.85
A	AA	16	4	64	295	21.69
A	BB	9	9	81	295	27.46

Examples and results

Examples	Results
Order Value is added to the table as a measure with the expression: `Sum (UnitPrice*UnitSales)`. Total Sales Value is added as a measure with the expression: `Sum(TOTAL UnitPrice*UnitSales)` % Sales is added as a measure with the expression `100*Column(1)/Column(2)`	The result of Column(1) is taken from the column Order Value, because this is the first measure column. The result of Column(2) is taken from Total Sales Value, because this is the second measure column. See the results in the column % Sales in the example *Percentage total sales (page 861)*.
Make the selection Customer A.	The selection changes the Total Sales Value, and therefore the %Sales. See the example *Percentage of sales for selected customer (page 861)*.

Data used in examples:

```
ProductData:
LOAD * inline [
Customer|Product|UnitSales|UnitPrice
Astrida|AA|4|16
Astrida|AA|10|15
Astrida|BB|9|9
Betacab|BB|5|10
Betacab|CC|2|20
Betacab|DD||25
Canutility|AA|8|15
Canutility|CC||19
] (delimiter is '|');
```

Dimensionality - chart function

Dimensionality() returns the number of dimensions for the current row. In the case of pivot tables, the function returns the total number of dimension columns that have non-aggregation content, that is, do not contain partial sums or collapsed aggregates.

Syntax:

```
Dimensionality ( )
```

Return data type: integer

Limitations:

This function is only available in charts. For all chart types, except pivot table, it will return the number of dimensions in all rows except the total, which will be 0.

Example: Chart expression using Dimensionality

Example: Chart expression

The Dimensionality() function can be used with a pivot table as a chart expression where you want to apply different cell formatting depending on the number of dimensions in a row that has non-aggregated data. This example uses the Dimensionality() function to apply a background color to table cells that match a given condition.

Load script

Load the following data as an inline load in the data load editor to create the chart expression example below.

```
ProductSales:

Load * inline [
Country,Product,Sales,Budget
Sweden,AA,100000,50000
Germany,AA,125000,175000
Canada,AA,105000,98000
Norway,AA,74850,68500
Ireland,AA,49000,48000
Sweden,BB,98000,99000
Germany,BB,115000,175000
```

```
Norway,BB,71850,68500
Ireland,BB,31000,48000
] (delimiter is ',');
```

Chart expression

Create a pivot table visualization in a Qlik Sense sheet with **Country** and **Product** as dimensions. Add **Sum (Sales)**, **Sum(Budget)**, and **Dimensionality()** as measures.

In the **Properties** panel, enter the following expression as the **Background color expression** for the **Sum (Sales)** measure:

```
If(Dimensionality()=1 and Sum(Sales)<Sum(Budget),RGB(255,156,156),
If(Dimensionality()=2 and Sum(Sales)<Sum(Budget),RGB(178,29,29)
))
```

Result:

Country / Product	Values		
	Sum(Sales)	Sum([Budget])	Dimensionality()
Canada	105000	98000	1
AA	105000	98000	2
Germany	240000	350000	1
Ireland	80000	96000	1
AA	49000	48000	2
BB	31000	48000	2
Norway	146700	137000	1
AA	74850	68500	2
BB	71850	68500	2
Sweden	198000	149000	1

Explanation

The expression `If(Dimensionality()=1 and Sum(Sales)<Sum(Budget),RGB(255,156,156), If (Dimensionality()=2 and Sum(Sales)<Sum(Budget),RGB(178,29,29)))` contains conditional statements that check the dimensionality value and the Sum(Sales) and Sum(Budget) for each product. If the conditions are met, a background color is applied to the Sum(Sales) value.

Exists

Exists() determines whether a specific field value has already been loaded into the field in the data load script. The function returns TRUE or FALSE, so can be used in the **where** clause of a **LOAD** statement or an **IF** statement.

*You can also use **Not Exists()** to determine if a field value has not been loaded, but caution is recommended if you use **Not Exists()** in a where clause. The **Exists()** function tests both previously loaded tables and previously loaded values in the current table. So, only the first occurrence will be loaded. When the second occurrence is encountered, the value is already loaded. See the examples for more information.*

Syntax:

```
Exists(field_name [, expr])
```

Return data type: Boolean

Arguments:

Arguments

Argument	Description
field_name	The name of the field where you want to search for a value. You can use an explicit field name without quotes. The field must already be loaded by the script. That means, you cannot refer to a field that is loaded in a clause further down in the script.
expr	The value that you want to check if it exists. You can use an explicit value or an expression that refers to one or several fields in the current load statement. *You cannot refer to fields that are not included in the current load statement.* This argument is optional. If you omit it, the function will check if the value of **field_name** in the current record already exists.

Examples and results:

Example 1

```
Exists (Employee)
```

Returns -1 (True) if the value of the field **Employee** in the current record already exists in any previously read record containing that field.

The statements `Exists (Employee, Employee)` and `Exists (Employee)` are equivalent.

Example 2

```
Exists(Employee, 'Bill')
```

Returns -1 (True) if the field value **'Bill'** is found in the current content of the field **Employee**.

Example 3

```
Employees:
LOAD * inline [
Employee|ID|Salary
Bill|001|20000
John|002|30000
Steve|003|35000
] (delimiter is '|');

Citizens:
Load * inline [
Employee|Address
Bill|New York
Mary|London
Steve|Chicago
Lucy|Madrid
Lucy|Paris
John|Miami
] (delimiter is '|') where Exists (Employee);

Drop Tables Employees;
```

This results in a table that you can use in a table visualization using the dimensions Employee and Address.

The where clause, `where Exists (Employee)`, means only the names from the table Citizens that are also in Employees are loaded into the new table. The Drop statement removes the table Employees to avoid confusion.

Results

Employee	Address
Bill	New York
John	Miami
Steve	Chicago

Example 4

```
Employees:
Load * inline [
Employee|ID|Salary
Bill|001|20000
John|002|30000
Steve|003|35000
] (delimiter is '|');

Citizens:
Load * inline [
Employee|Address
```

```
Bill|New York
Mary|London
Steve|Chicago
Lucy|Madrid
Lucy|Paris
John|Miami
] (delimiter is '|') where not Exists (Employee);

Drop Tables Employees;
```

The where clause includes not: `where not Exists (Employee)`.

This means that only the names from the table Citizens that are not in Employees are loaded into the new table.

Note that there are two values for Lucy in the Citizens table, but only one is included in the result table. When you load the first row with the value Lucy, it is included in the Employee field. Hence, when the second line is checked, the value already exists.

Results

Employee	Address
Mary	London
Lucy	Madrid

Example 5

This example shows how to load all values.

```
Employees:
Load Employee As Name;
LOAD * inline [
Employee|ID|Salary
Bill|001|20000
John|002|30000
Steve|003|35000
] (delimiter is '|');

Citizens:
Load * inline [
Employee|Address
Bill|New York
Mary|London
Steve|Chicago
Lucy|Madrid
Lucy|Paris
John|Miami
] (delimiter is '|') where not Exists (Name, Employee);

Drop Tables Employees;
```

To be able to get all values for Lucy, two things were changed:

- A preceding load to the Employees table was inserted where Employee was renamed to Name.
  `Load Employee As Name;`
- The Where condition in Citizens was changed to:
  `not Exists (Name, Employee).`

This creates fields for Name and Employee. When the second row with Lucy is checked, it still does not exist in Name.

Results

Employee	Address
Mary	London
Lucy	Madrid
Lucy	Paris

FieldIndex

FieldIndex() returns the position of the field value **value** in the field **field_name** (by load order).

Syntax:

```
FieldIndex(field_name , value)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
field_name	Name of the field for which the index is required. For example, the column in a table. Must be given as a string value. This means that the field name must be enclosed by single quotes.
value	The value of the field **field_name**.

Limitations:

If **value** cannot be found among the field values of the field **field_name**, 0 is returned.

Examples and results:

The following examples use the field: **First name** from the table**Names**.

Examples and results

Examples	Results
Add the example data to your app and run it.	The table **Names** is loaded, as in the sample data.

Examples	Results
Chart function: In a table containing the dimension First name, add as a measure:	
`FieldIndex ('First name','John')`	1, because 'John' appears first in the load order of the **First name** field. Note that in a filter pane **John** would appear as number 2 from the top as it's sorted alphabetically and not as in the load order.
`FieldIndex ('First name','Peter')`	4, because **FieldIndex()** returns only one value, that is the first occurrence in the load order.
Script function: Given the table **Names** is loaded, as in the example data:	

`John1:`
`Load FieldIndex('First name','John') as MyJohnPos`
`Resident Names;`
	`MyJohnPos=1`, because 'John' appears first in the load order of the **First name** field. Note that in a filter pane **John** would appear as number 2 from the top as it's sorted alphabetically and not as in the load order.

`Peter1:`
`Load FieldIndex('First name','Peter') as MyPeterPos`
`Resident Names;`
	`MyPeterPos=4`, because **FieldIndex()** returns only one value, that is the first occurrence in the load order.

Data used in example:

```
Names:
LOAD * inline [
First name|Last name|Initials|Has cellphone
John|Anderson|JA|Yes
Sue|Brown|SB|Yes
Mark|Carr|MC|No
Peter|Devonshire|PD|No
Jane|Elliot|JE|Yes
Peter|Franc|PF|Yes ] (delimiter is '|');

John1:
Load FieldIndex('First name','John') as MyJohnPos
Resident Names;

Peter1:
Load FieldIndex('First name','Peter') as MyPeterPos
Resident Names;
```

FieldValue

FieldValue() returns the value found in position **elem_no** of the field **field_name** (by load order).

Syntax:

```
FieldValue(field_name , elem_no)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
field_name	Name of the field for which the value is required. For example, the column in a table. Must be given as a string value. This means that the field name must be enclosed by single quotes.
elem_no	The position (element) number of the field, following the load order, that the value is returned for. This could correspond to the row in a table, but it depends on the order in which the elements (rows) are loaded.

Limitations:

If **elem_no** is larger than the number of field values, NULL is returned.

Example

Load script

Load the following data as an inline load in the data load editor to create the example below.

```
Names:

LOAD * inline [
First name|Last name|Initials|Has cellphone
John|Anderson|JA|Yes
Sue|Brown|SB|Yes
Mark|Carr|MC |No
Peter|Devonshire|PD|No
Jane|Elliot|JE|Yes
Peter|Franc|PF|Yes ] (delimiter is '|');

John1:

Load FieldValue('First name',1) as MyPos1
Resident Names;

Peter1:

Load FieldValue('First name',5) as MyPos2
Resident Names;
```

Create a visualization

Create a table visualization in a Qlik Sense sheet. Add fields **First name**, **MyPos1**, and **MyPos2** to the table.

Result

First name	MyPos1	MyPos2
Jane	John	Jane
John	John	Jane
Mark	John	Jane
Peter	John	Jane
Sue	John	Jane

Explanation

FieldValue('First name','1') results in John as the value for **MyPos1** for all first names because John appears first in the load order of the **First name** field. Note that in a filter pane John would appear as number 2 from the top, after Jane, as it's sorted alphabetically and not as in the load order.

FieldValue('First name','5') results in Jane as the value for **MyPos2** for all first names because Jane appears fifth in the load order of the **First name** field.

FieldValueCount

FieldValueCount() is an **integer** function that returns the number of distinct values in a field.

A partial reload can remove values from the data, which will not be reflected in the number returned. The returned number will correspond to all distinct values that were loaded in either the initial reload or any subsequent partial reload.

Syntax:

```
FieldValueCount(field_name)
```

Return data type: integer

Arguments:

Arguments

Argument	Description
field_name	Name of the field for which the value is required. For example, the column in a table. Must be given as a string value. This means that the field name must be enclosed by single quotes.

Examples and results:

The following examples use the field **First name** from the table **Names**.

Examples and results

Examples	Results
Add the example data to your app and run it.	The table **Names** is loaded, as in the sample data.
Chart function: In a table containing the dimension First name, add as a measure:	
`FieldValueCount('First name')`	5 as **Peter** appears twice.
`FieldValueCount('Initials')`	6 as **Initials** only has distinct values.
Script function: Given the table **Names** is loaded, as in the example data:	

`FieldCount1:`
`Load FieldValueCount('First name') as MyFieldCount1`
`Resident Names;`
	`MyFieldCount1`=5, because 'Peter' appears twice.

`FieldCount2:`
`Load FieldValueCount('Initials') as MyInitialsCount1`
`Resident Names;`
	`MyFieldCount1`=6, because 'Initials' only has distinct values.

Data used in examples:

```
Names:
LOAD * inline [
First name|Last name|Initials|Has cellphone
John|Anderson|JA|Yes
Sue|Brown|SB|Yes
Mark|Carr|MC|No
Peter|Devonshire|PD|No
Jane|Elliot|JE|Yes
Peter|Franc|PF|Yes ] (delimiter is '|');

FieldCount1:
Load FieldValueCount('First name') as MyFieldCount1
Resident Names;

FieldCount2:
Load FieldValueCount('Initials') as MyInitialsCount1
Resident Names;
```

LookUp

Lookup() looks into a table that is already loaded and returns the value of **field_name** corresponding to the first occurrence of the value **match_field_value** in the field **match_field_name**. The table can be the current table or another table previously loaded.

Syntax:

```
lookup(field_name, match_field_name, match_field_value [, table_name])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
field_name	Name of the field for which the return value is required. Input value must be given as a string (for example, quoted literals).
match_field_name	Name of the field to look up **match_field_value** in. Input value must be given as a string (for example, quoted literals).
match_field_value	Value to look up in **match_field_name** field.
table_name	Name of the table in which to look up the value. Input value must be given as a string (for example quoted literals). If **table_name** is omitted the current table is assumed.

Arguments without quotes refer to the current table. To refer to other tables, enclose an argument in single quotes.

Limitations:

The order in which the search is made is the load order, unless the table is the result of complex operations such as joins, in which case, the order is not well defined. Both **field_name** and **match_field_name** must be fields in the same table, specified by **table_name**.

If no match is found, NULL is returned.

Example

Load script

Load the following data as an inline load in the data load editor to create the example below.

```
ProductList:
Load * Inline [
ProductID|Product|Category|Price
1|AA|1|1
2|BB|1|3
3|CC|2|8
4|DD|3|2
] (delimiter is '|');

OrderData:
Load *, Lookup('Category', 'ProductID', ProductID, 'ProductList') as CategoryID
Inline [
```

```
InvoiceID|CustomerID|ProductID|Units
1|Astrida|1|8
1|Astrida|2|6
2|Betacab|3|10
3|Divadip|3|5
4|Divadip|4|10
] (delimiter is '|');

Drop Table ProductList;
```

Create a visualization

Create a table visualization in a Qlik Sense sheet. Add fields **ProductID**, **InvoiceID**, **CustomerID**, **Units**, and **CategoryID** to the table.

Result

Resulting table

ProductID	InvoiceID	CustomerID	Units	CategoryID
1	1	Astrida	8	1
2	1	Astrida	6	1
3	2	Betacab	10	2
3	3	Divadip	5	2
4	4	Divadip	10	3

Explanation

The sample data uses the **Lookup()** function in the following form:

```
Lookup('Category', 'ProductID', ProductID, 'ProductList')
```

The **ProductList** table is loaded first.

The **Lookup()** function is used to build the **OrderData** table. It specifies the third argument as **ProductID**. This is the field for which the value is to be looked up in the second argument **'ProductID'** in the **ProductList**, as denoted by the enclosing single quotes.

The function returns the value for '**Category**' (in the **ProductList** table), loaded as **CategoryID**.

The **drop** statement deletes the **ProductList** table from the data model because it is not required, which leaves the resulting **OrderData** table.

> *The Lookup() function is flexible and can access any previously loaded table. However, it is slow compared with the Applymap() function.*

See also:

ApplyMap (page 894)

NoOfRows - chart function

NoOfRows() returns the number of rows in the current column segment in a table. For bitmap charts, **NoOfRows()** returns the number of rows in the chart's straight table equivalent.

If the table or table equivalent has multiple vertical dimensions, the current column segment will include only rows with the same values as the current row in all dimension columns, except for the column showing the last dimension in the inter-field sort order.

Syntax:

```
NoOfRows([TOTAL])
```

Return data type: integer

Arguments:

Arguments

Argument	Description
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

Example: Chart expression using NoOfRows

Example - chart expression

Load script

Load the following data as an inline load in the data load editor to create the chart expression examples below.

```
        Temp:
LOAD * inline [
Region|SubRegion|RowNo()|NoOfRows()
Africa|Eastern
Africa|Western
Americas|Central
Americas|Northern
Asia|Eastern
Europe|Eastern
Europe|Northern
Europe|Western
Oceania|Australia
] (delimiter is '|');
```

Chart expression

Create a table visualization in a Qlik Sense sheet with **Region** and **SubRegion** as dimensions. Add `RowNo( )`, `NoOfRows()`, and `NoOfRows(Total)` as measures.

Result

Region	SubRegion	RowNo()	NoOfRows()	NoOfRows (Total)
Africa	Eastern	1	2	9
Africa	Western	2	2	9
Americas	Central	1	2	9
Americas	Northern	2	2	9
Asia	Eastern	1	1	9
Europe	Eastern	1	3	9
Europe	Northern	2	3	9
Eurrope	Western	3	3	9
Oceania	Australia	1	1	9

Explanation

In this example, the sort order is by the first dimension, Region. As a result, each column segment is made up of a group of regions that has the same value, for example, Africa.

The **RowNo()** column shows the row numbers for each column segment, for example, there are two rows for the Africa region. The row numbering then begins at 1 again for the next column segment, which is Americas.

The **NoOfRows()** column counts the number of rows in each column segment, for example, Europe has three rows in the column segment.

The **NoOfRows(Total)** column disregards the dimensions because of the `TOTAL` argument for `NoOfRows()` and counts the rows in the table.

If the table was sorted on the second dimension, SubRegion, the column segments would be based on that dimension so the row numbering would change for each SubRegion.

See also:

RowNo - chart function (page 500)

Peek

Peek() returns the value of a field in a table for a row that has already been loaded. The row number can be specified, as can the table. If no row number is specified, the last previously loaded record will be used.

The peek() function is most often used to find the relevant boundaries in a previously loaded table, that is, the first value or last value of a specific field. In most cases, this value is stored in a variable for later use, for example, as a condition in a do-while loop.

Syntax:

```
Peek(
field_name
[, row_no[, table_name ] ])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
field_name	Name of the field for which the return value is required.Input value must be given as a string (for example, quoted literals).
row_no	The row in the table that specifies the field required. Can be an expression, but must resolve to an integer. 0 denotes the first record, 1 the second, and so on. Negative numbers indicate order from the end of the table. -1 denotes the last record read. If no **row_no** is stated, -1 is assumed.
table_name	A table label without the ending colon. If no **table_name** is stated, the current table is assumed. If used outside the **LOAD** statement or referring to another table, the **table_name** must be included.

Limitations:

The function can only return values from already loaded records. This means that in the first record of a table, a call using -1 as row_no will return NULL.

Examples and results:

Example 1

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
EmployeeDates:
Load * Inline [
EmployeeCode|StartDate|EndDate
101|02/11/2010|23/06/2012
102|01/11/2011|30/11/2013
103|02/01/2012|
104|02/01/2012|31/03/2012
105|01/04/2012|31/01/2013
106|02/11/2013|
] (delimiter is '|');
```

```
First_last_Employee:
Load
EmployeeCode,
Peek('EmployeeCode',0,'EmployeeDates') As FirstCode,
Peek('EmployeeCode',-1,'EmployeeDates') As LastCode
Resident EmployeeDates;
```

Resulting table

Employee code	StartDate	EndDate	FirstCode	LastCode
101	02/11/2010	23/06/2012	101	106
102	01/11/2011	30/11/2013	101	106
103	02/01/2012		101	106
104	02/01/2012	31/03/2012	101	106
105	01/04/2012	31/01/2013	101	106
106	02/11/2013		101	106

FirstCode = 101 because `Peek('EmployeeCode',0, 'EmployeeDates')` returns the first value of EmployeeCode in the table EmployeeDates.

LastCode = 106 because `Peek('EmployeeCode',-1, 'EmployeeDates')` returns the last value of EmployeeCode in the table EmployeeDates.

Substituting the value of the argument **row_no** returns the values of other rows in the table, as follows:

`Peek('EmployeeCode',2, 'EmployeeDates')` returns the third value, 103, in the table as the FirstCode.

However, note that without specifying the table as the third argument **table_name** in these examples, the function references the current (in this case, internal) table.

Example 2

If you want to access data further down in a table, you need to do it in two steps: first, load the entire table into a temporary table, and then re-sort it when using **Peek()**.

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
T1:
LOAD * inline [
ID|Value
1|3
1|4
1|6
3|7
3|8
2|1
2|11
5|2
5|78
5|13
```

```
] (delimiter is '|');

T2:

LOAD *,
IF(ID=Peek('ID'), Peek('List')&','&Value,Value) AS List
RESIDENT T1
ORDER BY ID ASC;
DROP TABLE T1;
```

Create a table in a sheet in your app with **ID**, **List**, and **Value** as the dimensions.

Resulting table

ID	List	Value
1	3,4	4
1	3,4,6	6
1	3	3
2	1,11	11
2	1	1
3	7,8	8
3	7	7
5	2,78	78
5	2,78,13	13
5	2	2

The **IF()** statement is built from the temporary table T1.
`Peek('ID')` references the field ID in the previous row in the current table T2.
`Peek('List')` references the field List in the previous row in the table T2, currently being built as the expression is evaluated.

The statement is evaluated as follows:
If the current value of ID is the same as the previous value of ID, then write the value of Peek('List') concatenated with the current value of Value. Otherwise, write the current value of Value only.

If Peek('List') already contains a concatenated result, the new result of Peek('List') will be concatenated to it.

> *Note the **Order by** clause. This specifies how the table is ordered (by ID in ascending order). Without this, the Peek() function will use whatever arbitrary ordering the internal table has, which can lead to unpredictable results.*

Example 3

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
Amounts:
Load
Date#(Month,'YYYY-MM') as Month,
Amount,
Peek(Amount) as AmountMonthBefore
Inline
[Month,Amount
2022-01,2
2022-02,3
2022-03,7
2022-04,9
2022-05,4
2022-06,1];
```

Resulting table

Amount	AmountMonthBefore	Month
1	4	2022-06
2	-	2022-01
3	2	2022-02
4	9	2022-05
7	3	2022-03
9	7	2022-04

The field AmountMonthBefore will hold the amount from the previous month.

Here, the row_no and table_name parameters are omitted, so the default values are used. In this example, the following three function calls are equivalent:

- Peek(Amount)
- Peek(Amount,-1)
- Peek(Amount,-1,'Amounts')

Using -1 as row_no means that the value from previous row will be used. By substituting this value, values of other rows in the table can be fetched:

Peek(Amount,2) returns the third value in the table: 7.

Example 4

Data needs to be correctly sorted in order to get the correct results but, unfortunately, this is not always the case. Furthermore, the Peek() function cannot be used to reference data that has not yet been loaded. By using temporary tables and running multiple passes through the data, such problems can be avoided.

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
tmp1Amounts:
Load * Inline
[Month,Product,Amount
```

```
2022-01,B,3
2022-01,A,8
2022-02,B,4
2022-02,A,6
2022-03,B,1
2022-03,A,6
2022-04,A,5
2022-04,B,5
2022-05,B,6
2022-05,A,7
2022-06,A,4
2022-06,B,8];

tmp2Amounts:
Load *,
If(Product=Peek(Product),Peek(Amount)) as AmountMonthBefore
Resident tmp1Amounts
Order By Product, Month Asc;
Drop Table tmp1Amounts;

Amounts:
Load *,
If(Product=Peek(Product),Peek(Amount)) as AmountMonthAfter
Resident tmp2Amounts
Order By Product, Month Desc;
Drop Table tmp2Amounts;
```

Explanation

The initial table is sorted according to month, which means that the peek() function would in many cases return the amount for the wrong product. Hence, this table needs to be re-sorted. This is done by running a second pass through the data creating a new table tmp2Amounts. Note the Order By clause. It orders the records first by product, then by month in ascending order.

The If() function is needed since the AmountMonthBefore only should be calculated if the previous row contains the data for the same product but for the previous month. By comparing the product on the current row with the product on the previous row, this condition can be validated.

When the second table is created, the first table tmp1Amounts is dropped using a Drop Table statement.

Finally, a third pass is made through the data, but now with the months sorted in reverse order. This way, AmountMonthAfter can also be calculated.

Order by clauses specify how the table is ordered; without these, the Peek() function will use whatever arbitrary ordering the internal table has, which can lead to unpredictable results.

Result

Resulting table

Month	Product	Amount	AmountMonthBefore	AmountMonthAfter
2022-01	A	8	-	6
2022-02	B	3	-	4
2022-03	A	6	8	6
2022-04	B	4	3	1
2022-05	A	6	6	5
2022-06	B	1	4	5
2022-01	A	5	6	7
2022-02	B	5	1	6
2022-03	A	7	5	4
2022-04	B	6	5	8
2022-05	A	4	7	-
2022-06	B	8	6	-

Example 5

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
T1:
Load * inline [
Quarter, Value
2003q1, 10000
2003q1, 25000
2003q1, 30000
2003q2, 1250
2003q2, 55000
2003q2, 76200
2003q3, 9240
2003q3, 33150
2003q3, 89450
2003q4, 1000
2003q4, 3000
2003q4, 5000
2004q1, 1000
2004q1, 1250
2004q1, 3000
2004q2, 5000
2004q2, 9240
2004q2, 10000
2004q3, 25000
2004q3, 30000
```

```
2004q3, 33150
2004q4, 55000
2004q4, 76200
2004q4, 89450 ];

T2:
Load *, rangesum(SumVal,peek('AccSumVal')) as AccSumVal;
Load Quarter, sum(Value) as SumVal resident T1 group by Quarter;
```

Result

Resulting table

Quarter	SumVal	AccSumVal
2003q1	65000	65000
2003q2	132450	197450
2003q3	131840	329290
2003q4	9000	338290
2004q1	5250	343540
2004q2	24240	367780
2004q3	88150	455930
2004q4	220650	676580

Explanation

The load statement **Load *, rangesum(SumVal,peek('AccSumVal')) as AccSumVal** includes a recursive call where the previous values are added to the current value. This operation is used to calculate an accumulation of values in the script.

See also:

Previous

Previous() finds the value of the **expr** expression using data from the previous input record that has not been discarded because of a **where** clause. In the first record of an internal table, the function will return NULL.

Syntax:

```
Previous(expr)
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured. The expression can contain nested **previous()** functions in order to access records further back. Data are fetched directly from the input source, making it possible to refer also to fields that have not been loaded into Qlik Sense, that is,even if they have not been stored in its associative database.

Limitations:

In the first record of an internal table, the function returns NULL.

Example:

Input the following into your load script

```
Sales2013:
Load *, (Sales - Previous(Sales) )as Increase Inline [
Month|Sales
1|12
2|13
3|15
4|17
5|21
6|21
7|22
8|23
9|32
10|35
11|40
12|41
] (delimiter is '|');
```

By using the **Previous()** function in the **Load** statement, we can compare the current value of Sales with the preceding value, and use it in a third field, Increase.

Resulting table

Month	Sales	Increase
1	12	-
2	13	1
3	15	2
4	17	2

Month	Sales	Increase
5	21	4
6	21	0
7	22	1
8	23	1
9	32	9
10	35	3
11	40	5
12	41	1

Top - chart function

Top() evaluates an expression at the first (top) row of a column segment in a table. The row for which it is calculated depends on the value of **offset**, if present, the default being the top row. For charts other than tables, the**Top()** evaluation is made on the first row of the current column in the chart's straight table equivalent.

Syntax:

```
Top([TOTAL] expr [ , offset [,count ]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
offset	Specifying an **offset**of n, greater than 1, moves the evaluation of the expression down n rows below the top row. Specifying a negative offset number makes the **Top** function work like the **Bottom** function with the corresponding positive offset number.
count	By specifying a third parameter **count** greater than 1, the function will return a range of **count** values, one for each of the last **count** rows of the current column segment. In this form, the function can be used as an argument to any of the special range functions. *Range functions (page 901)*
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

A column segment is defined as a consecutive subset of cells having the same values for the dimensions in the current sort order. Inter-record chart functions are computed in the column segment excluding the right-most dimension in the equivalent straight table chart. If there is only one dimension in the chart, or if the TOTAL qualifier is specified, the expression evaluates across full table.

If the table or table equivalent has multiple vertical dimensions, the current column segment will include only rows with the same values as the current row in all dimension columns, except for the column showing the last dimension in the inter-field sort order.

Limitations:

Recursive calls will return NULL.

Examples and results:

Example: 1

In the screenshot of the table shown in this example, the table visualization is created from the dimension **Customer** and the measures: `Sum(Sales)` and `Top(Sum(Sales))`.

The column **Top(Sum(Sales))** returns 587 for all rows because this is the value of the top row: **Astrida**.

The table also shows more complex measures: one created from `Sum(Sales)+Top(Sum(Sales))` and one labeled **Top offset 3**, which is created using the expression `Sum(Sales)+Top(Sum(Sales), 3)` and has the argument **offset** set to 3. It adds the **Sum(Sales)** value for the current row to the value from the third row from the top row, that is, the current row plus the value for **Canutility**.

Example 1

Top and Bottom

Customer	Sum(Sales)	Top(Sum(Sales))	Sum(Sales)+Top(Sum(Sales))	Top offset 3
Totals	2566	587	3153	3249
Astrida	587	587	1174	1270
Betacab	539	587	1126	1222
Canutility	683	587	1270	1366
Divadip	757	587	1344	1440

Example: 2

In the screenshots of tables shown in this example, more dimensions have been added to the visualizations: **Month** and **Product**. For charts with more than one dimension, the results of expressions containing the **Above**, **Below**, **Top**, and **Bottom** functions depend on the order in which the column dimensions are sorted

by Qlik Sense. Qlik Sense evaluates the functions based on the column segments that result from the dimension that is sorted last. The column sort order is controlled in the properties panel under **Sorting** and is not necessarily the order in which the columns appear in a table.

First table for Example 2. The value of Top for the First value measure based on Month (Jan).

Customer	Product	Month	Sum(Sales)	First value
			2566	-
Astrida	AA	Jan	46	46
Astrida	AA	Feb	60	46
Astrida	AA	Mar	70	46
Astrida	AA	Apr	13	46
Astrida	AA	May	78	46
Astrida	AA	Jun	20	46
Astrida	AA	Jul	45	46
Astrida	AA	Aug	65	46
Astrida	AA	Sep	78	46
Astrida	AA	Oct	12	46
Astrida	AA	Nov	78	46
Astrida	AA	Dec	22	46

Second table for Example 2. The value of Top for the First value measure based on Product (AA for Astrida).

Customer	Product	Month	Sum(Sales)	First value
			2566	-
Astrida	AA	Jan	46	46
Astrida	BB	Jan	46	46
Astrida	AA	Feb	60	60
Astrida	BB	Feb	60	60
Astrida	AA	Mar	70	70
Astrida	BB	Mar	70	70
Astrida	AA	Apr	13	13
Astrida	BB	Apr	13	13

Please refer to Example: 2 in the **Above** function for further details.

Example 3

Example: 3	Result
The **Top** function can be used as input to the range functions. For example: `RangeAvg (Top(Sum (Sales),1,3))`.	In the arguments for the **Top()** function, offset is set to 1 and count is set to 3. The function finds the results of the expression**Sum(Sales)** on the three rows starting with the row below the bottom row in the column segment (because the offset=1), and the two rows below that (where there is a row). These three values are used as input to the RangeAvg() function, which finds the average of the values in the supplied range of numbers. A table with **Customer** as dimension gives the following results for the RangeAvg() expression.
	Astrida 603 Betacab 603 Canutility 603 Divadip: 603

```
Monthnames:
LOAD *, Dual(MonthText,MonthNumber) as Month INLINE [
MonthText, MonthNumber
Jan, 1
Feb, 2
Mar, 3
Apr, 4
May, 5
Jun, 6
Jul, 7
Aug, 8
Sep, 9
Oct, 10
Nov, 11
Dec, 12
];

Sales2013:
Crosstable (MonthText, Sales) LOAD * inline [
Customer|Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec
Astrida|46|60|70|13|78|20|45|65|78|12|78|22
Betacab|65|56|22|79|12|56|45|24|32|78|55|15
Canutility|77|68|34|91|24|68|57|36|44|90|67|27
Divadip|57|36|44|90|67|27|57|68|47|90|80|94
] (delimiter is '|');
```

See also:

- *Bottom - chart function (page 856)*
- *Above - chart function (page 848)*
- *Sum - chart function (page 286)*
- *RangeAvg (page 904)*
- *Range functions (page 901)*

SecondaryDimensionality - chart function

SecondaryDimensionality() returns the number of dimension pivot table rows that have non-aggregation content, that is, do not contain partial sums or collapsed aggregates. This function is the equivalent of the **dimensionality()** function for horizontal pivot table dimensions.

Syntax:

```
SecondaryDimensionality( )
```

Return data type: integer

Limitations:

Unless used in pivot tables, the **SecondaryDimensionality** function always returns 0.

After - chart function

After() returns the value of an expression evaluated with a pivot table's dimension values as they appear in the column after the current column within a row segment in the pivot table.

Syntax:

```
after([TOTAL] expr [, offset [, count ]])
```

This function returns NULL in all chart types except pivot tables.

Arguments:

Argument	Description
expr	The expression or field containing the data to be measured.
offset	Specifying an **offset** n, greater than 1 moves the evaluation of the expression n rows further to the right from the current row. Specifying an offset of 0 will evaluate the expression on the current row. Specifying a negative offset number makes the **After** function work like the **Before** function with the corresponding positive offset number.

Argument	Description
count	By specifying a third parameter **count** greater than 1, the function will return a range of values, one for each of the table rows up to the value of **count**, counting to the right from the original cell.
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

On the last column of a row segment a NULL value will be returned, as there is no column after this one.

If the pivot table has multiple horizontal dimensions, the current row segment will include only columns with the same values as the current column in all dimension rows except for the row showing the last horizontal dimension of the inter-field sort order. The inter-field sort order for horizontal dimensions in pivot tables is defined simply by the order of the dimensions from top to bottom.

Example:

```
after( sum( Sales ))
after( sum( Sales ), 2 )
after( total sum( Sales ))
```

`rangeavg (after(sum(x),1,3))` returns an average of the three results of the **sum(x)** function evaluated in the three columns immediately to the right of the current column.

Before - chart function

Before() returns the value of an expression evaluated with a pivot table's dimension values as they appear in the column before the current column within a row segment in the pivot table.

Syntax:

```
before([TOTAL] expr [, offset [, count]])
```

This function returns NULL in all chart types except pivot tables.

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
offset	Specifying an **offset** n, greater than 1 moves the evaluation of the expression n rows further to the left from the current row. Specifying an offset of 0 will evaluate the expression on the current row. Specifying a negative offset number makes the **Before** function work like the **After** function with the corresponding positive offset number.

Argument	Description
count	By specifying a third parameter **count** greater than 1, the function will return a range of values, one for each of the table rows up to the value of **count**, counting to the left from the original cell.
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

On the first column of a row segment a NULL value will be returned, as there is no column before this one.

If the pivot table has multiple horizontal dimensions, the current row segment will include only columns with the same values as the current column in all dimension rows except for the row showing the last horizontal dimension of the inter-field sort order.The inter-field sort order for horizontal dimensions in pivot tables is defined simply by the order of the dimensions from top to bottom.

Examples:

```
before( sum( Sales ))
before( sum( Sales ), 2 )
before( total sum( Sales ))
```

`rangeavg (before(sum(x),1,3))` returns an average of the three results of the **sum(x)** function evaluated in the three columns immediately to the left of the current column.

First - chart function

First() returns the value of an expression evaluated with a pivot table's dimension values as they appear in the first column of the current row segment in the pivot table. This function returns NULL in all chart types except pivot tables.

Syntax:

```
first([TOTAL] expr [, offset [, count]])
```

Arguments:

Arguments

Argument	Description
expression	The expression or field containing the data to be measured.
offset	Specifying an **offset** n, greater than 1 moves the evaluation of the expression n rows further to the right from the current row. Specifying an offset of 0 will evaluate the expression on the current row. Specifying a negative offset number makes the **First** function work like the **Last** function with the corresponding positive offset number.

Argument	Description
count	By specifying a third parameter **count** greater than 1, the function will return a range of values, one for each of the table rows up to the value of **count**, counting to the right from the original cell.
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

If the pivot table has multiple horizontal dimensions, the current row segment will include only columns with the same values as the current column in all dimension rows except for the row showing the last horizontal dimension of the inter-field sort order.The inter-field sort order for horizontal dimensions in pivot tables is defined simply by the order of the dimensions from top to bottom.

Examples:

```
first( sum( Sales ))
first( sum( Sales ), 2 )
first( total sum( Sales )
```

`rangeavg (first(sum(x),1,5))` returns an average of the results of the **sum(x)** function evaluated on the five leftmost columns of the current row segment.

Last - chart function

Last() returns the value of an expression evaluated with a pivot table's dimension values as they appear in the last column of the current row segment in the pivot table. This function returns NULL in all chart types except pivot tables.

Syntax:

```
last([TOTAL] expr [, offset [, count]])
```

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
offset	Specifying an **offset** n, greater than 1 moves the evaluation of the expression n rows further to the left from the current row. Specifying an offset of 0 will evaluate the expression on the current row. Specifying a negative offset number makes the **First** function work like the **Last** function with the corresponding positive offset number.
count	By specifying a third parameter **count** greater than 1, the function will return a range of values, one for each of the table rows up to the value of **count**, counting to the left from the original cell.
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

If the pivot table has multiple horizontal dimensions, the current row segment will include only columns with the same values as the current column in all dimension rows except for the row showing the last horizontal dimension of the inter-field sort order.The inter-field sort order for horizontal dimensions in pivot tables is defined simply by the order of the dimensions from top to bottom.

Example:

```
last( sum( Sales ))
last( sum( Sales ), 2 )
last( total sum( Sales )
```

`rangeavg (last(sum(x),1,5))` returns an average of the results of the **sum(x)** function evaluated on the five rightmost columns of the current row segment.

ColumnNo - chart function

ColumnNo() returns the number of the current column within the current row segment in a pivot table. The first column is number 1.

Syntax:

```
ColumnNo([total])
```

Arguments:

Arguments

Argument	Description
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

If the pivot table has multiple horizontal dimensions, the current row segment will include only columns with the same values as the current column in all dimension rows except for the row showing the last horizontal dimension of the inter-field sort order.The inter-field sort order for horizontal dimensions in pivot tables is defined simply by the order of the dimensions from top to bottom.

Example:

```
if( ColumnNo( )=1, 0, sum( Sales ) / before( sum( Sales )))
```

NoOfColumns - chart function

NoOfColumns() returns the number of columns in the current row segment in a pivot table.

Syntax:

```
NoOfColumns([total])
```

Arguments:

Arguments

Argument	Description
TOTAL	If the table is one-dimensional or if the qualifier **TOTAL** is used as argument, the current column segment is always equal to the entire column.

If the pivot table has multiple horizontal dimensions, the current row segment will include only columns with the same values as the current column in all dimension rows except for the row showing the last dimension in the inter-field sort order.The inter-field sort order for horizontal dimensions in pivot tables is defined simply by the order of the dimensions from top to bottom.

Example:

```
if( ColumnNo( )=NoOfColumns( ), 0, after( sum( Sales )))
```

5.17 Logical functions

This section describes functions handling logical operations. All functions can be used in both the data load script and in chart expressions.

IsNum

Returns -1 (True) if the expression can be interpreted as a number, otherwise 0 (False).

```
IsNum( expr )
```

IsText

Returns -1 (True) if the expression has a text representation, otherwise 0 (False).

```
IsText( expr )
```

*Both **IsNum** and **IsText** return 0 if the expression is NULL.*

Example:

The following example loads an inline table with mixed text and numerical values, and adds two fields to check if the value is a numerical value, respectively a text value.

```
Load *, IsNum(Value), IsText(Value)
Inline [
Value
23
Green
Blue
12
33Red];
```

The resulting table looks like this:

Resulting table

Value	IsNum(Value)	IsText(Value)
23	-1	0
Green	0	-1
Blue	0	-1
12	-1	0
33Red	0	-1

5.18 Mapping functions

This section describes functions for handling mapping tables. A mapping table can be used to replace field values or field names during script execution.

Mapping functions can only be used in the data load script.

Mapping functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

ApplyMap

The **ApplyMap** script function is used for mapping the output of an expression to a previously loaded mapping table.

```
ApplyMap ('mapname', expr [ , defaultexpr ] )
```

MapSubstring

The **MapSubstring** script function is used to map parts of any expression to a previously loaded mapping table. The mapping is case sensitive and non-iterative, and substrings are mapped from left to right.

```
MapSubstring ('mapname', expr)
```

ApplyMap

The **ApplyMap** script function is used for mapping the output of an expression to a previously loaded mapping table.

Syntax:

```
ApplyMap('map_name', expression [ , default_mapping ] )
```

Return data type: dual

Arguments:

Arguments

Argument	Description
map_name	The name of a mapping table that has previously been created through the **mapping load** or the **mapping select** statement. Its name must be enclosed by single, straight quotation marks. *If you use this function in a macro expanded variable and refer to a mapping table that does not exist, the function call fails and a field is not created.*
expression	The expression, the result of which should be mapped.
default_ mapping	If stated, this value will be used as a default value if the mapping table does not contain a matching value for expression. If not stated, the value of expression will be returned as is.

The output field of ApplyMap should not have the same name as one of its input fields. This may cause unexpected results. Example not to use: `ApplyMap('Map', A) as A`.

Example:

In this example we load a list of salespersons with a country code representing their country of residence. We use a table mapping a country code to a country to replace the country code with the country name. Only three countries are defined in the mapping table, other country codes are mapped to 'Rest of the world'.

```
// Load mapping table of country codes:
map1:
mapping LOAD *
Inline [
CCode, Country
Sw, Sweden
Dk, Denmark
No, Norway
] ;

// Load list of salesmen, mapping country code to country
// If the country code is not in the mapping table, put Rest of the world
Salespersons:
LOAD *,
ApplyMap('map1', CCode,'Rest of the world') As Country
Inline [
CCode, Salesperson
Sw, John
Sw, Mary
Sw, Per
```

```
Dk, Preben
Dk, Olle
No, Ole
Sf, Risttu
] ;

// We don't need the CCode anymore
Drop Field 'CCode';
```

The resulting table (Salespersons) looks like this:

Resulting table

Salesperson	Country
John	Sweden
Mary	Sweden
Per	Sweden
Preben	Denmark
Olle	Denmark
Ole	Norway
Risttu	Rest of the world

MapSubstring

The **MapSubstring** script function is used to map parts of any expression to a previously loaded mapping table. The mapping is case sensitive and non-iterative, and substrings are mapped from left to right.

Syntax:

```
MapSubstring('map_name', expression)
```

Return data type: string

Arguments:

Arguments

Argument	Description
map_name	The name of a mapping table previously read by a **mapping load** or a **mapping select** statement. The name must be enclosed by single straight quotation marks. *If you use this function in a macro expanded variable and refer to a mapping table that does not exist, the function call fails and a field is not created.*
expression	The expression whose result is to be mapped by substrings.

Example:

In this example we load a list of product models. Each model has a set of attributes that are described by a composite code. Using the mapping table with MapSubstring, we can expand the attribute codes to a description.

```
map2:
mapping LOAD *
Inline [
AttCode, Attribute
R, Red
Y, Yellow
B, Blue
C, Cotton
P, Polyester
S, Small
M, Medium
L, Large
] ;

Productmodels:
LOAD *,
MapSubString('map2', AttCode) as Description
Inline [
Model, AttCode
Twixie, R C S
Boomer, B P L
Raven, Y P M
Seedling, R C L
SeedlingPlus, R C L with hood
Younger, B C with patch
MultiStripe, R Y B C S/M/L
] ;
// We don't need the AttCode anymore
Drop Field 'AttCode';
```

The resulting table looks like this:

Resulting table

Model	Description
Twixie	Red Cotton Small
Boomer	Blue Polyester Large
Raven	Yellow Polyester Medium
Seedling	Red Cotton Large
SeedlingPlus	Red Cotton Large with hood
Younger	Blue Cotton with patch
MultiStripe	Red Yellow Blue Cotton Small/Medium/Large

5.19 Mathematical functions

This section describes functions for mathematical constants and Boolean values. These functions do not have any parameters, but the parentheses are still required.

All functions can be used in both the data load script and in chart expressions.

e

The function returns the base of the natural logarithms, **e** (2.71828...).

```
e( )
```

false

The function returns a dual value with text value 'False' and numeric value 0, which can be used as logical false in expressions.

```
false( )
```

pi

The function returns the value of π (3.14159...).

```
pi( )
```

rand

The function returns a random number between 0 and 1. This can be used to create sample data.

```
rand( )
```

Example:

This example script creates a table of 1000 records with randomly selected upper case characters, that is, characters in the range 65 to 91 (65+26).

```
Load
     Chr( Floor(rand() * 26) + 65) as UCaseChar,
     RecNo() as ID
     Autogenerate 1000;
```

true

The function returns a dual value with text value 'True' and numeric value -1, which can be used as logical true in expressions.

```
true( )
```

5.20 NULL functions

This section describes functions for returning or detecting NULL values.

All functions can be used in both the data load script and in chart expressions.

NULL functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

EmptyIsNull

The **EmptyIsNull** function converts empty strings to NULL. Hence, it returns NULL if the parameter is an empty string, otherwise it returns the parameter.

```
EmptyIsNull (expr )
```

IsNull

The **IsNull** function tests if the value of an expression is NULL and if so, returns -1 (True), otherwise 0 (False).

```
IsNull (expr )
```

Null

The **Null** function returns a NULL value.

```
NULL( )
```

EmptyIsNull

The **EmptyIsNull** function converts empty strings to NULL. Hence, it returns NULL if the parameter is an empty string, otherwise it returns the parameter.

Syntax:

```
EmptyIsNull(exp )
```

Examples and results:

Scripting examples

Example	Result
`EmptyIsNull(AdditionalComments)`	This expression will return as null any empty string values of the *AdditionalComments* field, instead of empty strings. Non-empty strings and numbers are returned.
`EmptyIsNull(PurgeChar (PhoneNumber, ' -()'))`	This expression will strip any dashes, spaces and parentheses from the *PhoneNumber* field. If there are no characters left, the EmptyIsNull function returns the empty string as null; an empty phone number is the same as no phone number.

IsNull

The **IsNull** function tests if the value of an expression is NULL and if so, returns -1 (True), otherwise 0 (False).

Syntax:

```
IsNull(expr )
```

*A string with length zero is not considered as a NULL and will cause **IsNull** to return False.*

Example: Data load script

In this example, an inline table with four rows is loaded, where the first three lines contain either nothing, - or 'NULL' in the Value column. We convert these values to true NULL value representations with the middle preceding **LOAD** using the **Null** function.

The first preceding **LOAD** adds a field checking if the value is NULL,using the **IsNull** function.

```
NullsDetectedAndConverted:

LOAD *,
If(IsNull(ValueNullConv), 'T', 'F') as IsItNull;

LOAD *,
If(len(trim(Value))= 0 or Value='NULL' or Value='-', Null(), Value ) as ValueNullConv;

LOAD * Inline
[ID, Value
0,
1,NULL
2,-
3,Value];
```

This is the resulting table. In the ValueNullConv column, the NULL values are represented by -.

Resulting table

ID	Value	ValueNullConv	IsItNull
0		-	T
1	NULL	-	T
2	-	-	T
3	Value	Value	F

NULL

The **Null** function returns a NULL value.

Syntax:

```
Null( )
```

Example: Data load script

In this example, an inline table with four rows is loaded, where the first three lines contain either nothing, - or 'NULL' in the Value column. We want to convert these values to true NULL value representations.

The middle preceding **LOAD** performs the conversion using the **Null** function.

The first preceding **LOAD** adds a field checking if the value is NULL, just for illustration purposes in this example.

```
NullsDetectedAndConverted:

LOAD *,
If(IsNull(ValueNullConv), 'T', 'F') as IsItNull;

LOAD *,
If(len(trim(Value))= 0 or Value='NULL' or Value='-', Null(), Value ) as ValueNullConv;

LOAD * Inline
[ID, Value
0,
1,NULL
2,-
3,Value];
```

This is the resulting table. In the ValueNullConv column, the NULL values are represented by -.

Resulting table

ID	Value	ValueNullConv	IsItNull
0		-	T
1	NULL	-	T
2	-	-	T
3	Value	Value	F

5.21 Range functions

The range functions are functions that take an array of values and produce a single value as a result. All range functions can be used in both the data load script and in chart expressions.

For example, in a visualization, a range function can calculate a single value from an inter-record array. In the data load script, a range function can calculate a single value from an array of values in an internal table.

Range functions replace the following general numeric functions: ***numsum****,* ***numavg****,* ***numcount****,* ***nummin*** *and* ***nummax****, which should now be regarded as obsolete.*

Basic range functions

RangeMax

RangeMax() returns the highest numeric values found within the expression or field.

```
RangeMax(first_expr[, Expression])
```

RangeMaxString

RangeMaxString() returns the last value in the text sort order that it finds in the expression or field.

```
RangeMaxString(first_expr[, Expression])
```

RangeMin

RangeMin() returns the lowest numeric values found within the expression or field.

```
RangeMin(first_expr[, Expression])
```

RangeMinString

RangeMinString() returns the first value in the text sort order that it finds in the expression or field.

```
RangeMinString(first_expr[, Expression])
```

RangeMode

RangeMode() finds the most commonly occurring value (mode value) in the expression or field.

```
RangeMode(first_expr[, Expression])
```

RangeOnly

RangeOnly() is a dual function that returns a value if the expression evaluates to one unique value. If this is not the case then **NULL** is returned.

```
RangeOnly(first_expr[, Expression])
```

RangeSum

RangeSum() returns the sum of a range of values. All non-numeric values are treated as 0.

```
RangeSum(first_expr[, Expression])
```

Counter range functions

RangeCount

RangeCount() returns the number of values, both text and numeric, in the expression or field.

```
RangeCount(first_expr[, Expression])
```

RangeMissingCount

RangeMissingCount() returns the number of non-numeric values (including NULL) in the expression or field.

```
RangeMissingCount(first_expr[, Expression])
```

RangeNullCount

RangeNullCount() finds the number of NULL values in the expression or field.

```
RangeNullCount(first_expr[, Expression])
```

RangeNumericCount

RangeNumericCount() finds the number of numeric values in an expression or field.

```
RangeNumericCount(first_expr[, Expression])
```

RangeTextCount

RangeTextCount() returns the number of text values in an expression or field.

```
RangeTextCount(first_expr[, Expression])
```

Statistical range functions

RangeAvg

RangeAvg() returns the average of a range. Input to the function can be either a range of values or an expression.

```
RangeAvg(first_expr[, Expression])
```

RangeCorrel

RangeCorrel() returns the correlation coefficient for two sets of data. The correlation coefficient is a measure of the relationship between the data sets.

```
RangeCorrel(x_values , y_values[, Expression])
```

RangeFractile

RangeFractile() returns the value that corresponds to the n-th **fractile** (quantile) of a range of numbers.

```
RangeFractile(fractile, first_expr[ ,Expression])
```

RangeKurtosis

RangeKurtosis() returns the value that corresponds to the kurtosis of a range of numbers.

```
RangeKurtosis(first_expr[, Expression])
```

RangeSkew

RangeSkew() returns the value corresponding to the skewness of a range of numbers.

```
RangeSkew(first_expr[, Expression])
```

RangeStdev

RangeStdev() finds the standard deviation of a range of numbers.

```
RangeStdev(expr1[, Expression])
```

Financial range functions

RangeIRR

RangeIRR() returns the internal rate of return for a series of cash flows represented by the input values.

```
RangeIRR (value[, value][, Expression])
```

RangeNPV

RangeNPV() returns the net present value of an investment based on a discount rate and a series of future periodic payments (negative values) and incomes (positive values). The result has a default number format of **money**.

```
RangeNPV (discount_rate, value[, value][, Expression])
```

RangeXIRR

RangeXIRR() returns the internal rate of return for a schedule of cash flows that is not necessarily periodic. To calculate the internal rate of return for a series of periodic cash flows, use the **RangeIRR** function.

```
RangeXIRR (values, dates[, Expression])
```

RangeXNPV

RangeXNPV() returns the net present value for a schedule of cash flows that is not necessarily periodic. The result has a default number format of money. To calculate the net present value for a series of periodic cash flows, use the **RangeNPV** function.

```
RangeXNPV (discount_rate, values, dates[, Expression])
```

See also:

Inter-record functions (page 845)

RangeAvg

RangeAvg() returns the average of a range. Input to the function can be either a range of values or an expression.

Syntax:

```
RangeAvg(first_expr[, Expression])
```

Return data type: numeric

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Limitations:

If no numeric value is found, NULL is returned.

Examples and results:

Scripting examples

Examples	Results
`RangeAvg (1,2,4)`	Returns 2.33333333

Examples	Results
`RangeAvg (1,'xyz')`	Returns 1
`RangeAvg (null( ), 'abc')`	Returns NULL

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
RangeTab3:
LOAD recno() as RangeID, RangeAvg(Field1,Field2,Field3) as MyRangeAvg INLINE [
Field1, Field2, Field3
10,5,6
2,3,7
8,2,8
18,11,9
5,5,9
9,4,2
];
```

The resulting table shows the returned values of MyRangeAvg for each of the records in the table.

Resulting table

RangeID	MyRangeAvg
1	7
2	4
3	6
4	12.666
5	6.333
6	5

Example with expression:

```
RangeAvg (Above(MyField),0,3))
```

Returns a sliding average of the result of the range of three values of **MyField** calculated on the current row and two rows above the current row. By specifying the third argument as 3, the **Above()** function returns three values, where there are sufficient rows above, which are taken as input to the **RangeAvg()** function.

Data used in examples:

*Disable sorting of **MyField** to ensure that the example works as expected.*

Sample data

MyField	RangeAvg (Above (MyField,0,3))	Comments
10	10	Because this is the top row, the range consists of one value only.
2	6	There is only one row above this row, so the range is: 10,2.
8	6.6666666667	The equivalent to RangeAvg(10,2,8)
18	9.333333333	-
5	10. 333333333	-
9	10.6666666667	-

```
RangeTab:
LOAD * INLINE [
MyField
10
2
8
18
5
9
] ;
```

See also:

Avg - chart function (page 324)
Count - chart function (page 291)

RangeCorrel

RangeCorrel() returns the correlation coefficient for two sets of data. The correlation coefficient is a measure of the relationship between the data sets.

Syntax:

```
RangeCorrel(x_value , y_value[, Expression])
```

Return data type: numeric

Data series should be entered as (x,y) pairs. For example, to evaluate two series of data, array 1 and array 2, where the array 1 = 2,6,9 and array 2 = 3,8,4 you would write `RangeCorrel (2,3,6,8,9,4)` which returns 0.269.

Arguments:

Arguments

Argument	Description
x-value, y-value	Each value represents a single value or a range of values as returned by an inter-record functions with a third optional parameter. Each value or range of values must correspond to an **x-value** or a range of **y-values**.
Expression	Optional expressions or fields containing the range of data to be measured.

Limitations:

The function needs at least two pairs of coordinates to be calculated.

Text values, NULL values and missing values return NULL.

Examples and results:

Function examples

Examples	Results
`RangeCorrel (2,3,6,8,9,4,8,5)`	Returns 0.2492. This function can be loaded in the script or added into a visualization in the expression editor.

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
RangeList:
Load * Inline [
ID1|x1|y1|x2|y2|x3|y3|x4|y4|x5|y5|x6|y6
01|46|60|70|13|78|20|45|65|78|12|78|22
02|65|56|22|79|12|56|45|24|32|78|55|15
03|77|68|34|91|24|68|57|36|44|90|67|27
04|57|36|44|90|67|27|57|68|47|90|80|94
] (delimiter is '|');

XY:
LOAD recno() as RangeID, * Inline [
X|Y
2|3
6|8
9|4
8|5
](delimiter is '|');
```

In a table with ID1 as a dimension and the measure: RangeCorrel(x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6)), the **RangeCorrel()** function finds the value of **Correl** over the range of six x,y pairs, for each of the ID1 values.

Resulting table

ID1	MyRangeCorrel
01	-0.9517
02	-0.5209
03	-0.5209
04	-0.1599

Example:

```
XY:
LOAD recno() as RangeID, * Inline [
X|Y
2|3
6|8
9|4
8|5
](delimiter is '|');
```

In a table with RangeID as a dimension and the measure: RangeCorrel(Below(X,0,4,BelowY,0,4)), the **RangeCorrel()** function uses the results of the **Below()** functions, which because of the third argument (count) set to 4, produce a range of four x-y values from the loaded table XY.

Resulting table

RangeID	MyRangeCorrel2
01	0.2492
02	-0.9959
03	-1.0000
04	-

The value for RangeID 01 is the same as manually entering RangeCorrel(2,3,6,8,9,4,8,5). For the other values of RangeID, the series produced by the Below() function are: (6,8,9,4,8,5), (9,4,8,5), and (8,5), the last of which produces a null result.

See also:

Correl - chart function (page 327)

RangeCount

RangeCount() returns the number of values, both text and numeric, in the expression or field.

Syntax:

```
RangeCount(first_expr[, Expression])
```

Return data type: integer

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be counted.
Expression	Optional expressions or fields containing the range of data to be counted.

Limitations:

NULL values are not counted.

Examples and results:

Function examples

Examples	Results
`RangeCount (1,2,4)`	Returns 3
`RangeCount (2,'xyz')`	Returns 2
`RangeCount (null( ))`	Returns 0
`RangeCount (2,'xyz', null())`	Returns 2

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
RangeTab3:
LOAD recno() as RangeID, RangeCount(Field1,Field2,Field3) as MyRangeCount INLINE [
Field1, Field2, Field3
10,5,6
2,3,7
8,2,8
18,11,9
5,5,9
9,4,2
];
```

The resulting table shows the returned values of MyRangeCount for each of the records in the table.

Results table

RangeID	MyRangeCount
1	3
2	3
3	3
4	3
5	3
6	3

Example with expression:

```
RangeCount (Above(MyField,1,3))
```

Returns the number of values contained in the three results of **MyField**. By specifying the first argument of the **Above()** function as 1 and second argument as 3, it returns the values from the first three fields above the current row, where there are sufficient rows, which are taken as input to the **RangeCount()** function.

Data used in examples:

Sample data

MyField	RangeCount(Above(MyField,1,3))
10	0
2	1
8	2
18	3
5	3
9	3

Data used in examples:

```
RangeTab:
LOAD * INLINE [
MyField
10
2
8
18
5
9
] ;
```

See also:

Count - chart function (page 291)

RangeFractile

RangeFractile() returns the value that corresponds to the n-th **fractile** (quantile) of a range of numbers.

RangeFractile() uses linear interpolation between closest ranks when calculating the fractile.

Syntax:

```
RangeFractile(fractile, first_expr[, Expression])
```

Return data type: numeric

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
fractile	A number between 0 and 1 corresponding to the fractile (quantile expressed as a fraction) to be calculated.
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Examples and results:

Function examples

Examples	Results
`RangeFractile (0.24,1,2,4,6)`	Returns 1.72
`RangeFractile(0.5,1,2,3,4,6)`	Returns 3
`RangeFractile (0.5,1,2,5,6)`	Returns 3.5

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
RangeTab:
LOAD recno() as RangeID, RangeFractile(0.5,Field1,Field2,Field3) as MyRangeFrac INLINE [
Field1, Field2, Field3
10,5,6
2,3,7
8,2,8
18,11,9
5,5,9
9,4,2
];
```

The resulting table shows the returned values of MyRangeFrac for each of the records in the table.

Resulting table

RangeID	MyRangeFrac
1	6
2	3
3	8
4	11
5	5
6	4

Example with expression:

```
RangeFractile (0.5, Above(Sum(MyField),0,3))
```

In this example, the inter-record function **Above()** contains the optional offset and count arguments. This produces a range of results that can be used as input to the any of the range functions. In this case, `Above(Sum(MyField),0,3)` returns the values of `MyField` for the current row and the two rows above. These values provide the input to the **RangeFractile()** function. So, for the bottom row in the table below, this is the equivalent of `RangeFractile(0.5, 3,4,6)`, that is, calculating the 0.5 fractile for the series 3, 4, and 6. The first two rows in the table below, the number of values in the range is reduced accordingly, where there no rows above the current row. Similar results are produced for other inter-record functions.

Sample data

MyField	RangeFractile(0.5, Above(Sum(MyField),0,3))
1	1
2	1.5
3	2
4	3
5	4
6	5

Data used in examples:

```
RangeTab:
LOAD * INLINE [
MyField
1
2
3
4
5
6
```

```
] ;
```

See also:

Above - chart function (page 848)
Fractile - chart function (page 331)

RangeIRR

RangeIRR() returns the internal rate of return for a series of cash flows represented by the input values.

The internal rate of return is the interest rate received for an investment consisting of payments (negative values) and income (positive values) that occur at regular periods.

Syntax:

```
RangeIRR(value[, value][, Expression])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	A single value or a range of values as returned by an inter record function with a third optional parameter. The function needs at least one positive and one negative value to be calculated.
Expression	Optional expressions or fields containing the range of data to be measured.

Limitations:

Text values, NULL values and missing values are disregarded.

Example table

Examples	Results
`RangeIRR(-70000,12000,15000,18000,21000,26000)`	Returns 0.0866

Examples	Results

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.
`RangeTab3:`
`LOAD *,`
`recno() as RangeID,`
`RangeIRR(Field1,Field2,Field3) as RangeIRR;`
`LOAD * INLINE [`
`Field1|Field2|Field3`
`-10000|5000|6000`
`-2000|NULL|7000`
`-8000|'abc'|8000`
`-1800|11000|9000`
`-5000|5000|9000`
`-9000|4000|2000`
`] (delimiter is '|');`
	The resulting table shows the returned values of RangeIRR for each of the records in the table.
	RangeID — RangeIRR
	1 — 0.0639
	2 — 0.8708
	3 — -
	4 — 5.8419
	5 — 0.9318
	6 — -0.2566

See also:

Inter-record functions (page 845)

RangeKurtosis

RangeKurtosis() returns the value that corresponds to the kurtosis of a range of numbers.

Syntax:

```
RangeKurtosis(first_expr[, Expression])
```

Return data type: numeric

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Limitations:

If no numeric value is found, NULL is returned.

Examples and results:

Function examples

Examples	Results
`RangeKurtosis (1,2,4,7)`	Returns -0.28571428571429

See also:

Kurtosis - chart function (page 338)

RangeMax

RangeMax() returns the highest numeric values found within the expression or field.

Syntax:

```
RangeMax(first_expr[, Expression])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Limitations:

If no numeric value is found, NULL is returned.

Examples and results:

Function examples

Examples	Results
`RangeMax (1,2,4)`	Returns 4
`RangeMax (1,'xyz')`	Returns 1
`RangeMax (null( ), 'abc')`	Returns NULL

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

`RangeTab3:`

```
LOAD recno() as RangeID, RangeMax(Field1,Field2,Field3) as MyRangeMax INLINE [
Field1, Field2, Field3
10,5,6
2,3,7
8,2,8
18,11,9
5,5,9
9,4,2
];
```

The resulting table shows the returned values of MyRangeMax for each of the records in the table.

Resulting table

RangeID	MyRangeMax
1	10
2	7
3	8
4	18
5	9
6	9

Example with expression:

```
RangeMax (Above(MyField,0,3))
```

Returns the maximum value in the range of three values of **MyField** calculated on the current row and two rows above the current row. By specifying the third argument as 3, the **Above()** function returns three values, where there are sufficient rows above, which are taken as input to the **RangeMax()** function.

Data used in examples:

*Disable sorting of **MyField** to ensure that the example works as expected.*

Sample data

MyField	RangeMax (Above(Sum(MyField),1,3))
10	10
2	10
8	10
18	18
5	18
9	18

Data used in examples:

```
RangeTab:
```

```
LOAD * INLINE [
MyField
10
2
8
18
5
9
] ;
```

RangeMaxString

RangeMaxString() returns the last value in the text sort order that it finds in the expression or field.

Syntax:

```
RangeMaxString(first_expr[, Expression])
```

Return data type: string

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Examples and results:

Function examples

Examples	Results
`RangeMaxString (1,2,4)`	Returns 4
`RangeMaxString ('xyz','abc')`	Returns 'xyz'
`RangeMaxString (5,'abc')`	Returns 'abc'
`RangeMaxString (null( ))`	Returns NULL

Example with expression:

```
RangeMaxString (Above(MaxString(MyField),0,3))
```

Returns the last (in text sort order) of the three results of the **MaxString(MyField)** function evaluated on the current row and two rows above the current row.

Data used in examples:

*Disable sorting of **MyField** to ensure that the example works as expected.*

Sample data

MyField	RangeMaxString(Above(MaxString(MyField),0,3))
10	10
abc	abc
8	abc
def	def
xyz	xyz
9	xyz

Data used in examples:

```
RangeTab:
LOAD * INLINE [
MyField
10
'abc'
8
'def'
'xyz'
9
] ;
```

See also:

MaxString - chart function (page 451)

RangeMin

RangeMin() returns the lowest numeric values found within the expression or field.

Syntax:

```
RangeMin(first_expr[, Expression])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Limitations:

If no numeric value is found, NULL is returned.

Examples and results:

Function examples

Examples	Results
`RangeMin (1,2,4)`	Returns 1
`RangeMin (1,'xyz')`	Returns 1
`RangeMin (null( ), 'abc')`	Returns NULL

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
RangeTab3:
LOAD recno() as RangeID, RangeMin(Field1,Field2,Field3) as MyRangeMin INLINE [
Field1, Field2, Field3
10,5,6
2,3,7
8,2,8
18,11,9
5,5,9
9,4,2
];
```

The resulting table shows the returned values of MyRangeMin for each of the records in the table.

Resulting table

RangeID	MyRangeMin
1	5
2	2
3	2
4	9
5	5
6	2

Example with expression:

```
RangeMin (Above(MyField,0,3)
```

Returns the minimum value in the range of three values of **MyField** calculated on the current row and two rows above the current row. By specifying the third argument as 3, the **Above()** function returns three values, where there are sufficient rows above, which are taken as input to the **RangeMin()** function.

Data used in examples:

Sample data

MyField	RangeMin(Above(MyField,0,3))
10	10
2	2
8	2
18	2
5	5
9	5

Data used in examples:

```
RangeTab:
LOAD * INLINE [
MyField
10
2
8
18
5
9
] ;
```

See also:

Min - chart function (page 278)

RangeMinString

RangeMinString() returns the first value in the text sort order that it finds in the expression or field.

Syntax:

```
RangeMinString(first_expr[, Expression])
```

Return data type: string

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Examples and results:

Function examples

Examples	Results
`RangeMinString (1,2,4)`	Returns 1
`RangeMinString ('xyz','abc')`	Returns 'abc'
`RangeMinString (5,'abc')`	Returns 5
`RangeMinString (null( ))`	Returns NULL

Example with expression:

```
RangeMinString (Above(MinString(MyField),0,3))
```

Returns the first (in text sort order) of the three results of the **MinString(MyField)** function evaluated on the current row and two rows above the current row.

Data used in examples:

*Disable sorting of **MyField** to ensure that the example works as expected.*

Sample data

MyField	RangeMinString(Above(MinString(MyField),0,3))
10	10
abc	10
8	8
def	8
xyz	8
9	9

Data used in examples:

```
RangeTab:
LOAD * INLINE [
MyField
10
'abc'
8
'def'
'xyz'
9
] ;
```

See also:

MinString - chart function (page 454)

RangeMissingCount

RangeMissingCount() returns the number of non-numeric values (including NULL) in the expression or field.

Syntax:

```
RangeMissingCount(first_expr[, Expression])
```

Return data type: integer

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be counted.
Expression	Optional expressions or fields containing the range of data to be counted.

Examples and results:

Function examples

Examples	Results
`RangeMissingCount (1,2,4)`	Returns 0
`RangeMissingCount (5,'abc')`	Returns 1
`RangeMissingCount (null( ))`	Returns 1

Example with expression:

```
RangeMissingCount (Above(MinString(MyField),0,3))
```

Returns the number of non-numeric values in the three results of the **MinString(MyField)** function evaluated on the current row and two rows above the current row.

*Disable sorting of **MyField** to ensure that the example works as expected.*

Sample data

MyField	RangeMissingCount (Above(MinString (MyField),0,3))	Explanation
10	2	Returns 2 because there are no rows above this row so 2 of the 3 values are missing.
abc	2	Returns 2 because there is only 1 row above the current row and the current row is non-numeric ('abc').
8	1	Returns 1 because 1 of the 3 rows includes a non-numeric ('abc').
def	2	Returns 2 because 2 of the 3 rows include non-numeric values ('def' and 'abc').
xyz	2	Returns 2 because 2 of the 3 rows include non-numeric values (' xyz' and 'def').
9	2	Returns 2 because 2 of the 3 rows include non-numeric values (' xyz' and 'def').

Data used in examples:

```
RangeTab:
LOAD * INLINE [
MyField
10
'abc'
8
'def'
'xyz'
9
] ;
```

See also:

MissingCount - chart function (page 294)

RangeMode

RangeMode() finds the most commonly occurring value (mode value) in the expression or field.

Syntax:

```
RangeMode(first_expr {, Expression})
```

Return data type: numeric

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Limitations:

If more than one value shares the highest frequency, NULL is returned.

Examples and results:

Function examples

Examples	Results
`RangeMode (1,2,9,2,4)`	Returns 2
`RangeMode ('a',4,'a',4)`	Returns NULL
`RangeMode (null( ))`	Returns NULL

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
RangeTab3:
LOAD recno() as RangeID, RangeMode(Field1,Field2,Field3) as MyRangeMode INLINE [
Field1, Field2, Field3
10,5,6
2,3,7
8,2,8
18,11,9
5,5,9
9,4,2
];
```

The resulting table shows the returned values of **MyRangeMode** for each of the records in the table.

Results table

RangeID	MyRangMode
1	-
2	-
3	8
4	-
5	5
6	-

Example with expression:

```
RangeMode (Above(MyField,0,3))
```

Returns the most commonly occurring value in the three results of **MyField** evaluated on the current row and two rows above the current row. By specifying the third argument as 3, the **Above()** function returns three values, where there are sufficient rows above, which are taken as input to the **RangeMode()** function.

Data used in example:

```
RangeTab:
LOAD * INLINE [
MyField
10
2
8
18
5
9
] ;
```

*Disable sorting of **MyField** to ensure that the example works as expected.*

Sample data

MyField	RangeMode(Above(MyField,0,3))
10	Returns 10 because there are no rows above so the single value is the most commonly occurring.
2	-
8	-
18	-
5	-
9	-

See also:

Mode - chart function (page 281)

RangeNPV

RangeNPV() returns the net present value of an investment based on a discount rate and a series of future periodic payments (negative values) and incomes (positive values). The result has a default number format of **money**.

For cash flows that are not necessarily periodic, see *RangeXNPV (page 937)*.

Syntax:

```
RangeNPV(discount_rate, value[,value][, Expression])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
discount_rate	The interest rate per period.
value	A payment or income occurring at the end of each period. Each value may be a single value or a range of values as returned by an inter-record function with a third optional parameter.
Expression	Optional expressions or fields containing the range of data to be measured.

Limitations:

Text values, NULL values and missing values are disregarded.

Examples	Results
`RangeNPV(0.1,-10000,3000,4200,6800)`	Returns 1188.44

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.
`RangeTab3:`
`LOAD *,`
`recno() as RangeID,`
`RangeNPV(Field1,Field2,Field3) as RangeNPV;`
`LOAD * INLINE [`
`Field1|Field2|Field3`
`10|5|-6000`
`2|NULL|7000`
`8|'abc'|8000`
`18|11|9000`
`5|5|9000`
`9|4|2000`
`] (delimiter is '|');`
	The resulting table shows the returned values of RangeNPV for each of the records in the table.
	RangeID — RangeNPV
	1 — $-49.13
	2 — $777.78
	3 — $98.77
	4 — $25.51
	5 — $250.83
	6 — $20.40

See also:

Inter-record functions (page 845)

RangeNullCount

RangeNullCount() finds the number of NULL values in the expression or field.

Syntax:

```
RangeNullCount(first_expr [, Expression])
```

Return data type: integer

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Examples and results:

Function examples

Examples	Results
`RangeNullCount (1,2,4)`	Returns 0
`RangeNullCount (5,'abc')`	Returns 0
`RangeNullCount (null( ), null( ))`	Returns 2

Example with expression:

```
RangeNullCount (Above(Sum(MyField),0,3))
```

Returns the number of NULL values in the three results of the **Sum(MyField)** function evaluated on the current row and two rows above the current row.

*Copying **MyField** in example below will not result in NULL value.*

Sample data

MyField	RangeNullCount(Above(Sum(MyField),0,3))
10	Returns 2 because there are no rows above this row so 2 of the 3 values are missing (=NULL).
'abc'	Returns 1 because there is only one row above the current row, so one of the three values is missing (=NULL).
8	Returns 0 because none of the three rows is a NULL value.

Data used in examples:

```
RangeTab:
LOAD * INLINE [
MyField
10
'abc'
8
] ;
```

See also:

NullCount - chart function (page 297)

RangeNumericCount

RangeNumericCount() finds the number of numeric values in an expression or field.

Syntax:

```
RangeNumericCount(first_expr[, Expression])
```

Return data type: integer

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Examples and results:

Function examples

Examples	Results
`RangeNumericCount (1,2,4)`	Returns 3
`RangeNumericCount (5,'abc')`	Returns 1
`RangeNumericCount (null( ))`	Returns 0

Example with expression:

```
RangeNumericCount (Above(MaxString(MyField),0,3))
```

Returns the number of numeric values in the three results of the **MaxString(MyField)** function evaluated on the current row and two rows above the current row.

*Disable sorting of **MyField** to ensure that the example works as expected.*

Sample data

MyField	RangeNumericCount(Above(MaxString(MyField),0,3))
10	1

MyField	RangeNumericCount(Above(MaxString(MyField),0,3))
abc	1
8	2
def	1
xyz	1
9	1

Data used in examples:

```
RangeTab:
LOAD * INLINE [
MyField
10
'abc'
8
def
xyz
9
] ;
```

See also:

NumericCount - chart function (page 300)

RangeOnly

RangeOnly() is a dual function that returns a value if the expression evaluates to one unique value. If this is not the case then **NULL** is returned.

Syntax:

```
RangeOnly(first_expr[, Expression])
```

Return data type: dual

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Examples and results:

Examples	Results
`RangeOnly (1,2,4)`	Returns NULL
`RangeOnly (5,'abc')`	Returns NULL
`RangeOnly (null( ), 'abc')`	Returns 'abc'
`RangeOnly(10,10,10)`	Returns 10

See also:

Only - chart function (page 284)

RangeSkew

RangeSkew() returns the value corresponding to the skewness of a range of numbers.

Syntax:

```
RangeSkew(first_expr[, Expression])
```

Return data type: numeric

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Limitations:

If no numeric value is found, NULL is returned.

Examples and results:

Function examples

Examples	Results
`rangeskew (1,2,4)`	Returns 0.93521952958283
`rangeskew (above (SalesValue,0,3))`	Returns a sliding skewness of the range of three values returned from the above() function calculated on the current row and the two rows above the current row.

Data used in example:

Sample data

CustID	RangeSkew(Above(SalesValue,0,3))
1-20	-, -, 0.5676, 0.8455, 1.0127, -0.8741, 1.7243, -1.7186, 1.5518, 1.4332, 0, 1.1066, 1.3458, 1.5636, 1.5439, 0.6952, -0.3766

```
SalesTable:
LOAD recno() as CustID, * inline [
SalesValue
101
163
126
139
167
86
83
22
32
70
108
124
176
113
95
32
42
92
61
21
] ;
```

See also:

Skew - chart function (page 367)

RangeStdev

RangeStdev() finds the standard deviation of a range of numbers.

Syntax:

```
RangeStdev(first_expr[, Expression])
```

Return data type: numeric

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Limitations:

If no numeric value is found, NULL is returned.

Examples and results:

Function examples

Examples	Results
`RangeStdev (1,2,4)`	Returns 1.5275252316519
`RangeStdev (null( ))`	Returns NULL
`RangeStdev (above (SalesValue),0,3))`	Returns a sliding standard of the range of three values returned from the above() function calculated on the current row and the two rows above the current row.

Data used in example:

Sample data

CustID	RangeStdev(SalesValue, 0,3))
1-20	-,43.841, 34.192, 18.771, 20.953, 41.138, 47.655, 36.116, 32.716, 25.325, 38,000, 27.737, 35.553, 33.650, 42.532, 33.858, 32.146, 25.239, 35.595

```
SalesTable:
LOAD recno() as CustID, * inline [
SalesValue
101
163
126
139
167
86
83
22
32
70
108
124
176
113
95
32
42
92
61
21
```

```
] ;
```

See also:

Stdev - chart function (page 370)

RangeSum

RangeSum() returns the sum of a range of values. All non-numeric values are treated as 0.

Syntax:

```
RangeSum(first_expr[, Expression])
```

Return data type: numeric

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Arguments

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Limitations:

The **RangeSum** function treats all non-numeric values as 0.

Examples and results:

Examples

Examples	Results
`RangeSum (1,2,4)`	Returns 7
`RangeSum (5,'abc')`	Returns 5
`RangeSum (null( ))`	Returns 0

Example:

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.

```
RangeTab3:
LOAD recno() as RangeID, Rangesum(Field1,Field2,Field3) as MyRangeSum INLINE [
Field1, Field2, Field3
10,5,6
2,3,7
8,2,8
18,11,9
```

```
5,5,9
9,4,2
];
```

The resulting table shows the returned values of MyRangeSum for each of the records in the table.

Resulting table

RangeID	MyRangeSum
1	21
2	12
3	18
4	38
5	19
6	15

Example with expression:

```
RangeSum (Above(MyField,0,3))
```

Returns the sum of the three values of **MyField)**: from the current row and two rows above the current row. By specifying the third argument as 3, the **Above()** function returns three values, where there are sufficient rows above, which are taken as input to the **RangeSum()** function.

Data used in examples:

*Disable sorting of **MyField** to ensure that the example works as expected.*

Sample data

MyField	RangeSum(Above(MyField,0,3))
10	10
2	12
8	20
18	28
5	31
9	32

Data used in examples:

```
RangeTab:
LOAD * INLINE [
MyField
10
2
8
18
```

```
5
9
] ;
```

See also:

Sum - chart function (page 286)
Above - chart function (page 848)

RangeTextCount

RangeTextCount() returns the number of text values in an expression or field.

Syntax:

```
RangeTextCount(first_expr[, Expression])
```

Return data type: integer

Arguments:

The arguments of this function may contain inter-record functions which in themselves return a list of values.

Argument

Argument	Description
first_expr	The expression or field containing the data to be measured.
Expression	Optional expressions or fields containing the range of data to be measured.

Examples and results:

Function examples

Examples	Results
`RangeTextCount (1,2,4)`	Returns 0
`RangeTextCount (5,'abc')`	Returns 1
`RangeTextCount (null( ))`	Returns 0

Example with expression:

```
RangeTextCount (Above(MaxString(MyField),0,3))
```

Returns the number of text values within the three results of the **MaxString(MyField)** function evaluated over the current row and two rows above the current row.

Data used in examples:

*Disable sorting of **MyField** to ensure that the example works as expected.*

Example data

MyField	MaxString(MyField)	RangeTextCount(Above(Sum(MyField),0,3))
10	10	0
abc	abc	1
8	8	1
def	def	2
xyz	xyz	2
9	9	2

Data used in examples:

```
RangeTab:
LOAD * INLINE [
MyField
10
'abc'
8
null()
'xyz'
9
] ;
```

See also:

TextCount - chart function (page 303)

RangeXIRR

RangeXIRR() returns the internal rate of return for a schedule of cash flows that is not necessarily periodic. To calculate the internal rate of return for a series of periodic cash flows, use the **RangeIRR** function.

Syntax:

```
RangeXIRR(value, date{, value, date})
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
value	A cash flow or a series of cash flows that correspond to a schedule of payments in dates. The series of values must contain at least one positive and one negative value.
date	A payment date or a schedule of payment dates that corresponds to the cash flow payments.

Limitations:

Text values, NULL values and missing values are disregarded.

All payments are discounted based on a 365-day year.

Examples	Results
`RangeXIRR(-2500,'2008-01-01',2750,'2008-09-01')`	Returns 0.1532

See also:

RangeIRR (page 913)

RangeXNPV

RangeXNPV() returns the net present value for a schedule of cash flows that is not necessarily periodic. The result has a default number format of money. To calculate the net present value for a series of periodic cash flows, use the **RangeNPV** function.

Syntax:

```
RangeXNPV(discount_rate, values, dates[, Expression])
```

Return data type: numeric

Arguments:

Arguments

Argument	Description
discount_rate	The interest rate per period.
values	A cash flow or a series of cash flows that corresponds to a schedule of payments in dates. Each value may be a single value or a range of values as returned by an inter-record function with a third optional parameter. The series of values must contain at least one positive and one negative value.
dates	A payment date or a schedule of payment dates that corresponds to the cash flow payments.

Limitations:

Text values, NULL values and missing values are disregarded.

All payments are discounted based on a 365-day year.

Example table

Examples	Results
`RangeXNPV(0.1, -2500,'2008-01-01',2750,'2008-09-01')`	Returns 80.25

Add the example script to your app and run it. To see the result, add the fields listed in the results column to a sheet in your app.
`RangeTab3:`
`LOAD *,`
`recno() as RangeID,`
`RangeXNPV(Field1,Field2,Field3) as RangeNPV;`
`LOAD * INLINE [`
`Field1|Field2|Field3`
`10|5|-6000`
`2|NULL|7000`
`8|'abc'|8000`
`18|11|9000`
`5|5|9000`
`9|4|2000`
`] (delimiter is '|');`
	The resulting table shows the returned values of RangeXNPV for each of the records in the table.
	RangeID — RangeXNPV
	1 — $-49.13
	2 — $777.78
	3 — $98.77
	4 — $25.51
	5 — $250.83
	6 — $20.40

5.22 Ranking and clustering functions

These functions can only be used in chart expressions.

Ranking functions in charts

Suppression of zero values is automatically disabled when these functions are used. NULL values are disregarded.

Rank

Rank() evaluates the rows of the chart in the expression, and for each row, displays the relative position of the value of the dimension evaluated in the expression. When evaluating the expression, the function compares the result with the result of the other rows containing the current column segment and returns the ranking of the current row within the segment.

```
Rank - chart function([TOTAL [<fld {, fld}>]] expr[, mode[, fmt]])
```

HRank

HRank() evaluates the expression, and compares the result with the result of the other columns containing the current row segment of a pivot table. The function then returns the ranking of the current column within the segment.

```
HRank - chart function([TOTAL] expr[, mode[, fmt]])
```

Clustering functions in charts

KMeans2D

The property group **Site license** contains properties related to the license for the Qlik Sense system. All fields are mandatory and must not be empty.

Site licence properties

Property name	Description
Owner name	The user name of the Qlik Sense product owner.
Owner organization	The name of the organization that the Qlik Sense product owner is a member of.
Serial number	The serial number assigned to the Qlik Sense software.
Control number	The control number assigned to the Qlik Sense software.
LEF access	The License Enabler File (LEF) assigned to the Qlik Sense software.

KMeans2D() evaluates the rows of the chart by applying k-means clustering, and for each chart row displays the cluster id of the cluster this data point has been assigned to. The columns that are used by the clustering algorithm are determined by the parameters coordinate_1, and coordinate_2, respectively. These are both aggregations. The number of clusters that are created is determined by the num_clusters parameter. Data can be optionally normalized by the norm parameter.

```
KMeans2D - chart function(num_clusters, coordinate_1, coordinate_2 [, norm])
```

KMeansND

KMeansND() evaluates the rows of the chart by applying k-means clustering, and for each chart row displays the cluster id of the cluster this data point has been assigned to. The columns that are used by the clustering algorithm are determined by the parameters coordinate_1, and coordinate_2, etc., up to n columns. These are all aggregations. The number of clusters that are created is determined by the num_clusters parameter.

```
KMeansND - chart function(num_clusters, num_iter, coordinate_1, coordinate_2
[,coordinate_3 [, ...]])
```

KMeansCentroid2D

KMeansCentroid2D() evaluates the rows of the chart by applying k-means clustering, and for each chart row displays the desired coordinate of the cluster this data point has been assigned to. The columns that are used by the clustering algorithm are determined by the parameters coordinate_1, and coordinate_2, respectively. These are both aggregations. The number of clusters that are created is determined by the num_clusters parameter. Data can be optionally normalized by the norm parameter.

```
KMeansCentroid2D - chart function(num_clusters, coordinate_no, coordinate_1,
coordinate_2 [, norm])
```

KMeansCentroidND

KMeansCentroidND() evaluates the rows of the chart by applying k-means clustering, and for each chart row displays the desired coordinate of the cluster this data point has been assigned to. The columns that are used by the clustering algorithm are determined by the parameters coordinate_1, coordinate_2, etc., up to n

columns. These are all aggregations. The number of clusters that are created is determined by the num_clusters parameter.

```
KMeansCentroidND - chart function(num_clusters, num_iter, coordinate_no,
coordinate_1, coordinate_2 [,coordinate_3 [, ...]])
```

Rank - chart function

Rank() evaluates the rows of the chart in the expression, and for each row, displays the relative position of the value of the dimension evaluated in the expression. When evaluating the expression, the function compares the result with the result of the other rows containing the current column segment and returns the ranking of the current row within the segment.

Column segments

Region	Country	Population	Rank(Population)
Americas	Mexico	128,932,753	2
Americas	Canada	37,742,154	3
Americas	United States of America	331,002,651	1
Europe	Sweden	10,099,265	4
Europe	United Kingdom	67,886,011	2
Europe	France	65,273,511	3
Europe	Germany	83,783,942	1

For charts other than tables, the current column segment is defined as it appears in the chart's straight table equivalent.

Syntax:

```
Rank([TOTAL] expr[, mode[, fmt]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
mode	Specifies the number representation of the function result.
fmt	Specifies the text representation of the function result.
TOTAL	If the chart is one-dimensional, or if the expression is preceded by the **TOTAL** qualifier, the function is evaluated along the entire column. If the table or table equivalent has multiple vertical dimensions, the current column segment will include only rows with the same values as the current row in all dimension columns except for the column showing the last dimension in the inter-field sort order.

The ranking is returned as a dual value, which in the case when each row has a unique ranking, is an integer between 1 and the number of rows in the current column segment.

In the case where several rows share the same ranking, the text and number representation can be controlled with the **mode** and **fmt** parameters.

mode

The second argument, **mode**, can take the following values:

mode examples

Value	Description
0 (default)	If all ranks within the sharing group fall on the low side of the middle value of the entire ranking, all rows get the lowest rank within the sharing group. If all ranks within the sharing group fall on the high side of the middle value of the entire ranking, all rows get the highest rank within the sharing group. If ranks within the sharing group span over the middle value of the entire ranking, all rows get the value corresponding to the average of the top and bottom ranking in the entire column segment.
1	Lowest rank on all rows.
2	Average rank on all rows.
3	Highest rank on all rows.
4	Lowest rank on first row, then incremented by one for each row.

fmt

The third argument, **fmt**, can take the following values:

fmtexamples

Value	Description
0 (default)	Low value **-** high value on all rows (for example 3 - 4).
1	Low value on all rows.
2	Low value on first row, blank on the following rows.

The order of rows for **mode** 4 and **fmt** 2 is determined by the sort order of the chart dimensions.

Examples and results:

Create two visualizations from the dimensions Product and Sales and another from Product and UnitSales. Add measures as shown in the following table.

Rank examples

Examples	Results
Example 1. Create a table with the dimensions `Customer` and `Sales` and the measure `Rank(Sales)`	The result depends on the sort order of the dimensions. If the table is sorted on Customer, the table lists all the values of Sales for Astrida, then Betacab, and so on. The results for Rank (Sales) will show 10 for the Sales value 12, 9 for the Sales value 13, and so on, with the rank value of 1 returned for the Sales value 78. The next column segment begins with Betacab, for which the first value of Sales in the segment is 12. The rank value of Rank(Sales) for this is given as 11. If the table is sorted on Sales, the column segments consist of the values of Sales and the corresponding Customer. Because there are two Sales values of 12 (for Astrida and Betacab), the value of Rank(Sales) for that column segment is 1-2, for each value of Customer. This is because there are two values of Customer for the Sales value 12. If there had been 4 values, the result would be 1-4, for all rows. This shows what the result looks like for the default value (0) of the argument fmt.
Example 2. Replace the dimension Customer with Product and add the measure `Rank(Sales,1,2)`	This returns 1 on the first row on each column segment and leaves all other rows blank, because arguments **mode** and **fmt** are set to 1 and 2 respectively.

Results for example 1, with table sorted on Customer:

Results table

Customer	Sales	Rank(Sales)
Astrida	12	10
Astrida	13	9
Astrida	20	8
Astrida	22	7
Astrida	45	6
Astrida	46	5
Astrida	60	4
Astrida	65	3
Astrida	70	2
Astrida	78	1
Betcab	12	11

Results for example 1, with table sorted on Sales:

Results table

Customer	Sales	Rank(Sales)
Astrida	12	1-2
Betacab	12	1-2
Astrida	13	1
Betacab	15	1
Astrida	20	1
Astrida	22	1-2
Betacab	22	1-2
Betacab	24	1-2
Canutility	24	1-2

Data used in examples:

```
ProductData:
Load * inline [
Customer|Product|UnitSales|UnitPrice
Astrida|AA|4|16
Astrida|AA|10|15
Astrida|BB|9|9
Betacab|BB|5|10
Betacab|CC|2|20
Betacab|DD|0|25
Canutility|AA|8|15
Canutility|CC|0|19
] (delimiter is '|');

Sales2013:
crosstable (Month, Sales) LOAD * inline [
Customer|Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec
Astrida|46|60|70|13|78|20|45|65|78|12|78|22
Betacab|65|56|22|79|12|56|45|24|32|78|55|15
Canutility|77|68|34|91|24|68|57|36|44|90|67|27
Divadip|57|36|44|90|67|27|57|68|47|90|80|94
] (delimiter is '|');
```

See also:

Sum - chart function (page 286)

HRank - chart function

HRank() evaluates the expression, and compares the result with the result of the other columns containing the current row segment of a pivot table. The function then returns the ranking of the current column within the segment.

Syntax:

```
HRank([ TOTAL ] expr [ , mode [, fmt ] ])
```

Return data type: dual

This function only works in pivot tables. In all other chart types it returns NULL.

Arguments:

Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
mode	Specifies the number representation of the function result.
fmt	Specifies the text representation of the function result.
TOTAL	If the chart is one-dimensional, or if the expression is preceded by the **TOTAL** qualifier, the function is evaluated along the entire column. If the table or table equivalent has multiple vertical dimensions, the current column segment will include only rows with the same values as the current row in all dimension columns except for the column showing the last dimension in the inter-field sort order.

If the pivot table is one-dimensional or if the expression is preceded by the **total** qualifier, the current row segment is always equal to the entire row. If the pivot table has multiple horizontal dimensions, the current row segment will include only columns with the same values as the current column in all dimension rows except for the row showing the last horizontal dimension of the inter-field sort order.

The ranking is returned as a dual value, which in the case when each column has a unique ranking will be an integer between 1 and the number of columns in the current row segment.

In the case where several columns share the same ranking, the text and number representation can be controlled with the **mode** and **format** arguments.

The second argument, **mode**, specifies the number representation of the function result:

mode examples

Value	Description
0 (default)	If all ranks within the sharing group fall on the low side of the middle value of the entire ranking, all columns get the lowest rank within the sharing group. If all ranks within the sharing group fall on the high side of the middle value of the entire ranking, all columns get the highest rank within the sharing group. If ranks within the sharing group span over the middle value of the entire ranking, all rows get the value corresponding to the average of the top and bottom ranking in the entire column segment.
1	Lowest rank on all columns in the group.
2	Average rank on all columns in the group.
3	Highest rank on all columns in the group.
4	Lowest rank on first column, then incremented by one for each column in the group.

The third argument, **format**, specifies the text representation of the function result:

format examples

Value	Description
0 (default)	Low value**&' - '&**high value on all columns in the group (for example 3 - 4).
1	Low value on all columns in the group.
2	Low value on first column, blank on the following columns in the group.

The order of columns for **mode** 4 and **format** 2 is determined by the sort order of the chart dimensions.

Examples:

```
HRank( sum( Sales ))
HRank( sum( Sales ), 2 )
HRank( sum( Sales ), 0, 1 )
```

Optimizing with k-means: A real-world example

The following example illustrates a real world use case where the KMeans clustering and Centroid functions are applied to a dataset. The KMeans function segregates data points into clusters that share similarities. The clusters become more compact and differentiated as the KMeans algorithm is applied over a configurable number of iterations.

KMeans is used across many fields in a wide variety of use cases; some examples of clustering use cases include customer segmentation, fraud detection, predicting account attrition, targeting client incentives, cybercrime identification, and delivery route optimization. The KMeans clustering algorithm is increasingly being used where enterprises are trying to infer patterns and optimize service offerings.

Qlik Sense KMeans and Centroid functions

Qlik Sense provides two KMeans functions that group data points into clusters based on similarity. See *KMeans2D - chart function (page 954)* and *KMeansND - chart function (page 969)*. The **KMeans2D** function accepts two dimensions and works well for visualizing results through a **scatter plot** chart. The **KMeansND** function accepts more than two dimensions. As it is easy to conceptualize a 2D outcome on standard charts, the following demonstration applies KMeans on a **scatter plot** chart using two dimensions. KMeans clustering can be visualized through coloring by expression; or by dimension as described in this example.

Qlik Sense centroid functions determine the arithmetic mean position of all the data points in the cluster and identify a central point, or centroid for that cluster. For each chart row (or record), the centroid function displays the coordinate of the cluster this data point has been assigned to. See *KMeansCentroid2D - chart function (page 984)* and *KMeansCentroidND - chart function (page 985)*.

Use case and example overview

The following example stages through a simulated real world scenario. A textile company in New York state, USA, must decrease expenses by minimizing delivery costs. One way to do that is to relocate warehouses closer to their distributors. The company employs 118 distributors across the state of New York. The following demonstration simulates how an operations manager could segment distributors into five clustered geographies using the KMeans function and then identify five optimal warehouse locations central to those clusters using the centroid function. The objective is to discover mapping coordinates that can be used to identify five central warehouse locations.

The dataset

The dataset is based on randomly generated names and addresses in New York state with real latitude and longitude coordinates. The dataset contains the following ten columns: id, first_name, last_name, telephone, address, city, state, zip, latitude, longitude. The dataset is available below as a file you can download locally and then upload to Qlik Sense or inline for data load editor. The app being created is named *Distributors KMeans and Centroid* and the first sheet in the app is named *Distribution cluster analysis*.

Select the following link to download the sample data file: DistributorData.csv

Distributor dataset: Inline load for data load editor in Qlik Sense (page 952)

Title: DistributorData

Total number of records: 118

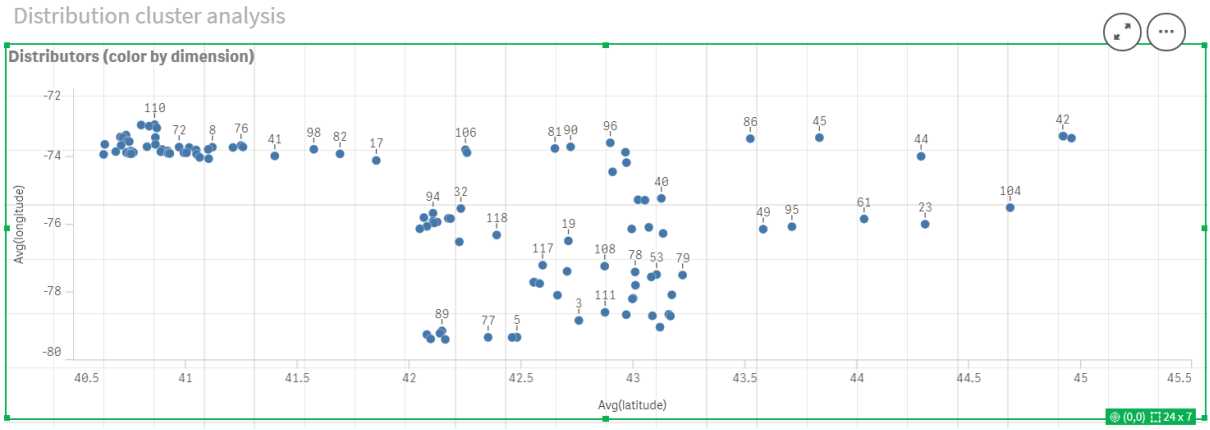
Applying the KMeans2D function

In this example, configuration of a **scatter plot** chart is demonstrated using the *DistributorData* dataset, the **KMeans2D** function is applied, and the chart is colored by dimension.

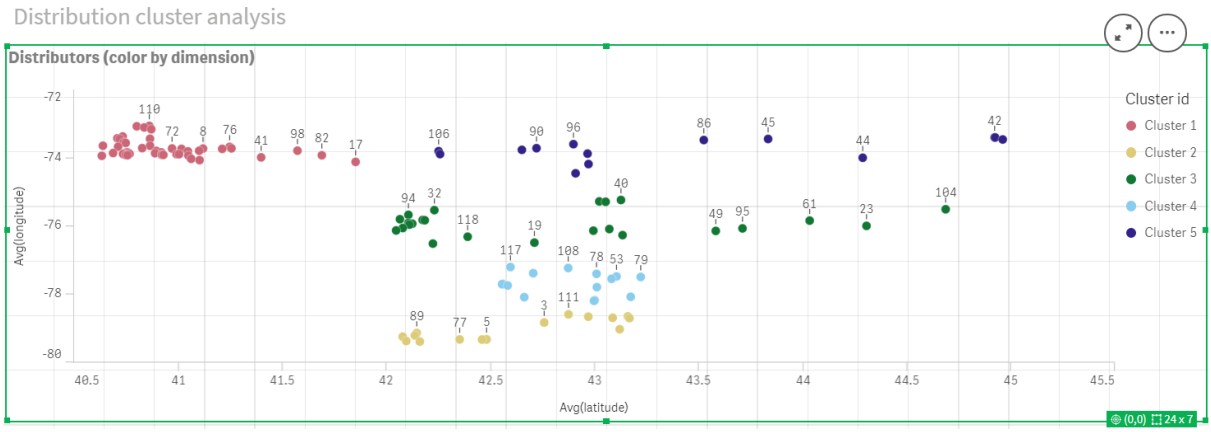
Note that Qlik Sense KMeans functions support auto-clustering using a method called depth difference (DeD). When a user sets 0 for the number of clusters, the optimal number of clusters for that dataset is determined. For this example however, a variable is created for the **num_clusters** argument (refer to *KMeans2D - chart function (page 954)* for syntax). Therefore, the desired number of clusters (k=5) is specified by a variable.

1. A **scatter plot** chart is dragged onto the sheet and named *Distributors (by dimension)*.
2. A **variable** is created to specify the number of clusters. The **variable** is named *vDistClusters*. For the variable **Definition**, *5* is entered.
3. **Data** configuration for the chart:
   a. Under **Dimensions**, *id* field is selected for **Bubble**. *Cluster id* is entered for the **Label**.
   b. Under **Measures**, *Avg([latitude])* is the expression for **X-axis**.
   c. Under **Measures**, *Avg([longitude])* is the expression for **Y-axis**.
4. **Appearance** configuration:
   a. Under **Colors and legend**, **Custom** is chosen for **Colors**.
   b. **By dimension** is selected for coloring the chart.
   c. The following expression is entered: *=pick(aggr(KMeans2D(vDistClusters,only(latitude),only (longitude)),id)+1, 'Cluster 1', 'Cluster 2', 'Cluster 3', 'Cluster 4', 'Cluster 5')*
   d. The checkbox for **Persistent colors** is selected.

Scatter plot before KMeans coloring by dimension is applied



Scatter plot after KMeans coloring by dimension is applied



Adding a **table**: *Distributors*

It can be helpful to have a table handy for quick access to relevant data. The **scatter plot** chart shows *ids* though a table with corresponding distributor names is added for reference.

1. A **table** named *Distributors* is dragged onto the sheet with the following **Columns** (Dimensions) added: *id*, *first_name*, and *last_name*.

Table: Distributor names

Distributors

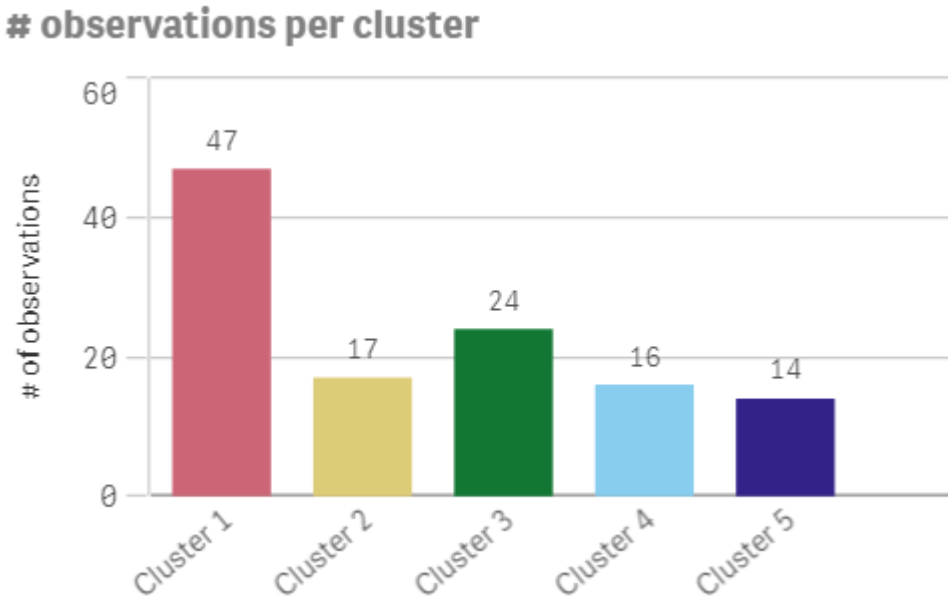
id	first_name	last_name
1	Kaiya	Snow
2	Dean	Roy
3	Eden	Paul
4	Bryanna	Higgins
5	Elisabeth	Lee
6	Skylar	Robinson
7	Cody	Bailey
8	Dario	Sims
9	Deacon	Hood

Adding a **bar chart**: *# observations per cluster*

For the warehouse distribution scenario, it is helpful to know how many distributors will be served by each warehouse. Therefore, a **bar chart** is created that measures how many distributors are assigned to each cluster.

1. A **bar chart** is dragged onto the sheet. The chart is named: *# observations per cluster*.
2. **Data** configuration for the **bar chart**:
   a. A **Dimension** labeled *Clusters* is added (the label can be added after the expression is applied). The following expression is entered: *=pick(aggr(KMeans2D(vDistClusters,only(latitude),only (longitude)),id)+1, 'Cluster 1', 'Cluster 2', 'Cluster 3', 'Cluster 4', 'Cluster 5')*
   b. A **Measure** labeled *# of observations* is added. The following expression is entered: *=count(aggr (KMeans2D(vDistClusters,only(latitude),only(longitude)),id))*
3. **Appearance** configuration:
   a. Under **Colors and legend**, **Custom** is chosen for **Colors**.
   b. **By dimension** is selected for coloring the chart.
   c. The following expression is entered: *=pick(aggr(KMeans2D(vDistClusters,only(latitude),only (longitude)),id)+1, 'Cluster 1', 'Cluster 2', 'Cluster 3', 'Cluster 4', 'Cluster 5')*
   d. The checkbox for **Persistent colors** is selected.
   e. **Show legend** is turned off.
   f. Under **Presentation**, **Value labels** is toggled to **Auto**.
   g. Under **X-axis**: **Clusters**, **Labels only** is selected.

Bar chart: # observations per cluster



Applying the **Centroid2D** function

A second table is added for the **Centroid2D** function that will identify the coordinates for potential warehouse locations. This table shows the central location (centroid values) for the five identified distributor groups.

1. A **Table** is dragged onto the sheet and named *Cluster centroids* with the following columns added::
   a. A **Dimension** labeled *Clusters* is added. The following expression is entered:=*pick(aggr (KMeans2D(vDistClusters,only(latitude),only(longitude)),id)+1,'Warehouse 1','Warehouse 2','Warehouse 3','Warehouse 4','Warehouse 5'*)
   b. A **Measure** labeled *latitude (D1)* is added. The following expression is entered:=*only(aggr (KMeansCentroid2D(vDistClusters,0,only(latitude),only(longitude)),id))*
   Note the parameter **coordinate_no** corresponds to the first dimension(0). In this case the dimension *latitude* is plotted against the x-axis. If we were working with the **CentroidND** function and there were up to six dimensions, these parameter entries could be any of six values: 0,1,2,3,4,or 5.
   c. A **Measure** labeled *longitude (D2)* is added. The following expression is entered:=*only(aggr (KMeansCentroid2D(vDistClusters,1,only(latitude),only(longitude)),id))*
   The parameter **coordinate_no** in this expression corresponds to the second dimension(1). The dimension *longitude* is plotted against the y-axis.

Table: Cluster centroid calculations

Cluster centroids		
Clusters	latitude (D1)	longitude (D2)
Totals	-	-
Warehouse 1	40.945422240426	-73.719966482979
Warehouse 2	42.590538729412	-79.067889217647
Warehouse 3	42.805089516667	-75.901621883333
Warehouse 4	42.8581692625	-77.6800485875
Warehouse 5	43.436770771429	-73.734622635714

Centroid mapping

The next step is to map the centroids. It is up to the app developer if they prefer to place the visualization on separate sheets.

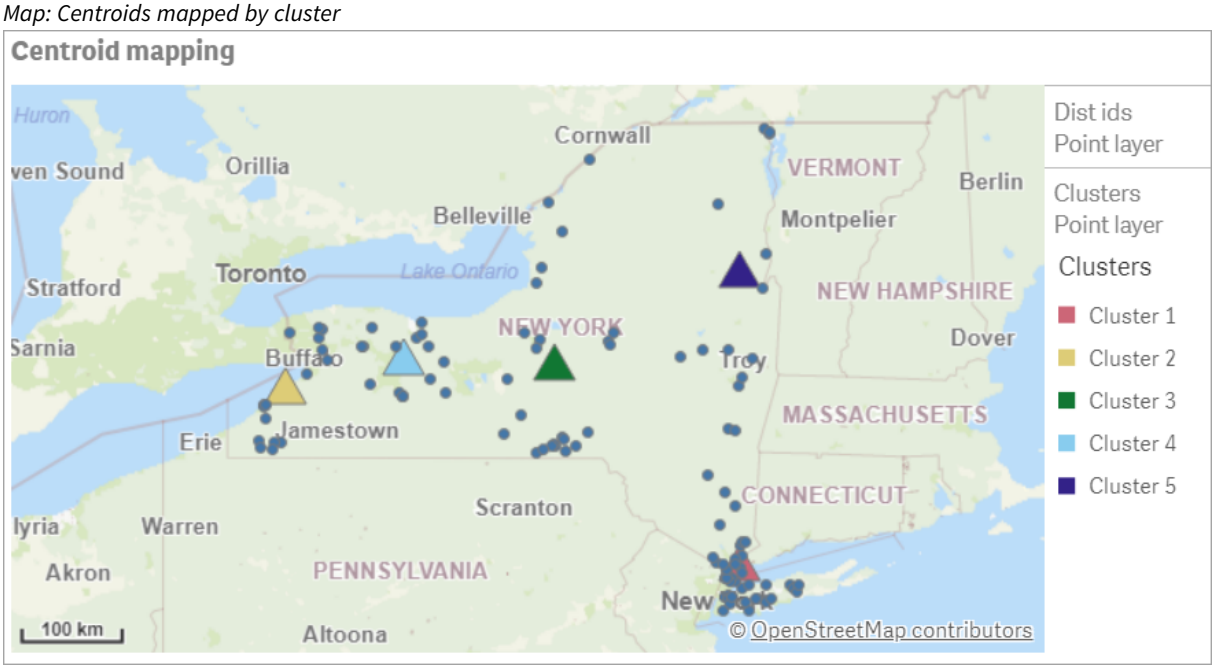
1. A **map** named *Centroid mapping* is dragged onto the sheet.
2. In the **Layers** section. **Add layer** is selected, then **Point layer** is selected.
   a. The **Field** *id* is selected and *Dist ids* **Label** is added.
   b. In the **Location** section, the checkbox for **Latitude and Longitude fields** is selected.
   c. For **Latitude**, the *latitude* field is selected.
   d. For **Longitude**, the *longitude* field is selected.
   e. In the **Size & Shape** section, **Bubble** is selected for **Shape**, and the **Size** is decreased to preference on the slider.
   f. In the **Colors** section, **Single color** is selected and blue is selected for the **Color** and grey for the **Outline** color (these choices are also a matter of preference).
3. In the **Layers** section, a second **Point layer** is added by selecting **Add layer** and then selecting **Point layer**.
   a. The following expression is entered: *=aggr(KMeans2D(vDistClusters,only(latitude),only (longitude)),id)*
   b. The **Label** *Clusters* is added.
   c. In the **Location** section, the checkbox for **Latitude and Longitude fields** is selected.
   d. For **Latitude** which in this case is plotted along the x-axis, the following expression is added: *=aggr(KMeansCentroid2D(vDistClusters,0,only(latitude),only(longitude)),id)*
   e. For **Longitude** which in this case is plotted along the y-axis, the following expression is added: *=aggr(KMeansCentroid2D(vDistClusters,1,only(latitude),only(longitude)),id)*
   f. In the **Size & Shape** section, **Triangle** is selected for **Shape**, and the **Size** is decreased on the slider to preference.
   g. Under **Colors and legend**, **Custom** is selected for **Colors**.

h. **By dimension** is selected for coloring the chart. The following expression is entered: *=pick(aggr (KMeans2D(vDistClusters,only(latitude),only(longitude)),id)+1,'Cluster 1','Cluster 2','Cluster 3','Cluster 4','Cluster 5')*

i. The dimension is labeled *Clusters*.

4. In **Map settings**, **Adaptive** is selected for **Projection**. **Metric** is selected for **Units of measurement.**

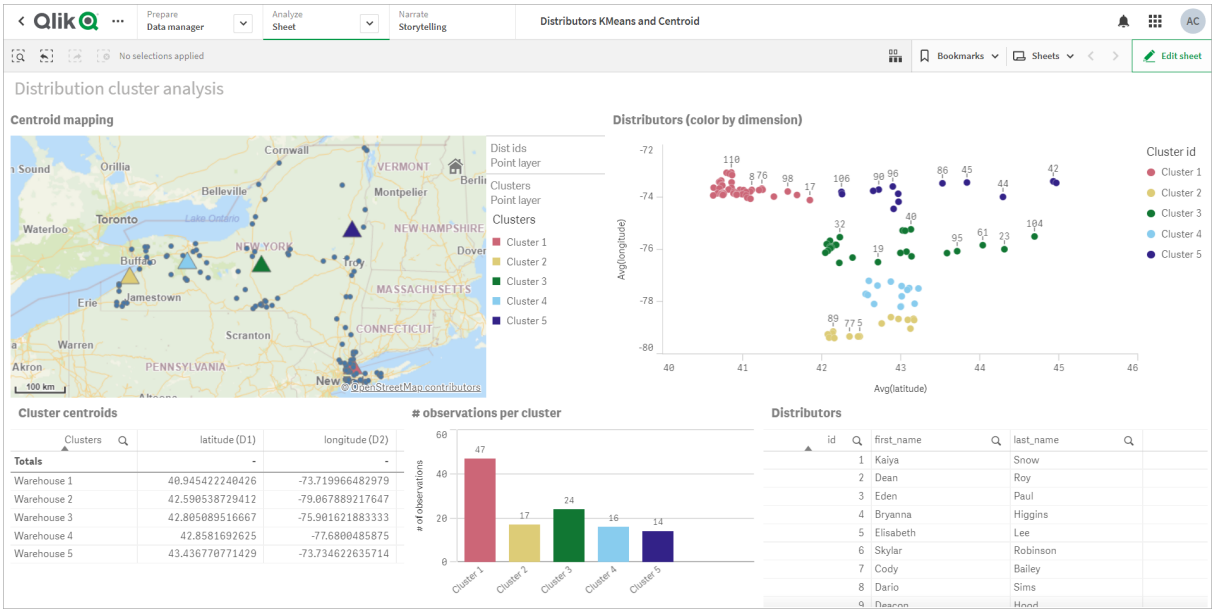
Map: Centroids mapped by cluster



Conclusion

Using the KMeans function for this real-world scenario, distributors have been segmented into similar groups or clusters based on similarity; in this case, proximity to one another. The Centroid function was applied to those clusters to identify five mapping coordinates. Those coordinates provide an initial central location at which to build or locate warehouses. The centroid function is applied to the **map** chart, so that app users can visualize where the centroids are located relative to surrounding cluster data points. The resulting coordinates represent potential warehouse locations that could minimize delivery costs to distributors in New York state.

App: KMeans and centroid analysis example



Distributor dataset: Inline load for data load editor in Qlik Sense

```
DistributorData:
Load * Inline [
id,first_name,last_name,telephone,address,city,state,zip,latitude,longitude
1,Kaiya,Snow,(716) 201-1212,6231 Tonawanda Creek Rd #APT 308,Lockport,NY,14094,43.08926,-
78.69313
2,Dean,Roy,(716) 201-1588,6884 E High St,Lockport,NY,14094,43.16245,-78.65036
3,Eden,Paul,(716) 202-4596,4647 Southwestern Blvd #APT 350,Hamburg,NY,14075,42.76003,-78.83194
4,Bryanna,Higgins,(716) 203-7041,418 Park Ave,Dunkirk,NY,14048,42.48279,-79.33088
5,Elisabeth,Lee,(716) 203-7043,36 E Courtney St,Dunkirk,NY,14048,42.48299,-79.31928
6,Skylar,Robinson,(716) 203-7166,26 Greco Ln,Dunkirk,NY,14048,42.4612095,-79.3317925
7,Cody,Bailey,(716) 203-7201,114 Lincoln Ave,Dunkirk,NY,14048,42.4801269,-79.322232
8,Dario,Sims,(408) 927-1606,N Castle Dr,Armonk,NY,10504,41.11979,-73.714864
9,Deacon,Hood,(410) 244-6221,4856 44th St,Woodside,NY,11377,40.748372,-73.905445
10,Zackery,Levy,(410) 363-8874,61 Executive Blvd,Farmingdale,NY,11735,40.7197457,-73.430239
11,Rey,Hawkins,(412) 344-8687,4585 Shimerville Rd,Clarence,NY,14031,42.972075,-78.6592452
12,Phillip,Howard,(413) 269-4049,464 Main St #101,Port Washington,NY,11050,40.8273756,-
73.7009971
13,Shirley,Tyler,(434) 985-8943,114 Glann Rd,Apalachin,NY,13732,42.0482515,-76.1229725
14,Aniyah,Jarvis,(440) 244-1808,87 N Middletown Rd,Pearl River,NY,10965,41.0629,-74.0159
15,Alayna,Woodard,(478) 335-3704,70 W Red Oak Ln,West Harrison,NY,10604,41.0162722,-73.7234926
16,Jermaine,Lambert,(508) 561-9836,24 Kellogg Rd,New Hartford,NY,13413,43.0555739,-75.2793197
17,Harper,Gibbs,(239) 466-0238,Po Box 33,Cottekill,NY,12419,41.853392,-74.106082
18,Osvaldo,Graham,(252) 246-0816,6878 Sand Hill Rd,East Syracuse,NY,13057,43.073215,-76.081448
19,Roberto,Wade,(270) 469-1211,3936 Holley Rd,Moravia,NY,13118,42.713044,-76.481227
20,Kate,Mcguire,(270) 788-3080,6451 State 64 Rte #3,Naples,NY,14512,42.707366,-77.380489
21,Dale,Andersen,(281) 480-5690,205 W Service Rd,Champlain,NY,12919,44.9645392,-73.4470831
22,Lorelai,Burch,(302) 644-2133,1 Brewster St,Glen Cove,NY,11542,40.865177,-73.633019
23,Amiyah,Flowers,(303) 223-0055,46600 Us Interstate 81 Rte,Alexandria
Bay,NY,13607,44.309626,-75.988365
```

```
24,Mckinley,Clements,(303) 918-3230,200 Summit Lake Dr,Valhalla,NY,10595,41.101145,-73.778298
25,Marc,Gibson,(607) 203-1233,25 Robinson St,Binghamton,NY,13901,42.107416,-75.901614
26,Kali,Norman,(607) 203-1400,1 Ely Park Blvd #APT 15,Binghamton,NY,13905,42.125866,-75.925026
27,Laci,Cain,(607) 203-1437,16 Zimmer Road,Kirkwood,NY,13795,42.066516,-75.792627
28,Mohammad,Perez,(607) 203-1652,71 Endicott Ave #APT 12,Johnson City,NY,13790,42.111894,-
75.952187
29,Izabelle,Pham,(607) 204-0392,434 State 369 Rte,Port Crane,NY,13833,42.185838,-75.823074
30,Kiley,Mays,(607) 204-0870,244 Ballyhack Rd #14,Port Crane,NY,13833,42.175612,-75.814917
31,Peter,Trevino,(607) 205-1374,125 Melbourne St.,Vestal,NY,13850,42.080254,-76.051124
32,Ani,Francis,(607) 208-4067,48 Caswell St,Afton,NY,13730,42.232065,-75.525674
33,Jared,Sheppard,(716) 386-3002,4709 430th Rte,Bemus Point,NY,14712,42.162175,-79.39176
34,Dulce,Atkinson,(914) 576-2266,501 Pelham Rd,New Rochelle,NY,10805,40.895449,-73.782602
35,Jayla,Beasley,(716) 526-1054,5010 474th Rte,Ashville,NY,14710,42.096859,-79.375561
36,Dane,Donovan,(718) 545-3732,5014 31st Ave,Woodside,NY,11377,40.756967,-73.909506
37,Brendon,Clay,(585) 322-7780,133 Cummings Ave,Gainesville,NY,14066,42.664309,-78.085651
38,Asia,Nunez,(718) 426-1472,2407 Gilmore ,East Elmhurst,NY,11369,40.766662,-73.869185
39,Dawson,Odonnell,(718) 342-2179,5019 H Ave,Brooklyn,NY,11234,40.633245,-73.927591
40,Kyle,Collins,(315) 733-7078,502 Rockhaven Rd,Utica,NY,13502,43.129184,-75.226726
41,Eliza,Hardin,(315) 331-8072,502 Sladen Place,West Point,NY,10996,41.3993,-73.973003
42,Kasen,Klein,(518) 298-4581,2407 Lake Shore Rd,Chazy,NY,12921,44.925561,-73.387373
43,Reuben,Bradford,(518) 298-4581,33 Lake Flats Dr,Champlain,NY,12919,44.928092,-73.387884
44,Henry,Grimes,(518) 523-3990,2407 Main St,Lake Placid,NY,12946,44.291487,-73.98474
45,Kyan,Livingston,(518) 585-7364,241 Alexandria Ave,Ticonderoga,NY,12883,43.836553,-73.43155
46,Kaitlyn,Short,(516) 678-3189,241 Chance Dr,Oceanside,NY,11572,40.638534,-73.63079
47,Damaris,Jacobs,(914) 664-5331,241 Claremont Ave,Mount Vernon,NY,10552,40.919852,-73.827848
48,Alivia,Schroeder,(315) 469-4473,241 Lafayette Rd,Syracuse,NY,13205,42.996446,-76.12957
49,Bridget,Strong,(315) 298-4355,241 Maltby Rd,Pulaski,NY,13142,43.584966,-76.136317
50,Francis,Lee,(585) 201-7021,166 Ross St,Batavia,NY,14020,43.0031502,-78.17487
51,Makaila,Phelps,(585) 201-7422,58 S Main St,Batavia,NY,14020,42.99941,-78.1939285
52,Jazlynn,Stephens,(585) 203-1087,1 Sinclair Dr,Pittsford,NY,14534,43.084157,-77.545452
53,Ryann,Randolph,(585) 203-1519,331 Eaglehead Rd,East Rochester,NY,14445,43.10785,-77.475552
54,Rosa,Baker,(585) 204-4011,42 Ossian St,Dansville,NY,14437,42.560761,-77.70088
55,Marcel,Barry,(585) 204-4013,42 Jefferson St,Dansville,NY,14437,42.557735,-77.702983
56,Dennis,Schmitt,(585) 204-4061,750 Dansville Mount Morris Rd,Dansville,NY,14437,42.584458,-
77.741648
57,Cassandra,Kim,(585) 204-4138,3 Perine Ave APT1,Dansville,NY,14437,42.562865,-77.69661
58,Kolton,Jacobson,(585) 206-5047,4925 Upper Holly Rd,Holley,NY,14470,43.175957,-78.074465
59,Nathanael,Donovan,(718) 393-3501,9604 57th Ave,Corona,NY,11373,40.736077,-73.864858
60,Robert,Frazier,(718) 271-3067,300 56th Ave,Corona,NY,11373,40.735304,-73.873997
61,Jessie,Mora,(315) 405-8991,9607 Forsyth Loop,Watertown,NY,13603,44.036466,-75.833437
62,Martha,Rollins,(347) 242-2642,22 Main St,Corona,NY,11373,40.757727,-73.829331
63,Emely,Townsend,(718) 699-0751,60 Sanford Ave,Corona,NY,11373,40.755466,-73.831029
64,Kylie,Cooley,(347) 561-7149,9608 95th Ave,Ozone Park,NY,11416,40.687564,-73.845715
65,Wendy,Cameron,(585) 571-4185,9608 Union St,Scottsville,NY,14546,43.013327,-77.7907839
66,Kayley,Peterson,(718) 654-5027,961 E 230th St,Bronx,NY,10466,40.889275,-73.850555
67,Camden,Ochoa,(718) 760-8699,59 Vark St,Yonkers,NY,10701,40.929322,-73.89957
68,Priscilla,Castillo,(910) 326-7233,9359 Elm St,Chadwicks,NY,13319,43.024902,-75.26886
69,Dana,Schultz,(913) 322-4580,99 Washington Ave,Hastings on Hudson,NY,10706,40.99265,-
73.879748
70,Blaze,Medina,(914) 207-0015,60 Elliott Ave,Yonkers,NY,10705,40.921498,-73.896682
71,Finnegan,Tucker,(914) 207-0015,90 Hillside Drive,Yonkers,NY,10705,40.922514,-73.892911
72,Pranav,Palmer,(914) 214-8376,5 Bruce Ave,Harrison,NY,10528,40.970916,-73.711493
73,Kolten,Wong,(914) 218-8268,70 Barker St,Mount Kisco,NY,10549,41.211993,-73.723202
74,Jasiah,Vazquez,(914) 231-5199,30 Broadway,Dobbs Ferry,NY,10522,41.004629,-73.879825
75,Lamar,Pierce,(914) 232-0380,68 Ridge Rd,Katonah,NY,10536,41.256662,-73.707964
76,Carla,Coffey,(914) 232-0469,197 Beaver Dam Rd,Katonah,NY,10536,41.247934,-73.664363
```

```
77,Brooklynn,Harmon,(716) 595-3227,8084 Glasgow Rd,Cassadega,NY,14718,42.353861,-79.329558
78,Raquel,Hodges,(585) 398-8125,809 County Road ,Victor,NY,14564,43.011745,-77.398806
79,Jerimiah,Gardner,(585) 787-9127,809 Houston Rd,Webster,NY,14580,43.224204,-77.491353
80,Clarence,Hammond,(720) 746-1619,809 Pierpont Ave,Piermont,NY,10968,41.0491181,-73.918622
81,Rhys,Gill,(518) 427-7887,81 Columbia St,Albany,NY,12210,42.652824,-73.752096
82,Edith,Parrish,(845) 452-7621,81 Glenwood Ave,Poughkeepsie,NY,12603,41.691058,-73.910829
83,Kobe,Mcintosh,(845) 371-1101,81 Heitman Dr,Spring Valley,NY,10977,41.103227,-74.054396
84,Ayden,Waters,(516) 796-2722,81 Kingfisher Rd,Levittown,NY,11756,40.738939,-73.52826
85,Francis,Rogers,(631) 427-7728,81 Knollwood Ave,Huntington,NY,11743,40.864905,-73.426107
86,Jaden,Landry,(716) 496-4038,12839 39th Rte,Chaffee,NY,14030,43.527396,-73.462786
87,Giancarlo,Campos,(518) 885-5717,1284 Saratoga Rd,Ballston Spa,NY,12020,42.968594,-73.862847
88,Eduardo,Contreras,(716) 285-8987,1285 Saunders Sett Rd,Niagara Falls,NY,14305,43.122963,-
79.029274
89,Gabriela,Davidson,(716) 267-3195,1286 Mee Rd,Falconer,NY,14733,42.147339,-79.137976
90,Evangeline,Case,(518) 272-9435,1287 2nd Ave,Watervliet,NY,12189,42.723132,-73.703818
91,Tyrone,Ellison,(518) 843-4691,1287 Midline Rd,Amsterdam,NY,12010,42.9730876,-74.1700608
92,Bryce,Bass,(518) 943-9549,1288 Leeds Athens Rd,Athens,NY,12015,42.259381,-73.876897
93,Londyn,Butler,(518) 922-7095,129 Argersinger Rd,Fultonville,NY,12072,42.910969,-74.441917
94,Graham,Becker,(607) 655-1318,129 Baker Rd,Windsor,NY,13865,42.107271,-75.66408
95,Rolando,Fitzgerald,(315) 465-4166,17164 County 90 Rte,Mannsville,NY,13661,43.713443,-
76.06232
96,Grant,Hoover,(518) 692-8363,1718 County 113 Rte,Schaghticote,NY,12154,42.900648,-73.585036
97,Mark,Goodwin,(631) 584-6761,172 Cambon Ave,Saint James,NY,11780,40.871152,-73.146032
98,Deacon,Cantu,(845) 221-7940,172 Carpenter Rd,Hopewell Junction,NY,12533,41.57388,-73.77609
99,Tristian,Walsh,(516) 997-4750,172 E Cabot Ln,Westbury,NY,11590,40.7480397,-73.54819
100,Abram,Alexander,(631) 588-3817,172 Lorenzo Cir,Ronkonkoma,NY,11779,40.837123,-73.09367
101,Lesly,Bush,(516) 489-3791,172 Nassau Blvd,Garden City,NY,11530,40.71147,-73.660753
102,Pamela,Espinoza,(716) 201-1520,172 Niagara St ,Lockport,NY,14094,43.169871,-78.70093
103,Bryanna,Newton,(914) 328-4332,172 Warren Ave,White Plains,NY,10603,41.047207,-73.79572
104,Marcelo,Schmitt,(315) 393-4432,319 Mansion Ave,Ogdensburg,NY,13669,44.690246,-75.49992
105,Layton,Valenzuela,(631) 676-2113,319 Singingwood Dr,Holbrook,NY,11741,40.801391,-73.058993
106,Roderick,Rocha,(518) 671-6037,319 Warren St,Hudson,NY,12534,42.252527,-73.790629
107,Camryn,Terrell,(315) 635-1680,3192 Olive Dr,Baldinsville,NY,13027,43.136843,-76.260303
108,Summer,Callahan,(585) 394-4195,3192 Smith Road,Canandaigua,NY,14424,42.875457,-77.228039
109,Pierre,Novak,(716) 665-2524,3194 Falconer Kimball Stand Rd,Falconer,NY,14733,42.138439,-
79.211091
110,Kennedi,Fry,(315) 543-2301,32 College Rd,Selden,NY,11784,40.861624,-73.04757
111,Wyatt,Pruitt,(716) 681-4042,277 Ransom Rd,Lancaster ,NY,14086,42.87702,-78.591302
112,Lilly,Jensen,(631) 841-0859,2772 Schliegel Blvd,Amityville,NY,11701,40.708021,-73.413015
113,Tristin,Hardin,(631) 920-0927,278 Fulton Street,West Babylon,NY,11704,40.733578,-73.357321
114,Tanya,Stafford,(716) 484-0771,278 Sampson St,Jamestown,NY,14701,42.0797,-79.247805
115,Paris,Cordova,(607) 589-4857,278 Washburn Rd,Spencer,NY,14883,42.225046,-76.510257
116,Alfonso,Morse,(718) 359-5582,200 Colden St,Flushing,NY,11355,40.750403,-73.822752
117,Maurice,Hooper,(315) 595-6694,4435 Italy Hill Rd,Branchport,NY,14418,42.597957,-77.199267
118,Iris,Wolf,(607) 539-7288,444 Harford Rd,Brooktondale,NY,14817,42.392164,-76.30756
];
```

KMeans2D - chart function

KMeans2D() evaluates the rows of the chart by applying k-means clustering, and for each chart row displays the cluster id of the cluster this data point has been assigned to. The columns that are used by the clustering algorithm are determined by the parameters coordinate_1, and coordinate_2, respectively. These are both aggregations. The number of clusters that are created is determined by the num_clusters parameter. Data can be optionally normalized by the norm parameter.

KMeans2D returns one value per data point. The returned value is a dual and is the integer value corresponding to the cluster each data point has been assigned to.

Syntax:

```
KMeans2D(num_clusters, coordinate_1, coordinate_2 [, norm])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
num_clusters	Integer that specifies the number of clusters.
coordinate_1	The aggregation that calculates the first coordinate, usually the x-axis of the scatter chart that can be made from the chart. The additional parameter, coordinate_2, calculates the second coordinate.
norm	The optional normalization method applied to datasets before KMeans clustering. Possible values: 0 or 'none' for no normalization 1 or 'zscore' for z-score normalization 2 or 'minmax' for min-max normalization If no parameter is supplied or if the supplied parameter is incorrect, no normalization is applied. Z-score normalizes data based on feature mean and standard deviation. Z-score does not ensure each feature has the same scale but it is a better approach than min-max when dealing with outliers. Min-max normalization ensures that the features have the same scale by taking the minimum and maximum values of each and recalculating each datapoint.

Example: Chart expression
In this example, we create a scatter plot chart using the *Iris* dataset, and then use KMeans to color the data by expression.

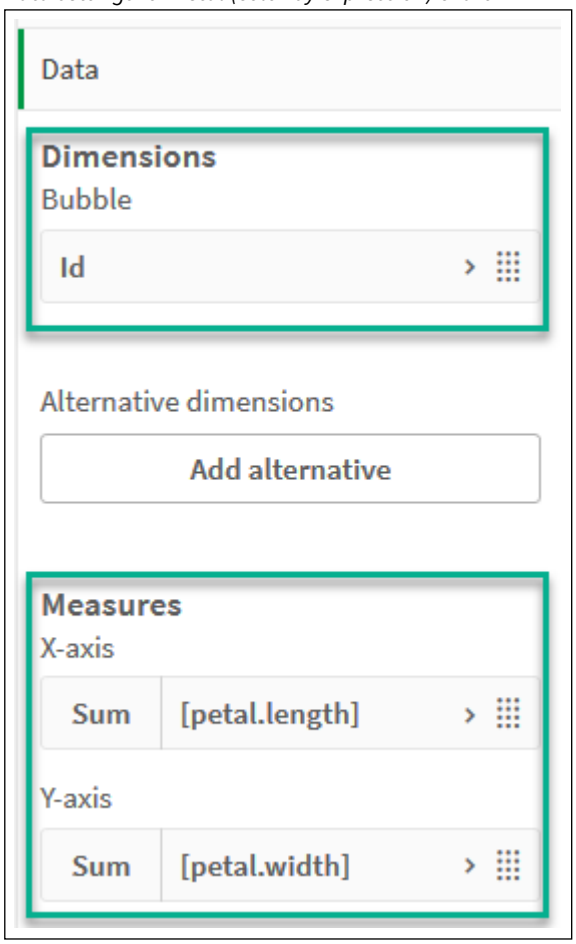
We also create a variable for the *num_clusters* argument, and then use a variable input box to change the number of clusters.

The *Iris* data set is publicly available in a variety of formats. We have provided the data as an inline table to load using the data load editor in Qlik Sense. Note that we added an *Id* column to the data table for this example.

After loading the data in Qlik Sense, we do the following:

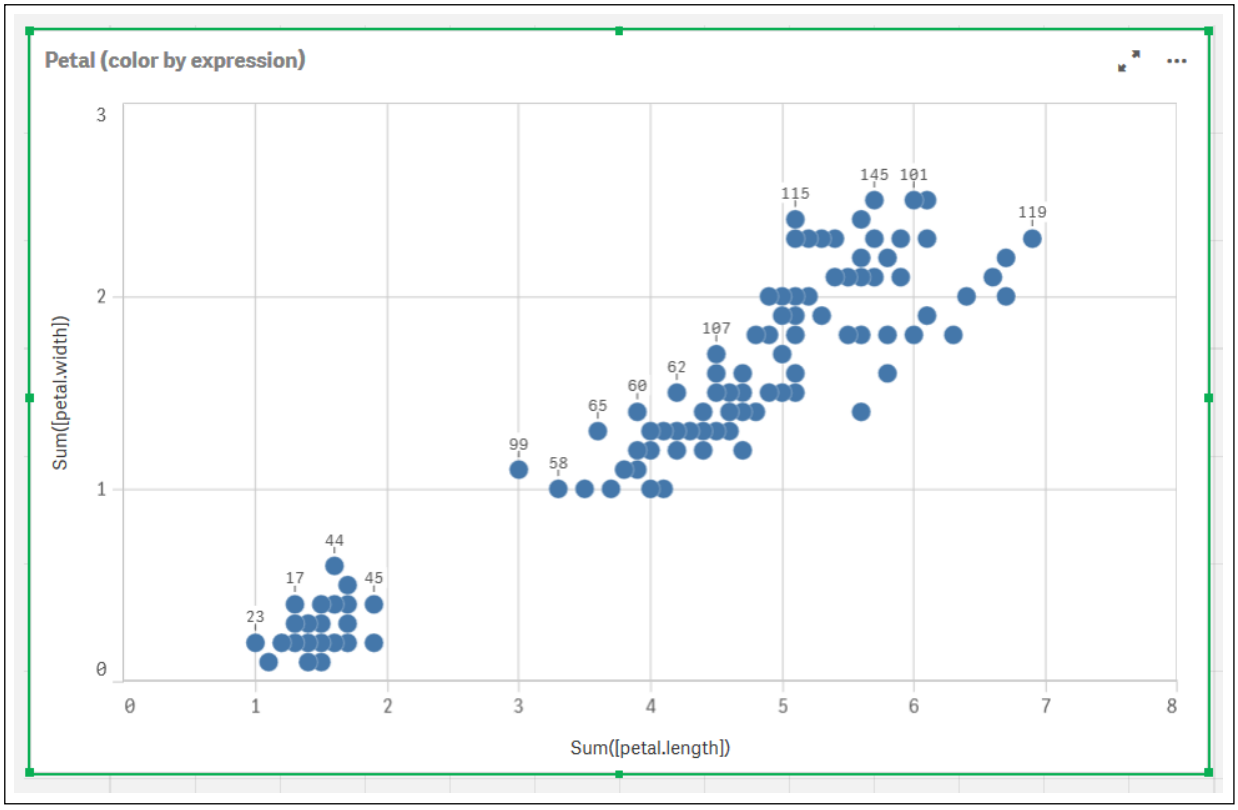
1. Drag a **Scatter plot** chart onto a new sheet. Name the chart *Petal (color by expression)*.
2. Create a variable to specify the number of clusters. For the variable **Name**, enter *KmeansPetalClusters*. For the variable **Definition**, enter *=2*.
3. Configure **Data** for the chart:
   i. Under **Dimensions**, choose *id* for the field for **Bubble**. Enter Cluster Id for the Label.
   ii. Under **Measures**, choose *Sum([petal.length])* for the expression for **X-axis**.
   iii. Under **Measures**, choose *Sum([petal.width])* for the expression for **Y-axis**.

   Data settings for Petal (color by expression) chart



The data points are plotted on the chart.

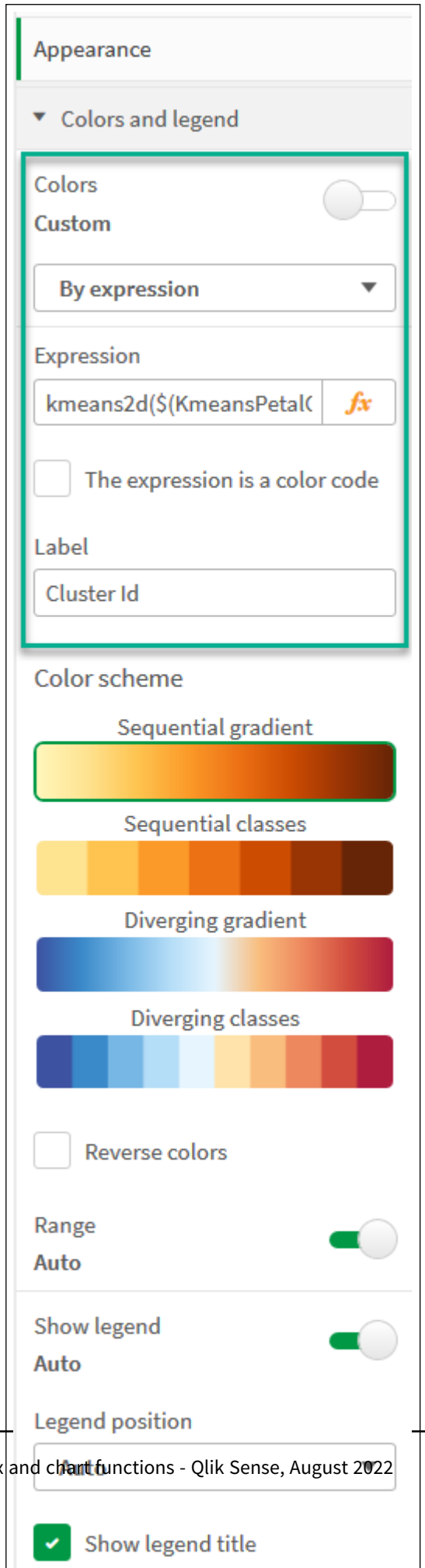
Data points on Petal (color by expression) chart



4. Configure **Appearance** for the chart:
   i. Under **Colors and legend**, choose **Custom** for **Colors**.
   ii. Choose to color the chart **By expression**.
   iii. Enter the following for **Expression**: *kmeans2d($(KmeansPetalClusters), Sum([petal.length]), Sum([petal.width]))*
   Note that *KmeansPetalClusters* is the variable that we set to *2*.
   Alternatively, enter the following: *kmeans2d(2, Sum([petal.length]), Sum([petal.width]))*
   iv. Deselect the check box for **The expression is a color code**.

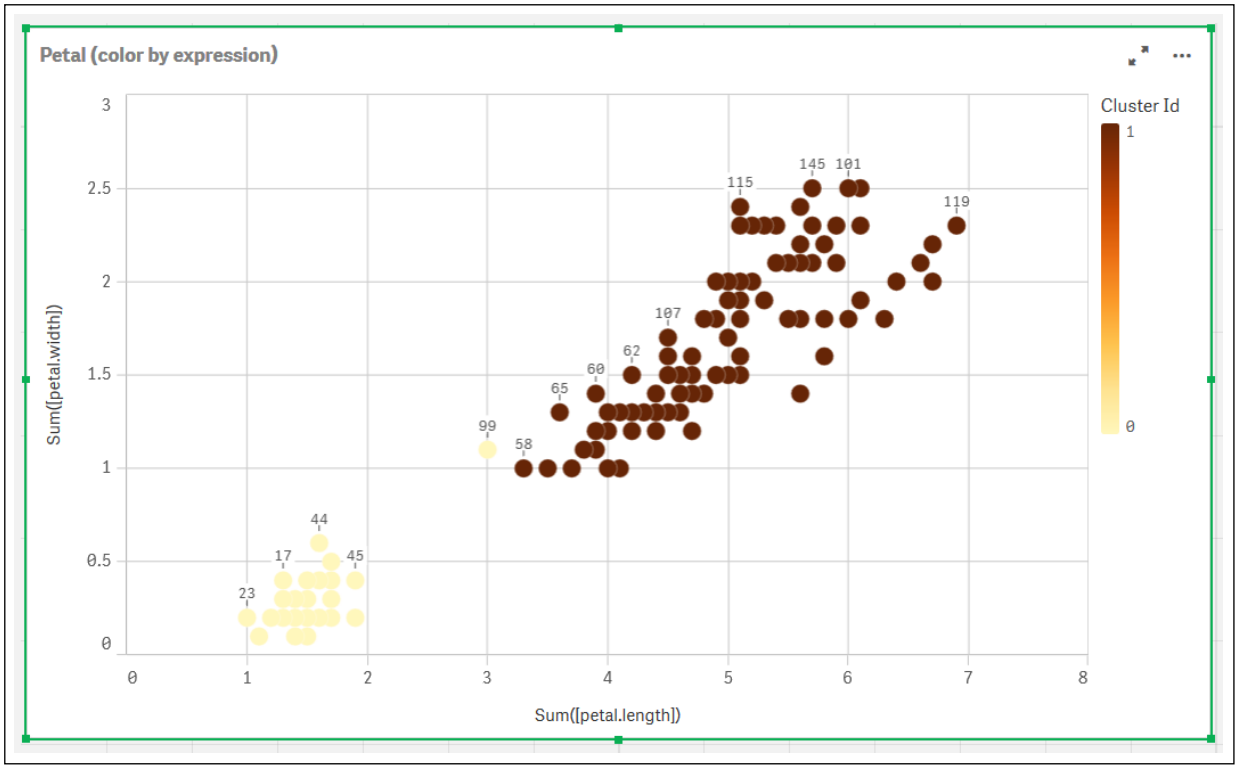
v. Enter the following for **Label**: *Cluster Id*

Appearance settings for Petal (color by expression) chart


Appearance
Colors and legend
Colors
Custom
By expression
Expression
kmeans2d($(KmeansPetalC
fx
The expression is a color code
Label
Cluster Id
Color scheme
Sequential gradient
Sequential classes
Diverging gradient
Diverging classes
Reverse colors
Range
Auto
Show legend
Auto
Legend position
Auto
Show legend title

The two clusters on the chart are colored by the KMeans expression.

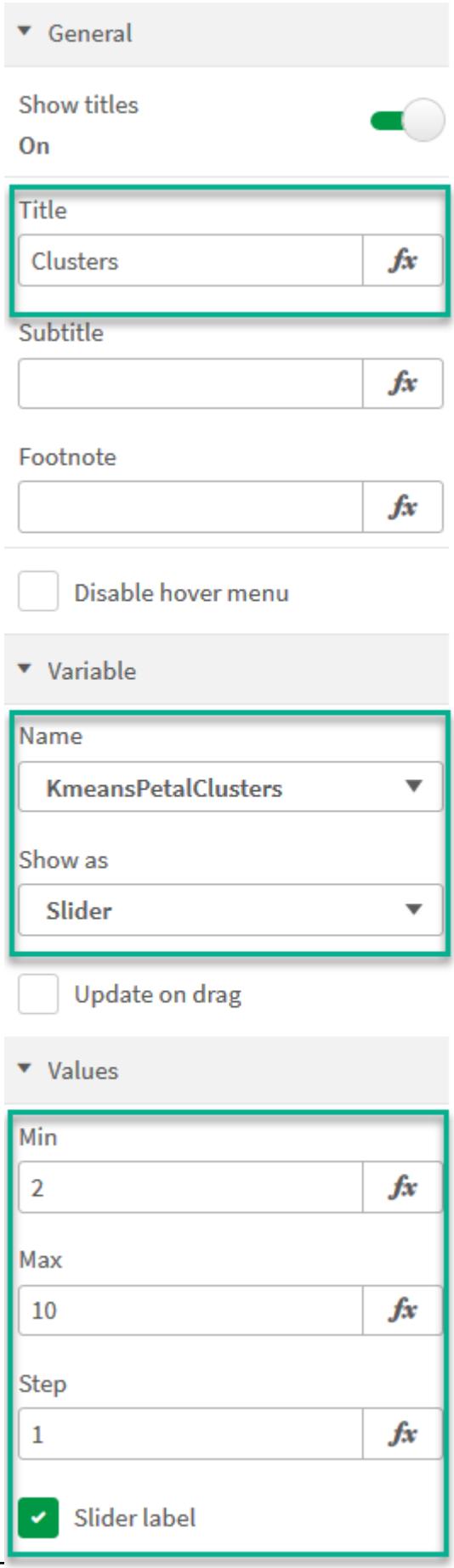
Clusters colored by expression on Petal (color by expression) chart



5. Add a **Variable input** box for the number of clusters.
   i. Under **Custom objects** in the **Assets** panel, choose **Qlik Dashboard bundle**. If we did not have access to the dashboard bundle, we could still change the number of clusters using the variable that we created, or directly as an integer in the expression.
   ii. Drag a **Variable input** box onto the sheet.
   iii. Under **Appearance**, click **General**.
   iv. Enter the following for **Title**: *Clusters*
   v. Click **Variable**.
   vi. Choose the following variable for **Name**: *KmeansPetalClusters*.
   vii. Choose **Slider** for **Show as**.

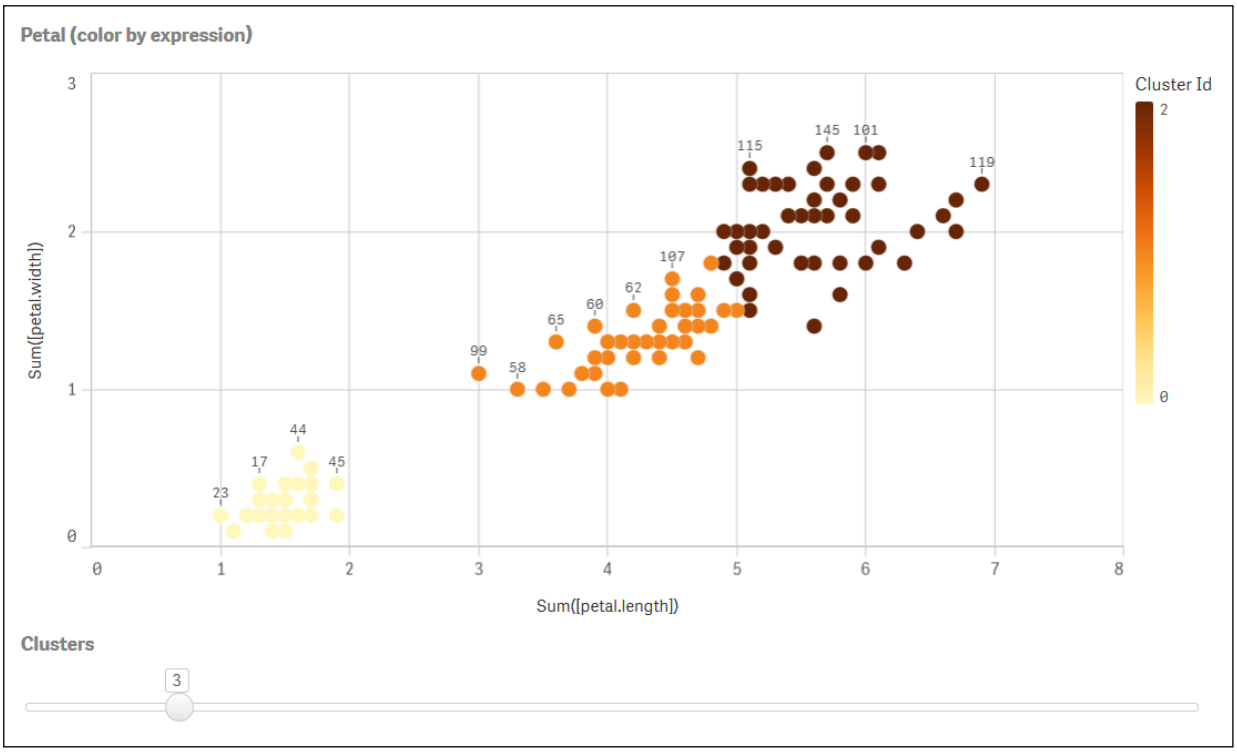
viii. Choose **Values**, and configure the settings as required,

Appearance for Clusters variable input box


General
Show titles
On
Title
Clusters
Subtitle
Footnote
Disable hover menu
Variable
Name
KmeansPetalClusters
Show as
Slider
Update on drag
Values
Min
2
Max
10
Step
1
Slider label

When we are done editing, we can change the number of clusters using the slider in the *Clusters* variable input box.

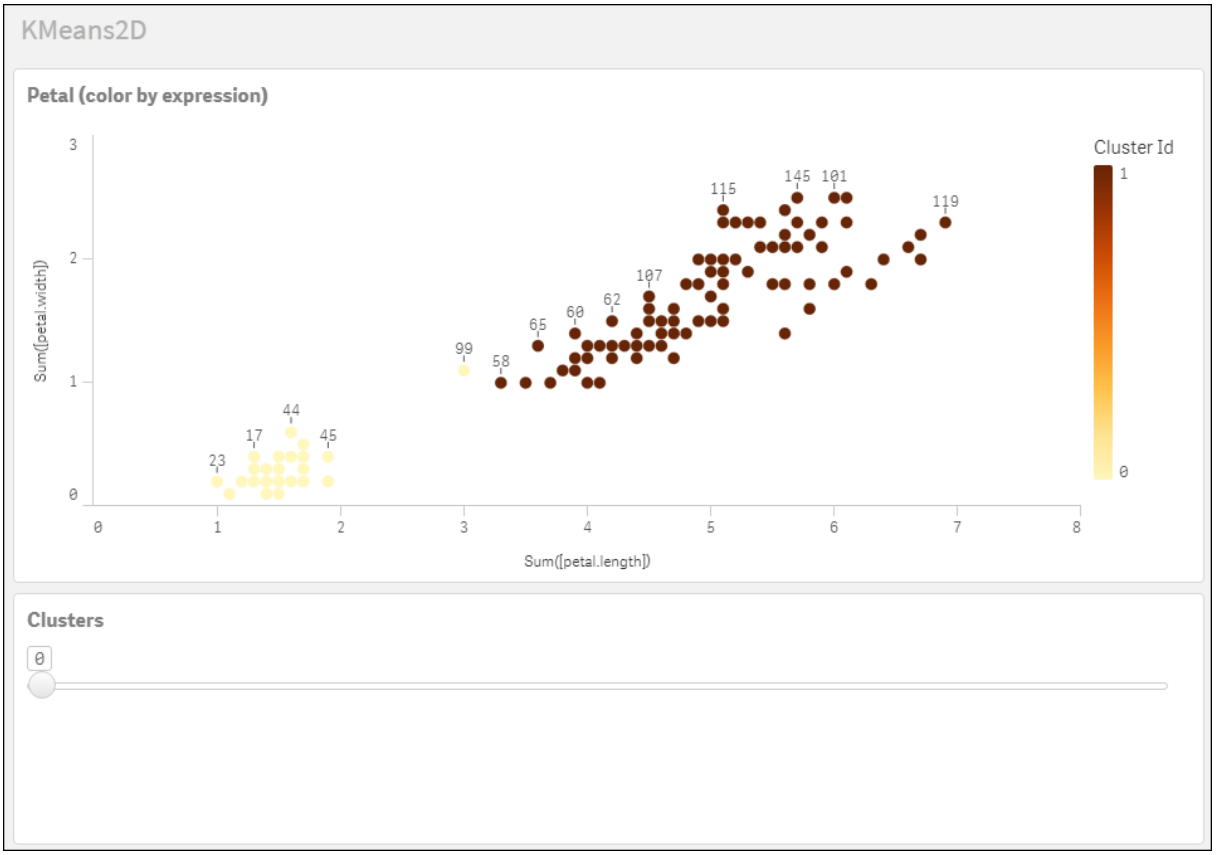
Clusters colored by expression on Petal (color by expression) chart



Auto-clustering

KMeans functions support auto-clustering using a method called depth difference (DeD). When a user sets 0 for the number of clusters, an optimal number of clusters for that dataset is determined. Note that while an integer for the number of clusters (*k*) is not explicitly returned, it is calculated within the KMeans algorithm. For example, if 0 is specified in the function for the value of *KmeansPetalClusters* or set through a variable input box, cluster assignments are automatically calculated for the dataset based on an optimal number of clusters.

KMeans depth difference method determines optimal number of clusters when (k) is set to 0



Iris data set: Inline load for data load editor in Qlik Sense

```
IrisData:
Load * Inline [
sepal.length, sepal.width, petal.length, petal.width, variety, id
5.1, 3.5, 1.4, 0.2, Setosa, 1
4.9, 3, 1.4, 0.2, Setosa, 2
4.7, 3.2, 1.3, 0.2, Setosa, 3
4.6, 3.1, 1.5, 0.2, Setosa, 4
5, 3.6, 1.4, 0.2, Setosa, 5
5.4, 3.9, 1.7, 0.4, Setosa, 6
4.6, 3.4, 1.4, 0.3, Setosa, 7
5, 3.4, 1.5, 0.2, Setosa, 8
4.4, 2.9, 1.4, 0.2, Setosa, 9
4.9, 3.1, 1.5, 0.1, Setosa, 10
5.4, 3.7, 1.5, 0.2, Setosa, 11
4.8, 3.4, 1.6, 0.2, Setosa, 12
4.8, 3, 1.4, 0.1, Setosa, 13
4.3, 3, 1.1, 0.1, Setosa, 14
5.8, 4, 1.2, 0.2, Setosa, 15
5.7, 4.4, 1.5, 0.4, Setosa, 16
5.4, 3.9, 1.3, 0.4, Setosa, 17
5.1, 3.5, 1.4, 0.3, Setosa, 18
5.7, 3.8, 1.7, 0.3, Setosa, 19
5.1, 3.8, 1.5, 0.3, Setosa, 20
5.4, 3.4, 1.7, 0.2, Setosa, 21
```

```
5.1, 3.7, 1.5, 0.4, Setosa, 22
4.6, 3.6, 1, 0.2, Setosa, 23
5.1, 3.3, 1.7, 0.5, Setosa, 24
4.8, 3.4, 1.9, 0.2, Setosa, 25
5, 3, 1.6, 0.2, Setosa, 26
5, 3.4, 1.6, 0.4, Setosa, 27
5.2, 3.5, 1.5, 0.2, Setosa, 28
5.2, 3.4, 1.4, 0.2, Setosa, 29
4.7, 3.2, 1.6, 0.2, Setosa, 30
4.8, 3.1, 1.6, 0.2, Setosa, 31
5.4, 3.4, 1.5, 0.4, Setosa, 32
5.2, 4.1, 1.5, 0.1, Setosa, 33
5.5, 4.2, 1.4, 0.2, Setosa, 34
4.9, 3.1, 1.5, 0.1, Setosa, 35
5, 3.2, 1.2, 0.2, Setosa, 36
5.5, 3.5, 1.3, 0.2, Setosa, 37
4.9, 3.1, 1.5, 0.1, Setosa, 38
4.4, 3, 1.3, 0.2, Setosa, 39
5.1, 3.4, 1.5, 0.2, Setosa, 40
5, 3.5, 1.3, 0.3, Setosa, 41
4.5, 2.3, 1.3, 0.3, Setosa, 42
4.4, 3.2, 1.3, 0.2, Setosa, 43
5, 3.5, 1.6, 0.6, Setosa, 44
5.1, 3.8, 1.9, 0.4, Setosa, 45
4.8, 3, 1.4, 0.3, Setosa, 46
5.1, 3.8, 1.6, 0.2, Setosa, 47
4.6, 3.2, 1.4, 0.2, Setosa, 48
5.3, 3.7, 1.5, 0.2, Setosa, 49
5, 3.3, 1.4, 0.2, Setosa, 50
7, 3.2, 4.7, 1.4, Versicolor, 51
6.4, 3.2, 4.5, 1.5, Versicolor, 52
6.9, 3.1, 4.9, 1.5, Versicolor, 53
5.5, 2.3, 4, 1.3, Versicolor, 54
6.5, 2.8, 4.6, 1.5, Versicolor, 55
5.7, 2.8, 4.5, 1.3, Versicolor, 56
6.3, 3.3, 4.7, 1.6, Versicolor, 57
4.9, 2.4, 3.3, 1, Versicolor, 58
6.6, 2.9, 4.6, 1.3, Versicolor, 59
5.2, 2.7, 3.9, 1.4, Versicolor, 60
5, 2, 3.5, 1, Versicolor, 61
5.9, 3, 4.2, 1.5, Versicolor, 62
6, 2.2, 4, 1, Versicolor, 63
6.1, 2.9, 4.7, 1.4, Versicolor, 64
5.6, 2.9, 3.6, 1.3, Versicolor, 65
6.7, 3.1, 4.4, 1.4, Versicolor, 66
5.6, 3, 4.5, 1.5, Versicolor, 67
5.8, 2.7, 4.1, 1, Versicolor, 68
6.2, 2.2, 4.5, 1.5, Versicolor, 69
5.6, 2.5, 3.9, 1.1, Versicolor, 70
5.9, 3.2, 4.8, 1.8, Versicolor, 71
6.1, 2.8, 4, 1.3, Versicolor, 72
6.3, 2.5, 4.9, 1.5, Versicolor, 73
6.1, 2.8, 4.7, 1.2, Versicolor, 74
6.4, 2.9, 4.3, 1.3, Versicolor, 75
6.6, 3, 4.4, 1.4, Versicolor, 76
```

```
6.8, 2.8, 4.8, 1.4, Versicolor, 77
6.7, 3, 5, 1.7, Versicolor, 78
6, 2.9, 4.5, 1.5, Versicolor, 79
5.7, 2.6, 3.5, 1, Versicolor, 80
5.5, 2.4, 3.8, 1.1, Versicolor, 81
5.5, 2.4, 3.7, 1, Versicolor, 82
5.8, 2.7, 3.9, 1.2, Versicolor, 83
6, 2.7, 5.1, 1.6, Versicolor, 84
5.4, 3, 4.5, 1.5, Versicolor, 85
6, 3.4, 4.5, 1.6, Versicolor, 86
6.7, 3.1, 4.7, 1.5, Versicolor, 87
6.3, 2.3, 4.4, 1.3, Versicolor, 88
5.6, 3, 4.1, 1.3, Versicolor, 89
5.5, 2.5, 4, 1.3, Versicolor, 90
5.5, 2.6, 4.4, 1.2, Versicolor, 91
6.1, 3, 4.6, 1.4, Versicolor, 92
5.8, 2.6, 4, 1.2, Versicolor, 93
5, 2.3, 3.3, 1, Versicolor, 94
5.6, 2.7, 4.2, 1.3, Versicolor, 95
5.7, 3, 4.2, 1.2, Versicolor, 96
5.7, 2.9, 4.2, 1.3, Versicolor, 97
6.2, 2.9, 4.3, 1.3, Versicolor, 98
5.1, 2.5, 3, 1.1, Versicolor, 99
5.7, 2.8, 4.1, 1.3, Versicolor, 100
6.3, 3.3, 6, 2.5, Virginica, 101
5.8, 2.7, 5.1, 1.9, Virginica, 102
7.1, 3, 5.9, 2.1, Virginica, 103
6.3, 2.9, 5.6, 1.8, Virginica, 104
6.5, 3, 5.8, 2.2, Virginica, 105
7.6, 3, 6.6, 2.1, Virginica, 106
4.9, 2.5, 4.5, 1.7, Virginica, 107
7.3, 2.9, 6.3, 1.8, Virginica, 108
6.7, 2.5, 5.8, 1.8, Virginica, 109
7.2, 3.6, 6.1, 2.5, Virginica, 110
6.5, 3.2, 5.1, 2, Virginica, 111
6.4, 2.7, 5.3, 1.9, Virginica, 112
6.8, 3, 5.5, 2.1, Virginica, 113
5.7, 2.5, 5, 2, Virginica, 114
5.8, 2.8, 5.1, 2.4, Virginica, 115
6.4, 3.2, 5.3, 2.3, Virginica, 116
6.5, 3, 5.5, 1.8, Virginica, 117
7.7, 3.8, 6.7, 2.2, Virginica, 118
7.7, 2.6, 6.9, 2.3, Virginica, 119
6, 2.2, 5, 1.5, Virginica, 120
6.9, 3.2, 5.7, 2.3, Virginica, 121
5.6, 2.8, 4.9, 2, Virginica, 122
7.7, 2.8, 6.7, 2, Virginica, 123
6.3, 2.7, 4.9, 1.8, Virginica, 124
6.7, 3.3, 5.7, 2.1, Virginica, 125
7.2, 3.2, 6, 1.8, Virginica, 126
6.2, 2.8, 4.8, 1.8, Virginica, 127
6.1, 3, 4.9, 1.8, Virginica, 128
6.4, 2.8, 5.6, 2.1, Virginica, 129
7.2, 3, 5.8, 1.6, Virginica, 130
7.4, 2.8, 6.1, 1.9, Virginica, 131
```

```
7.9, 3.8, 6.4, 2, Virginica, 132
6.4, 2.8, 5.6, 2.2, Virginica, 133
6.3, 2.8, 5.1, 1.5, Virginica, 134
6.1, 2.6, 5.6, 1.4, Virginica, 135
7.7, 3, 6.1, 2.3, Virginica, 136
6.3, 3.4, 5.6, 2.4, Virginica, 137
6.4, 3.1, 5.5, 1.8, Virginica, 138
6, 3, 4.8, 1.8, Virginica, 139
6.9, 3.1, 5.4, 2.1, Virginica, 140
6.7, 3.1, 5.6, 2.4, Virginica, 141
6.9, 3.1, 5.1, 2.3, Virginica, 142
5.8, 2.7, 5.1, 1.9, Virginica, 143
6.8, 3.2, 5.9, 2.3, Virginica, 144
6.7, 3.3, 5.7, 2.5, Virginica, 145
6.7, 3, 5.2, 2.3, Virginica, 146
6.3, 2.5, 5, 1.9, Virginica, 147
6.5, 3, 5.2, 2, Virginica, 148
6.2, 3.4, 5.4, 2.3, Virginica, 149
5.9, 3, 5.1, 1.8, Virginica, 150
];
```

KMeansND - chart function

KMeansND() evaluates the rows of the chart by applying k-means clustering, and for each chart row displays the cluster id of the cluster this data point has been assigned to. The columns that are used by the clustering algorithm are determined by the parameters coordinate_1, and coordinate_2, etc., up to n columns. These are all aggregations. The number of clusters that are created is determined by the num_clusters parameter.

KMeansND returns one value per data point. The returned value is a dual and is the integer value corresponding to the cluster each data point has been assigned to.

Syntax:

```
KMeansND(num_clusters, num_iter, coordinate_1, coordinate_2 [,coordinate_3 [,
...]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
num_clusters	Integer that specifies the number of clusters.
num_iter	The number of iterations of clustering with reinitialized cluster centers.
coordinate_1	The aggregation that calculates the first coordinate, usually the x-axis (of a scatter chart that can be made from the chart). The additional parameters calculate the second, third, and fourth coordinates, etc.

Example: Chart expression

In this example, we create a scatter plot chart using the *Iris* dataset, and then use KMeans to color the data by expression.

We also create a variable for the *num_clusters* argument, and then use a variable input box to change the number of clusters.

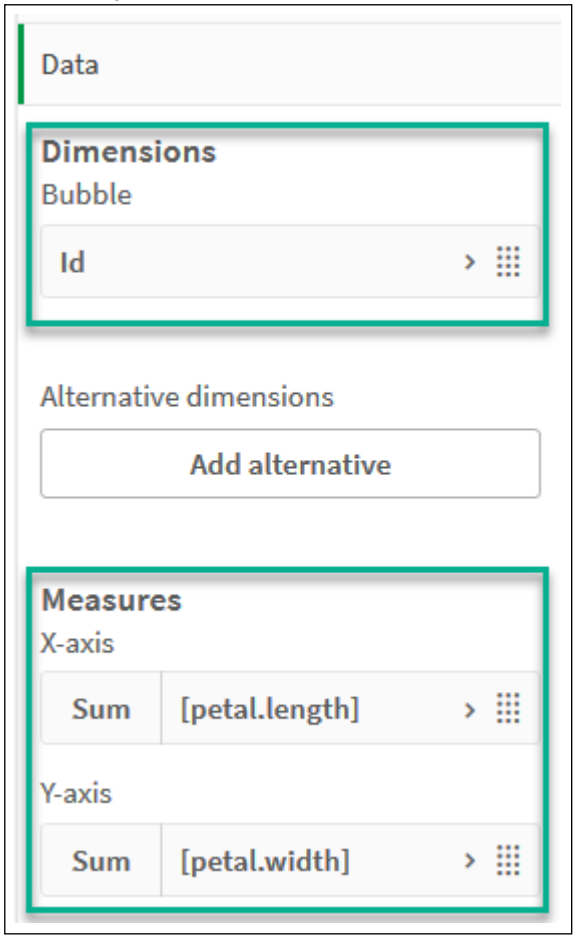
Additionally, we create a variable for the *num_iter* argument, and then use a second variable input box to change the number of iterations.

The *Iris* data set is publicly available in a variety of formats. We have provided the data as an inline table to load using the data load editor in Qlik Sense. Note that we added an *Id* column to the data table for this example.

After loading the data in Qlik Sense, we do the following:

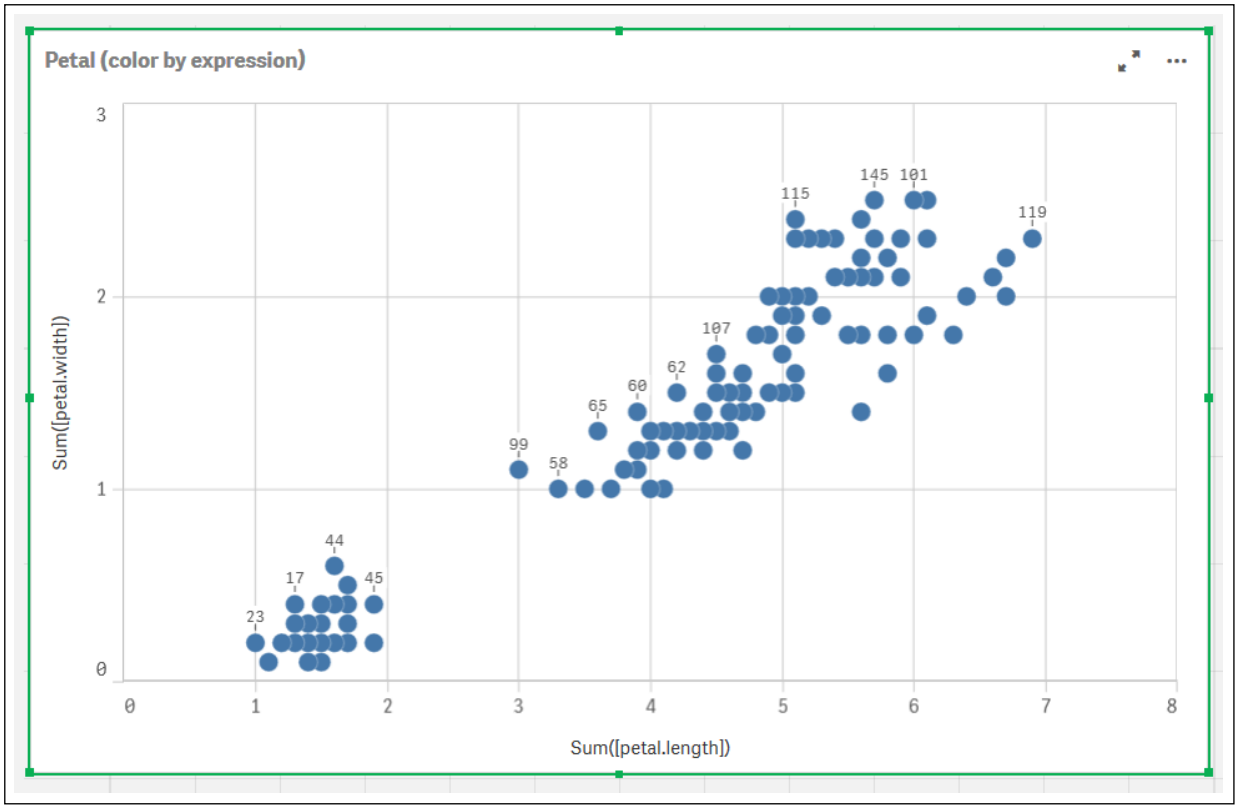
1. Drag a **Scatter plot** chart onto a new sheet. Name the chart *Petal (color by expression)*.
2. Create a variable to specify the number of clusters. For the variable **Name**, enter *KmeansPetalClusters*. For the variable **Definition**, enter *=2*.
3. Create a variable to specify the number of iterations. For the variable **Name**, enter *KmeansNumberIterations*. For the variable **Definition**, enter *=1*.
4. Configure **Data** for the chart:
    i. Under **Dimensions**, choose *id* for the field for **Bubble**. Enter Cluster Id for the Label.
    ii. Under **Measures**, choose *Sum([petal.length])* for the expression for **X-axis**.
    iii. Under **Measures**, choose *Sum([petal.width])* for the expression for **Y-axis**.

Data settings for Petal (color by expression) chart



The data points are plotted on the chart.

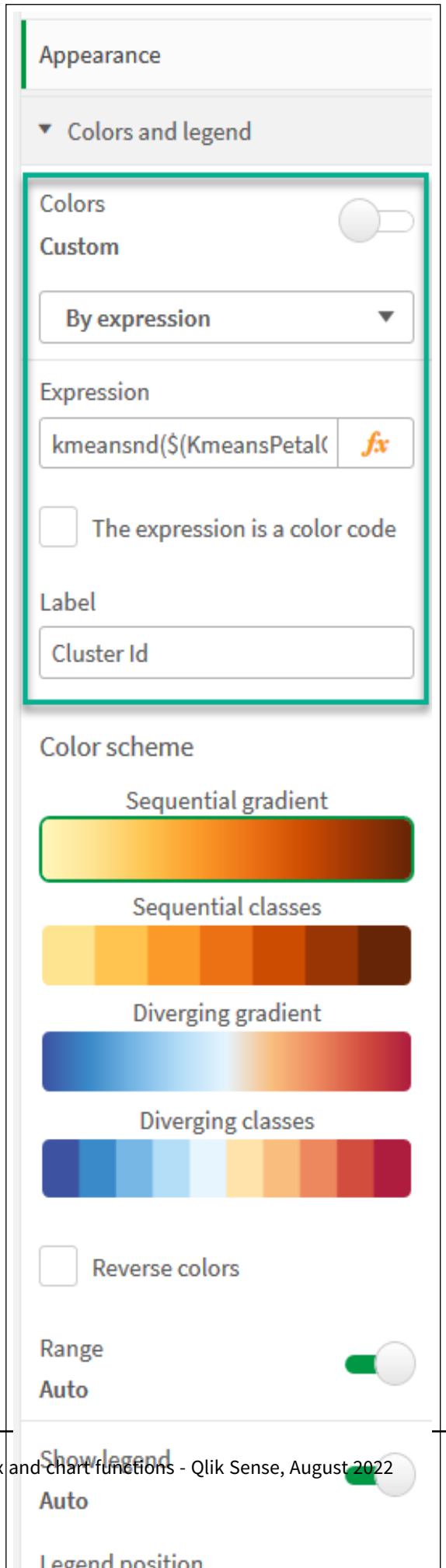
Data points on Petal (color by expression) chart



5. Configure **Appearance** for the chart:
   i. Under **Colors and legend**, choose **Custom** for **Colors**.
   ii. Choose to color the chart **By expression**.
   iii. Enter the following for **Expression**: *kmeansnd ($(KmeansPetalClusters),$(KmeansNumberIterations), Sum([petal.length]), Sum ([petal.width]),Sum([sepal.length]), Sum([sepal.width]))*
   Note that *KmeansPetalClusters* is the variable that we set to *2*. *KmeansNumberIterations* is the variable that we set to *1*.
   Alternatively, enter the following: *kmeansnd(2, 2, Sum([petal.length]), Sum([petal.width]),Sum ([sepal.length]), Sum([sepal.width]))*
   iv. Deselect the check box for **The expression is a color code**.

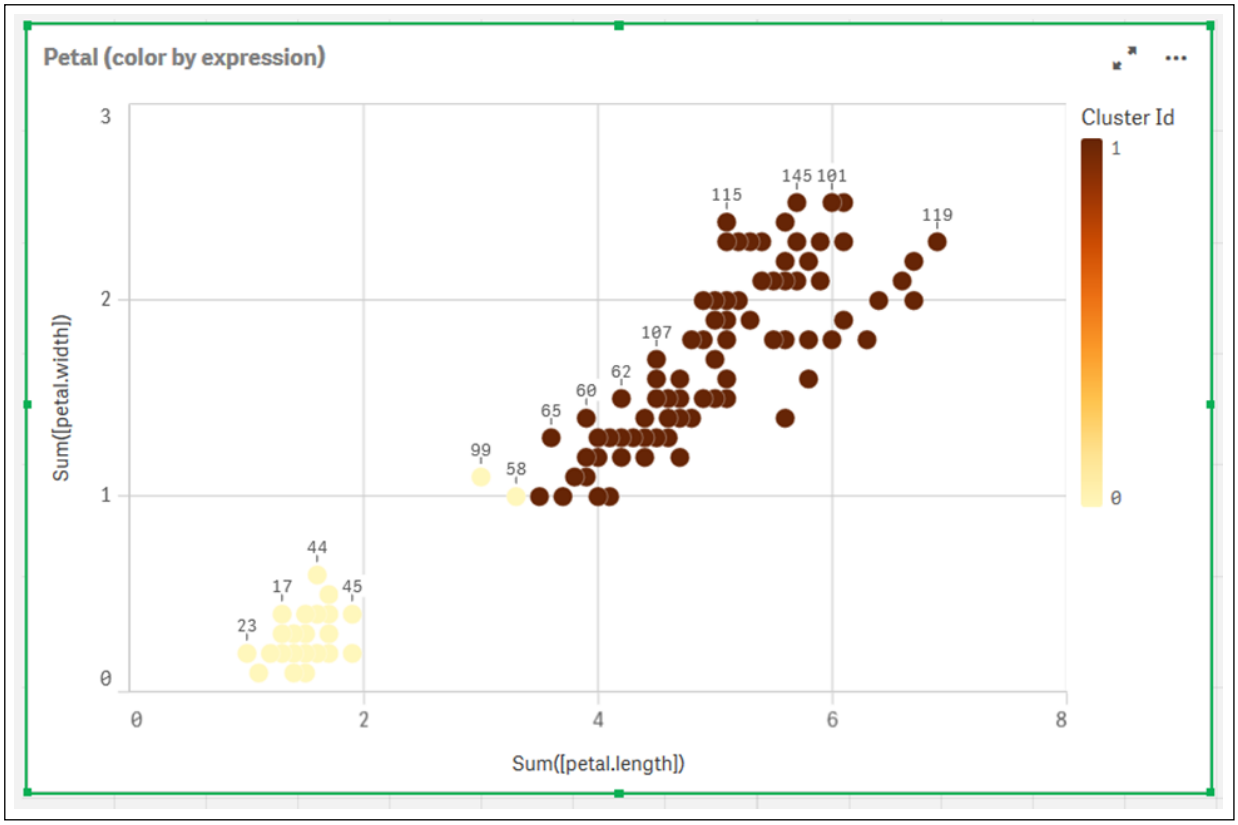
v. Enter the following for **Label**: *Cluster Id*

Appearance settings for Petal (color by expression) chart


Appearance
Colors and legend
Colors
Custom
By expression
Expression
kmeansnd($(KmeansPetal(
fx
The expression is a color code
Label
Cluster Id
Color scheme
Sequential gradient
Sequential classes
Diverging gradient
Diverging classes
Reverse colors
Range
Auto
Show legend
Auto
Legend position

The two clusters on the chart are colored by the KMeans expression.

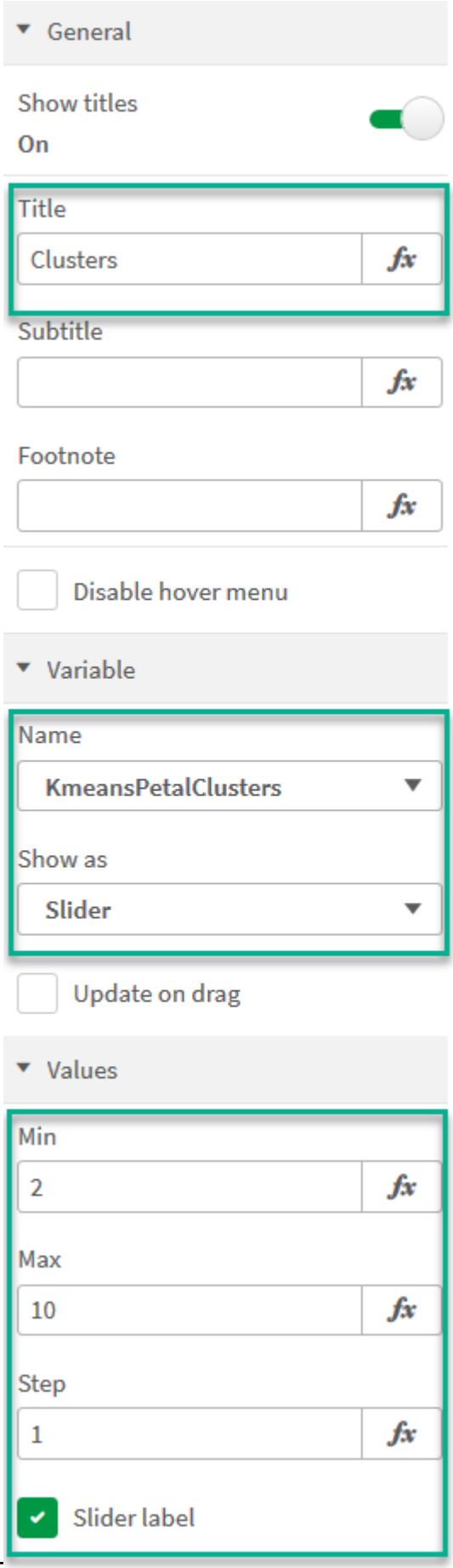
Clusters colored by expression on Petal (color by expression) chart



6. Add a **Variable input** box for the number of clusters.
   i. Under **Custom objects** in the **Assets** panel, choose **Qlik Dashboard bundle**. If we did not have access to the dashboard bundle, we could still change the number of clusters using the variable that we created, or directly as an integer in the expression.
   ii. Drag a **Variable input** box onto the sheet.
   iii. Under **Appearance**, click **General**.
   iv. Enter the following for **Title**: *Clusters*
   v. Click **Variable**.
   vi. Choose the following variable for **Name**: *KmeansPetalClusters*.
   vii. Choose **Slider** for **Show as**.

viii. Choose **Values**, and configure the settings as required,

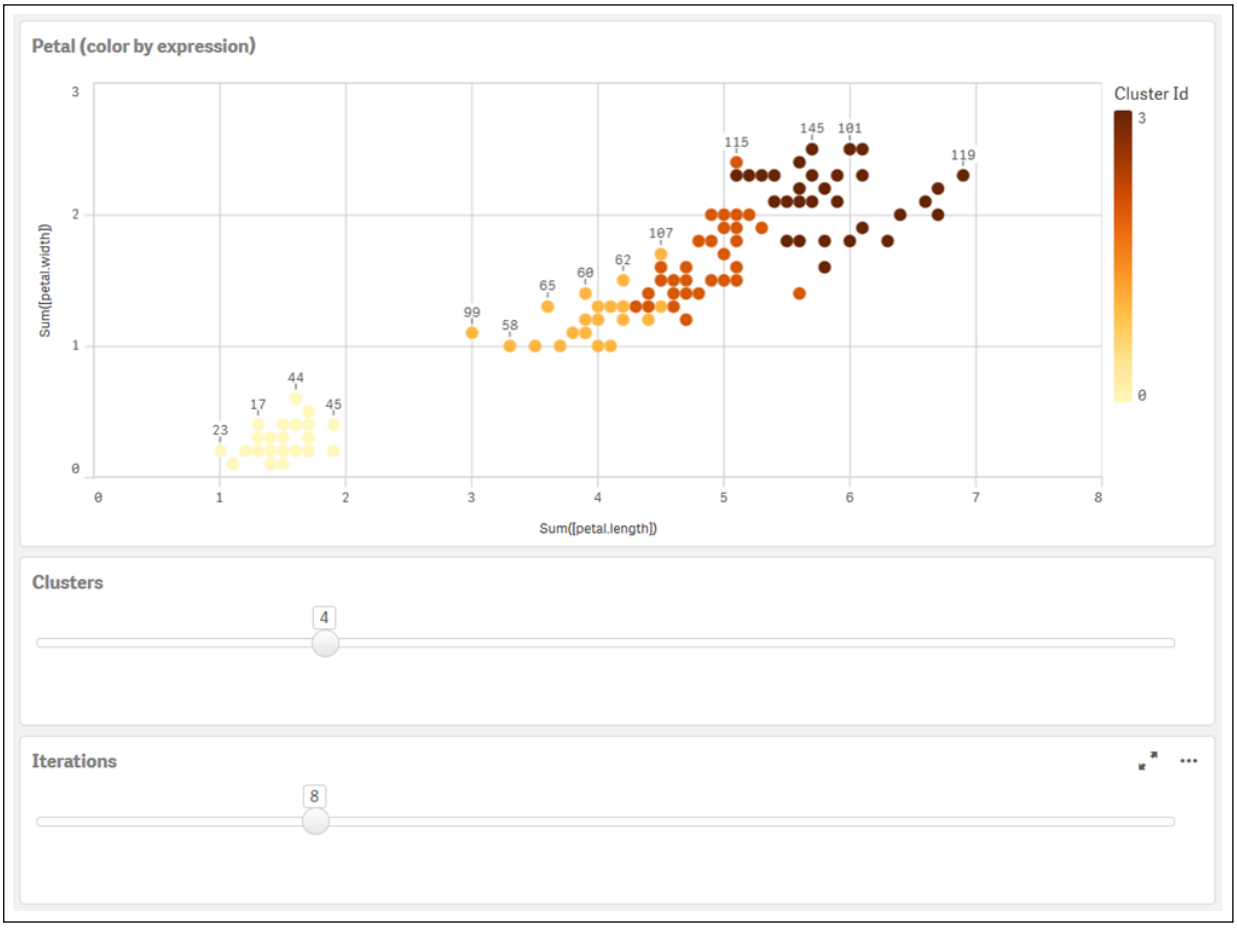
Appearance for Clusters variable input box


General
Show titles
On
Title
Clusters
Subtitle
Footnote
Disable hover menu
Variable
Name
KmeansPetalClusters
Show as
Slider
Update on drag
Values
Min
2
Max
10
Step
1
Slider label

7. Add a **Variable input** box for the number of iterations.
   i. Drag a **Variable input** box onto the sheet.
   ii. Under **Appearance**, choose **General**.
   iii. Enter the following for **Title**: *Iterations*
   iv. Under **Appearance**, choose **Variable**.
   v. Choose the following variable under **Name**: *KmeansNumberIterations*.
   vi. Configure the additional settings as required,

   We can now change the number of clusters and iterations using the sliders in the variable input boxes.

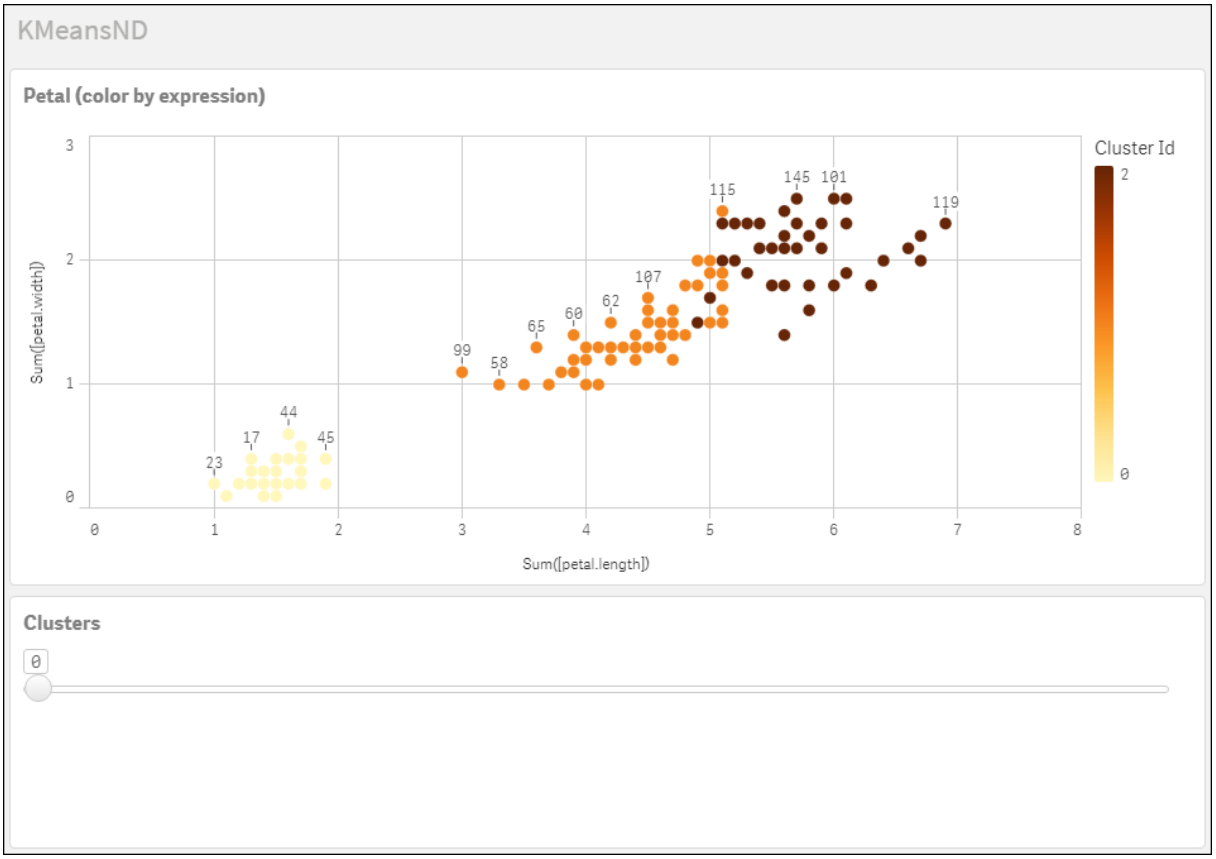
   Clusters colored by expression on Petal (color by expression) chart



Auto-clustering

KMeans functions support auto-clustering using a method called depth difference (DeD). When a user sets 0 for the number of clusters, an optimal number of clusters for that dataset is determined. Note that while an integer for the number of clusters (*k*) is not explicitly returned, it is calculated within the KMeans algorithm. For example, if 0 is specified in the function for the value of *KmeansPetalClusters* or set through a variable input box, cluster assignments are automatically calculated for the dataset based on an optimal number of clusters. Given the Iris dataset, if 0 is selected for the number of clusters, the algorithm will determine (auto-cluster) an optimal number of clusters (3) for this dataset.

KMeans depth difference method determines optimal number of clusters when (k) is set to 0.



Iris data set: Inline load for data load editor in Qlik Sense

```
IrisData:
Load * Inline [
sepal.length, sepal.width, petal.length, petal.width, variety, id
5.1, 3.5, 1.4, 0.2, Setosa, 1
4.9, 3, 1.4, 0.2, Setosa, 2
4.7, 3.2, 1.3, 0.2, Setosa, 3
4.6, 3.1, 1.5, 0.2, Setosa, 4
5, 3.6, 1.4, 0.2, Setosa, 5
5.4, 3.9, 1.7, 0.4, Setosa, 6
4.6, 3.4, 1.4, 0.3, Setosa, 7
5, 3.4, 1.5, 0.2, Setosa, 8
4.4, 2.9, 1.4, 0.2, Setosa, 9
4.9, 3.1, 1.5, 0.1, Setosa, 10
5.4, 3.7, 1.5, 0.2, Setosa, 11
4.8, 3.4, 1.6, 0.2, Setosa, 12
4.8, 3, 1.4, 0.1, Setosa, 13
4.3, 3, 1.1, 0.1, Setosa, 14
5.8, 4, 1.2, 0.2, Setosa, 15
5.7, 4.4, 1.5, 0.4, Setosa, 16
5.4, 3.9, 1.3, 0.4, Setosa, 17
5.1, 3.5, 1.4, 0.3, Setosa, 18
5.7, 3.8, 1.7, 0.3, Setosa, 19
5.1, 3.8, 1.5, 0.3, Setosa, 20
5.4, 3.4, 1.7, 0.2, Setosa, 21
```

```
5.1, 3.7, 1.5, 0.4, Setosa, 22
4.6, 3.6, 1, 0.2, Setosa, 23
5.1, 3.3, 1.7, 0.5, Setosa, 24
4.8, 3.4, 1.9, 0.2, Setosa, 25
5, 3, 1.6, 0.2, Setosa, 26
5, 3.4, 1.6, 0.4, Setosa, 27
5.2, 3.5, 1.5, 0.2, Setosa, 28
5.2, 3.4, 1.4, 0.2, Setosa, 29
4.7, 3.2, 1.6, 0.2, Setosa, 30
4.8, 3.1, 1.6, 0.2, Setosa, 31
5.4, 3.4, 1.5, 0.4, Setosa, 32
5.2, 4.1, 1.5, 0.1, Setosa, 33
5.5, 4.2, 1.4, 0.2, Setosa, 34
4.9, 3.1, 1.5, 0.1, Setosa, 35
5, 3.2, 1.2, 0.2, Setosa, 36
5.5, 3.5, 1.3, 0.2, Setosa, 37
4.9, 3.1, 1.5, 0.1, Setosa, 38
4.4, 3, 1.3, 0.2, Setosa, 39
5.1, 3.4, 1.5, 0.2, Setosa, 40
5, 3.5, 1.3, 0.3, Setosa, 41
4.5, 2.3, 1.3, 0.3, Setosa, 42
4.4, 3.2, 1.3, 0.2, Setosa, 43
5, 3.5, 1.6, 0.6, Setosa, 44
5.1, 3.8, 1.9, 0.4, Setosa, 45
4.8, 3, 1.4, 0.3, Setosa, 46
5.1, 3.8, 1.6, 0.2, Setosa, 47
4.6, 3.2, 1.4, 0.2, Setosa, 48
5.3, 3.7, 1.5, 0.2, Setosa, 49
5, 3.3, 1.4, 0.2, Setosa, 50
7, 3.2, 4.7, 1.4, Versicolor, 51
6.4, 3.2, 4.5, 1.5, Versicolor, 52
6.9, 3.1, 4.9, 1.5, Versicolor, 53
5.5, 2.3, 4, 1.3, Versicolor, 54
6.5, 2.8, 4.6, 1.5, Versicolor, 55
5.7, 2.8, 4.5, 1.3, Versicolor, 56
6.3, 3.3, 4.7, 1.6, Versicolor, 57
4.9, 2.4, 3.3, 1, Versicolor, 58
6.6, 2.9, 4.6, 1.3, Versicolor, 59
5.2, 2.7, 3.9, 1.4, Versicolor, 60
5, 2, 3.5, 1, Versicolor, 61
5.9, 3, 4.2, 1.5, Versicolor, 62
6, 2.2, 4, 1, Versicolor, 63
6.1, 2.9, 4.7, 1.4, Versicolor, 64
5.6, 2.9, 3.6, 1.3, Versicolor, 65
6.7, 3.1, 4.4, 1.4, Versicolor, 66
5.6, 3, 4.5, 1.5, Versicolor, 67
5.8, 2.7, 4.1, 1, Versicolor, 68
6.2, 2.2, 4.5, 1.5, Versicolor, 69
5.6, 2.5, 3.9, 1.1, Versicolor, 70
5.9, 3.2, 4.8, 1.8, Versicolor, 71
6.1, 2.8, 4, 1.3, Versicolor, 72
6.3, 2.5, 4.9, 1.5, Versicolor, 73
6.1, 2.8, 4.7, 1.2, Versicolor, 74
6.4, 2.9, 4.3, 1.3, Versicolor, 75
6.6, 3, 4.4, 1.4, Versicolor, 76
```

```
6.8, 2.8, 4.8, 1.4, Versicolor, 77
6.7, 3, 5, 1.7, Versicolor, 78
6, 2.9, 4.5, 1.5, Versicolor, 79
5.7, 2.6, 3.5, 1, Versicolor, 80
5.5, 2.4, 3.8, 1.1, Versicolor, 81
5.5, 2.4, 3.7, 1, Versicolor, 82
5.8, 2.7, 3.9, 1.2, Versicolor, 83
6, 2.7, 5.1, 1.6, Versicolor, 84
5.4, 3, 4.5, 1.5, Versicolor, 85
6, 3.4, 4.5, 1.6, Versicolor, 86
6.7, 3.1, 4.7, 1.5, Versicolor, 87
6.3, 2.3, 4.4, 1.3, Versicolor, 88
5.6, 3, 4.1, 1.3, Versicolor, 89
5.5, 2.5, 4, 1.3, Versicolor, 90
5.5, 2.6, 4.4, 1.2, Versicolor, 91
6.1, 3, 4.6, 1.4, Versicolor, 92
5.8, 2.6, 4, 1.2, Versicolor, 93
5, 2.3, 3.3, 1, Versicolor, 94
5.6, 2.7, 4.2, 1.3, Versicolor, 95
5.7, 3, 4.2, 1.2, Versicolor, 96
5.7, 2.9, 4.2, 1.3, Versicolor, 97
6.2, 2.9, 4.3, 1.3, Versicolor, 98
5.1, 2.5, 3, 1.1, Versicolor, 99
5.7, 2.8, 4.1, 1.3, Versicolor, 100
6.3, 3.3, 6, 2.5, Virginica, 101
5.8, 2.7, 5.1, 1.9, Virginica, 102
7.1, 3, 5.9, 2.1, Virginica, 103
6.3, 2.9, 5.6, 1.8, Virginica, 104
6.5, 3, 5.8, 2.2, Virginica, 105
7.6, 3, 6.6, 2.1, Virginica, 106
4.9, 2.5, 4.5, 1.7, Virginica, 107
7.3, 2.9, 6.3, 1.8, Virginica, 108
6.7, 2.5, 5.8, 1.8, Virginica, 109
7.2, 3.6, 6.1, 2.5, Virginica, 110
6.5, 3.2, 5.1, 2, Virginica, 111
6.4, 2.7, 5.3, 1.9, Virginica, 112
6.8, 3, 5.5, 2.1, Virginica, 113
5.7, 2.5, 5, 2, Virginica, 114
5.8, 2.8, 5.1, 2.4, Virginica, 115
6.4, 3.2, 5.3, 2.3, Virginica, 116
6.5, 3, 5.5, 1.8, Virginica, 117
7.7, 3.8, 6.7, 2.2, Virginica, 118
7.7, 2.6, 6.9, 2.3, Virginica, 119
6, 2.2, 5, 1.5, Virginica, 120
6.9, 3.2, 5.7, 2.3, Virginica, 121
5.6, 2.8, 4.9, 2, Virginica, 122
7.7, 2.8, 6.7, 2, Virginica, 123
6.3, 2.7, 4.9, 1.8, Virginica, 124
6.7, 3.3, 5.7, 2.1, Virginica, 125
7.2, 3.2, 6, 1.8, Virginica, 126
6.2, 2.8, 4.8, 1.8, Virginica, 127
6.1, 3, 4.9, 1.8, Virginica, 128
6.4, 2.8, 5.6, 2.1, Virginica, 129
7.2, 3, 5.8, 1.6, Virginica, 130
7.4, 2.8, 6.1, 1.9, Virginica, 131
```

```
7.9, 3.8, 6.4, 2, Virginica, 132
6.4, 2.8, 5.6, 2.2, Virginica, 133
6.3, 2.8, 5.1, 1.5, Virginica, 134
6.1, 2.6, 5.6, 1.4, Virginica, 135
7.7, 3, 6.1, 2.3, Virginica, 136
6.3, 3.4, 5.6, 2.4, Virginica, 137
6.4, 3.1, 5.5, 1.8, Virginica, 138
6, 3, 4.8, 1.8, Virginica, 139
6.9, 3.1, 5.4, 2.1, Virginica, 140
6.7, 3.1, 5.6, 2.4, Virginica, 141
6.9, 3.1, 5.1, 2.3, Virginica, 142
5.8, 2.7, 5.1, 1.9, Virginica, 143
6.8, 3.2, 5.9, 2.3, Virginica, 144
6.7, 3.3, 5.7, 2.5, Virginica, 145
6.7, 3, 5.2, 2.3, Virginica, 146
6.3, 2.5, 5, 1.9, Virginica, 147
6.5, 3, 5.2, 2, Virginica, 148
6.2, 3.4, 5.4, 2.3, Virginica, 149
5.9, 3, 5.1, 1.8, Virginica, 150
];
```

KMeansCentroid2D - chart function

KMeansCentroid2D() evaluates the rows of the chart by applying k-means clustering, and for each chart row displays the desired coordinate of the cluster this data point has been assigned to. The columns that are used by the clustering algorithm are determined by the parameters coordinate_1, and coordinate_2, respectively. These are both aggregations. The number of clusters that are created is determined by the num_clusters parameter. Data can be optionally normalized by the norm parameter.

KMeansCentroid2D returns one value per data point. The returned value is a dual and is one of the coordinates of the position corresponding to the cluster center the data point has been assigned to.

Syntax:

```
KMeansCentroid2D(num_clusters, coordinate_no, coordinate_1, coordinate_2 [, norm])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
num_clusters	Integer that specifies the number of clusters.
coordinate_no	The desired coordinate number of the centroids (corresponding, for example, to the x, y, or z axis).
coordinate_1	The aggregation that calculates the first coordinate, usually the x-axis of the scatter chart that can be made from the chart. The additional parameter, coordinate_2, calculates the second coordinate.

Argument	Description
norm	The optional normalization method applied to datasets before KMeans clustering. Possible values: 0 or 'none' for no normalization 1 or 'zscore' for z-score normalization 2 or 'minmax' for min-max normalization If no parameter is supplied or if the supplied parameter is incorrect, no normalization is applied. Z-score normalizes data based on feature mean and standard deviation. Z-score does not ensure each feature has the same scale but it is a better approach than min-max when dealing with outliers. Min-max normalization ensures that the features have the same scale by taking the minimum and maximum values of each and recalculating each datapoint.

Auto-clustering

KMeans functions support auto-clustering using a method called depth difference (DeD). When a user sets 0 for the number of clusters, an optimal number of clusters for that dataset is determined. Note that while an integer for the number of clusters (*k*) is not explicitly returned, it is calculated within the KMeans algorithm. For example, if 0 is specified in the function for the value of *KmeansPetalClusters* or set through a variable input box, cluster assignments are automatically calculated for the dataset based on an optimal number of clusters.

KMeansCentroidND - chart function

KMeansCentroidND() evaluates the rows of the chart by applying k-means clustering, and for each chart row displays the desired coordinate of the cluster this data point has been assigned to. The columns that are used by the clustering algorithm are determined by the parameters coordinate_1, coordinate_2, etc., up to n columns. These are all aggregations. The number of clusters that are created is determined by the num_clusters parameter.

KMeansCentroidND returns one value per row. The returned value is a dual and is one of the coordinates of the position corresponding to the cluster center the data point has been assigned to.

Syntax:

```
KMeansCentroidND((num_clusters, num_iter, coordinate_no, coordinate_1,
coordinate_2 [,coordinate_3 [, ...]])
```

Return data type: dual

Arguments:

Arguments

Argument	Description
num_clusters	Integer that specifies the number of clusters.
num_iter	The number of iterations of clustering with reinitialized cluster centers.
coordinate_no	The desired coordinate number of the centroids (corresponding, for example, to the x, y, or z axis).
coordinate_1	The aggregation that calculates the first coordinate, usually the x-axis (of a scatter chart that can be made from the chart). The additional parameters calculate the second, third, and fourth coordinates, etc.

Auto-clustering

KMeans functions support auto-clustering using a method called depth difference (DeD). When a user sets 0 for the number of clusters, an optimal number of clusters for that dataset is determined. Note that while an integer for the number of clusters (*k*) is not explicitly returned, it is calculated within the KMeans algorithm. For example, if 0 is specified in the function for the value of *KmeansPetalClusters* or set through a variable input box, cluster assignments are automatically calculated for the dataset based on an optimal number of clusters.

5.23 Statistical distribution functions

The statistical distribution DIST functions measure the probability of the distribution function at the point in the distribution given by the supplied value. The INV functions calculate the value, given the probability of the distribution. In contrast, the groups of statistical aggregation functions calculate the aggregated values of series of statistical test values for various statistical hypothesis tests.

The statistical distribution functions described below are all implemented in Qlik Sense using the Cephes function library. For references and details on algorithms used, accuracy, and so on, see: Cephes library. The Cephes function library is used by permission.

All functions can be used in both the data load script and in chart expressions.

Statistical distribution functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

CHIDIST

CHIDIST() returns the one-tailed probability of the chi^2 distribution. The chi^2 distribution is associated with a ch^{i2} test.

```
CHIDIST (value, degrees_freedom)
```

CHIINV

CHIINV() returns the inverse of the one-tailed probability of the chi^2 distribution.

```
CHIINV (prob, degrees_freedom)
```

NORMDIST

NORMDIST() returns the cumulative normal distribution for the specified mean and standard deviation. If mean = 0 and standard_dev = 1, the function returns the standard normal distribution.

```
NORMDIST (value, mean, standard_dev)
```

NORMINV

NORMINV() returns the inverse of the normal cumulative distribution for the specified mean and standard deviation.

```
NORMINV (prob, mean, standard_dev)
```

TDIST

TDIST() returns the probability for the Student's t-distribution where a numeric value is a calculated value of t for which the probability is to be computed.

```
TDIST (value, degrees_freedom, tails)
```

TINV

TINV() returns the t-value of the Student's t-distribution as a function of the probability and the degrees of freedom.

```
TINV (prob, degrees_freedom)
```

FDIST

FDIST() returns the F-probability distribution.

```
FDIST (value, degrees_freedom1, degrees_freedom2)
```

FINV

FINV() returns the inverse of the F-probability distribution.

```
FINV (prob, degrees_freedom1, degrees_freedom2)
```

See also:

Statistical aggregation functions (page 317)

CHIDIST

CHIDIST() returns the one-tailed probability of the chi^2 distribution. The chi^2 distribution is associated with a ch^{i2} test.

Syntax:

```
CHIDIST(value, degrees_freedom)
```

Return data type: number

Arguments:

Arguments

Argument	Description
value	The value at which you want to evaluate the distribution. The value must not be negative.
degrees_ freedom	A positive integer stating the number of degrees of freedom.

This function is related to the **CHIINV** function in the following way:
`If prob = CHIDIST(value,df), then CHIINV(prob, df) = value`

Limitations:

All arguments must be numeric, else NULL will be returned.

Examples and results:

Example	Result
CHIDIST(8, 15)	Returns 0.9238

CHIINV

CHIINV() returns the inverse of the one-tailed probability of the chi^2 distribution.

Syntax:

```
CHIINV(prob, degrees_freedom)
```

Return data type: number

Arguments:

Arguments

Argument	Description
prob	A probability associated with the chi^2 distribution. It must be a number between 0 and 1.
degrees_ freedom	An integer stating the number of degrees of freedom.

This function is related to the **CHIDIST** function in the following way:
`If prob = CHIDIST(value,df), then CHIINV(prob, df) = value`

Limitations:

All arguments must be numeric, else NULL will be returned.

Examples and results:

Example	Result
CHIINV(0.9237827, 15)	Returns 8.0000

FDIST

FDIST() returns the F-probability distribution.

Syntax:

```
FDIST(value, degrees_freedom1, degrees_freedom2)
```

Return data type: number

Arguments:

Arguments

Argument	Description
value	The value at which you want to evaluate the distribution. **Value** must not be negative.
degrees_freedom1	A positive integer stating the number of numerator degrees of freedom.
degrees_freedom2	A positive integer stating the number of denominator degrees of freedom.

This function is related to the **FINV** function in the following way:
`If prob = FDIST(value, df1, df2), then FINV(prob, df1, df2) = value`

Limitations:

All arguments must be numeric, else NULL will be returned.

Examples and results:

Example	Result
FDIST(15, 8, 6)	Returns 0.0019

FINV

FINV() returns the inverse of the F-probability distribution.

Syntax:

```
FINV(prob, degrees_freedom1, degrees_freedom2)
```

Return data type: number

Arguments:

Arguments

Argument	Description
prob	A probability associated with the F-probability distribution and must be a number between 0 and 1.
degrees_ freedom	An integer stating the number of degrees of freedom.

This function is related to the **FDIST** function in the following way:
`If prob = FDIST(value, df1, df2), then FINV(prob, df1, df2) = value`

Limitations:

All arguments must be numeric, else NULL will be returned.

Examples and results:

Example	Result
FINV(0.0019369, 8, 6)	Returns 15.0000

NORMDIST

NORMDIST() returns the cumulative normal distribution for the specified mean and standard deviation. If mean = 0 and standard_dev = 1, the function returns the standard normal distribution.

Syntax:

```
NORMDIST(value, [mean], [standard_dev], [cumulative])
```

Return data type: number

Arguments:

Arguments

Argument	Description
value	The value at which you want to evaluate the distribution.
mean	Optional value stating the arithmetic mean for the distribution. If you do not state this argument, the default value is 0.
standard_dev	Optional positive value stating the standard deviation of the distribution. If you do not state this argument, the default value is 1.
cumulative	You can optionally select to use a standard normal distribution or a cumulative distribution. 0 = standard normal distribution 1 = cumulative distribution (default)

This function is related to the **NORMINV** function in the following way:
`If prob = NORMDIST(value, m, sd), then NORMINV(prob, m, sd) = value`

Limitations:

All arguments must be numeric, else NULL will be returned.

Examples and results:

Example	Result
NORMDIST(0.5, 0, 1)	Returns 0.6915

NORMINV

NORMINV() returns the inverse of the normal cumulative distribution for the specified mean and standard deviation.

Syntax:

```
NORMINV(prob, mean, standard_dev)
```

Return data type: number

Arguments:

Arguments

Argument	Description
prob	A probability associated with the normal distribution. It must be a number between 0 and 1.
mean	A value stating the arithmetic mean for the distribution.
standard_dev	A positive value stating the standard deviation of the distribution.

This function is related to the **NORMDIST** function in the following way:
`If prob = NORMDIST(value, m, sd), then NORMINV(prob, m, sd) = value`

Limitations:

All arguments must be numeric, else NULL will be returned.

Examples and results:

Example	Result
NORMINV(0.6914625, 0, 1)	Returns 0.5000

TDIST

TDIST() returns the probability for the Student's t-distribution where a numeric value is a calculated value of t for which the probability is to be computed.

Syntax:

```
TDIST(value, degrees_freedom, tails)
```

Return data type: number

Arguments:

Arguments

Argument	Description
value	The value at which you want to evaluate the distribution and must not be negative.
degrees_freedom	A positive integer stating the number of degrees of freedom.
tails	Must be either 1 (one-tailed distribution) or 2 (two-tailed distribution).

This function is related to the **TINV** function in the following way:
`If prob = TDIST(value, df ,2), then TINV(prob, df) = value`

Limitations:

All arguments must be numeric, else NULL will be returned.

Examples and results:

Example	Result
TDIST(1, 30, 2)	Returns 0.3253

TINV

TINV() returns the t-value of the Student's t-distribution as a function of the probability and the degrees of freedom.

Syntax:

```
TINV(prob, degrees_freedom)
```

Return data type: number

Arguments:

Arguments

Argument	Description
prob	A two-tailed probability associated with the t-distribution. It must be a number between 0 and 1.
degrees_ freedom	An integer stating the number of degrees of freedom.

Limitations:

All arguments must be numeric, else NULL will be returned.

This function is related to the **TDIST** function in the following way:
`If prob = TDIST(value, df ,2), then TINV(prob, df) = value.`

Examples and results:

Example	Result
TINV(0.3253086, 30)	Returns 1.0000

5.24 String functions

This section describes functions for handling and manipulating strings.

All functions can be used in both the data load script and in chart expressions, except for **Evaluate** which can only be used in the data load script.

String functions overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Capitalize

Capitalize() returns the string with all words in initial uppercase letters.

```
Capitalize (text)
```

Chr

Chr() returns the Unicode character corresponding to the input integer.

```
Chr (int)
```

Evaluate

Evaluate() finds if the input text string can be evaluated as a valid Qlik Sense expression, and if so, returns the value of the expression as a string. If the input string is not a valid expression, NULL is returned.

```
Evaluate (expression_text)
```

FindOneOf

FindOneOf() searches a string to find the position of the occurrence of any character from a set of provided characters. The position of the first occurrence of any character from the search set is returned unless a third argument (with a value greater than 1) is supplied. If no match is found, **0** is returned.

```
FindOneOf (text, char_set[, count])
```

Hash128

Hash128() returns a 128-bit hash of the combined input expression values. The result is a 22-character string.

```
Hash128 (expr{, expression})
```

Hash160

Hash160() returns a 160-bit hash of the combined input expression values. The result is a 27-character string.

```
Hash160 (expr{, expression})
```

Hash256

Hash256() returns a 256-bit hash of the combined input expression values. The result is a 43-character string.

```
Hash256 (expr{, expression})
```

Index

Index() searches a string to find the starting position of the nth occurrence of a provided substring. An optional third argument provides the value of n, which is 1 if omitted. A negative value searches from the end of the string. The positions in the string are numbered from **1** and up.

```
Index  (text, substring[, count])
```

IsJson

IsJson() tests whether a specified string contains valid JSON (JavaScript Object Notation) data. You can also validate a specific JSON data type.

```
IsJson (json [, type])
```

JsonGet

JsonGet() returns the path of a JSON (JavaScript Object Notation) data string. The data must be valid JSON but can contain extra spaces or newlines.

```
JsonGet (json, path)
```

JsonSet

JsonSet() modifies a string containing JSON (JavaScript Object Notation) data. It can set or insert a JSON value with the new location specified by the path. The data must be valid JSON but can contain extra spaces or newlines.

```
JsonSet(json, path, value)
```

KeepChar

KeepChar() returns a string consisting of the first string ,'text', less any of the characters NOT contained in the second string, "keep_chars".

```
KeepChar (text, keep_chars)
```

Left

Left() returns a string consisting of the first (leftmost) characters of the input string, where the number of characters is determined by the second argument.

```
Left (text, count)
```

Len

Len() returns the length of the input string.

```
Len (text)
```

LevenshteinDist

LevenshteinDist() returns the Levenshtein distance between two strings. It is defined as the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into the other. The function is useful for fuzzy string comparisons.

```
LevenshteinDist (text1, text2)
```

Lower

Lower() converts all the characters in the input string to lower case.

```
Lower (text)
```

LTrim

LTrim() returns the input string trimmed of any leading spaces.

```
LTrim (text)
```

Mid

Mid() returns the part of the input string starting at the position of the character defined by the second argument, 'start', and returning the number of characters defined by the third argument, 'count'. If 'count' is omitted, the rest of the input string is returned. The first character in the input string is numbered 1.

```
Mid (text, start[, count])
```

Ord

Ord() returns the Unicode code point number of the first character of the input string.

```
Ord (text)
```

PurgeChar

PurgeChar() returns a string consisting of the characters contained in the input string ('text'), excluding any that appear in the second argument ('remove_chars').

```
PurgeChar (text, remove_chars)
```

Repeat

Repeat() forms a string consisting of the input string repeated the number of times defined by the second argument.

```
Repeat (text[, repeat_count])
```

Replace

Replace() returns a string after replacing all occurrences of a given substring within the input string with another substring. The function is non-recursive and works from left to right.

```
Replace (text, from_str, to_str)
```

Right

Right() returns a string consisting of the last (rightmost) characters of the input string, where the number of characters is determined by the second argument.

```
Right (text, count)
```

RTrim

RTrim() returns the input string trimmed of any trailing spaces.

```
RTrim (text)
```

SubField

SubField() is used to extract substring components from a parent string field, where the original record fields consist of two or more parts separated by a delimiter.

```
SubField (text, delimiter[, field_no ])
```

SubStringCount

SubStringCount() returns the number of occurrences of the specified substring in the input string text. If there is no match, 0 is returned.

```
SubStringCount (text, substring)
```

TextBetween

TextBetween() returns the text in the input string that occurs between the characters specified as delimiters.

```
TextBetween (text, delimiter1, delimiter2[, n])
```

Trim

Trim() returns the input string trimmed of any leading and trailing spaces.

```
Trim (text)
```

Upper

Upper() converts all the characters in the input string to upper case for all text characters in the expression. Numbers and symbols are ignored.

```
Upper (text)
```

Capitalize

Capitalize() returns the string with all words in initial uppercase letters.

Syntax:

```
Capitalize(text)
```

Return data type: string

Example: Chart expressions

Example	Result
`Capitalize ( 'star trek' )`	Returns 'Star Trek'
`Capitalize ( 'AA bb cC Dd')`	Returns 'Aa Bb Cc Dd'

Example: Load script

```
Load
String,
Capitalize(String)
Inline
[String
rHode iSland
washingTon d.C.
new york];
```

Result

String	Capitalize(String)
rHode iSland	Rhode Island
washingTon d.C.	Washington D.C.
new york	New York

Chr

Chr() returns the Unicode character corresponding to the input integer.

Syntax:

```
Chr(int)
```

Return data type: string

Examples and results:

Example	Result
Chr(65)	Returns the string 'A'
Chr(163)	Returns the string '£'
Chr(35)	Returns the string '#'

Evaluate

Evaluate() finds if the input text string can be evaluated as a valid Qlik Sense expression, and if so, returns the value of the expression as a string. If the input string is not a valid expression, NULL is returned.

Syntax:

```
Evaluate(expression_text)
```

Return data type: dual

This string function cannot be used in chart expressions.

Examples and results:

Function example	Result
`Evaluate ( 5 * 8 )`	Returns '40'

Load script example

```
Load
Evaluate(String) as Evaluated,
String
Inline
[String
4
5+3
0123456789012345678
Today()
];
```

Result

String	Evaluated
4	4
5+3	8
0123456789012345678	0123456789012345678
Today()	2022-02-02

FindOneOf

FindOneOf() searches a string to find the position of the occurrence of any character from a set of provided characters. The position of the first occurrence of any character from the search set is returned unless a third argument (with a value greater than 1) is supplied. If no match is found, **0** is returned.

Syntax:

```
FindOneOf(text, char_set[, count])
```

Return data type: integer

Arguments:

Arguments

Argument	Description
text	The original string.
char_set	A set of characters to search for in text.
count	Defines which occurrence of any of the character to search for. For example, a value of 2 searches for the second occurrence.

Example: Chart expressions

Example	Result
`FindOneOf( 'my example text string', 'et%s')`	Returns '4' because ‘e’ is the fourth character in the example string.
`FindOneOf( 'my example text string', 'et%s', 3)`	Returns '12' because the search is for any of the characters e, t, % or s, and "t" is the third occurrence in position 12 of the example string.
`FindOneOf( 'my example text string', '¤%&')`	Returns '0' because none of the characters ¤, %, or & exist in the example string.

Example: Load script

```
Load *
Inline
[SsearchFor, Occurrence
et%s,1
et%s,3
¤%&,1]
```

Result

SearchFor	Occurrence	FindOneOf('my example text string', SearchFor, Occurrence)
et%s	1	4
et%s	3	12
¤%&	1	0

Hash128

Hash128() returns a 128-bit hash of the combined input expression values. The result is a 22-character string.

Syntax:

```
Hash128(expr{, expression})
```

Return data type: string

Example: Chart expressions

Example	Result
Hash128 ('abc', 'xyz', '123')	Returns 'MA&5]6+3=:>:>G%S<U*S2+'.
Hash128 (Region, Year, Month) Note: Region, Year, and Month are table fields.	Returns 'G7*=6GKPJ(Z+)^KM?<$'A+'.

Example: Load script

```
Hash_128:
Load *,
Hash128(Region, Year, Month) as Hash128;
Load * inline [
Region, Year, Month
abc, xyz, 123
EU, 2022, 01
UK, 2022, 02
US, 2022, 02 ];
```

Result

Region	Year	Month	Hash128
abc	xyz	123	MA&5]6+3=:>;>G%S<U*S2+
EU	2022	01	B40^K&[T@!;VB'XR]<5=/$
UK	2022	02	O5T;+1?[B&"F&1//MA[MN!
US	2022	02	C6@#]4#_G-(]J7EQY#KRW0

Hash160

Hash160() returns a 160-bit hash of the combined input expression values. The result is a 27-character string.

Syntax:

```
Hash160(expr{, expression})
```

Return data type: string

Example: Chart expressions

Example	Result
Hash160 ('abc', 'xyz', '123')	Returns 'MA&5]6+3=:>;>G%S<U*S2I:`=X*'.
Hash160 (Region, Year, Month) Note: Region, Year, and Month are table fields.	Returns 'G7*=6GKPJ (Z+)^KM?<$'AI.)?U$'.

Example: Load script

```
Hash_160:
Load *,
Hash160(Region, Year, Month) as Hash160;
Load * inline [
Region, Year, Month
abc, xyz, 123
EU, 2022, 01
UK, 2022, 02
US, 2022, 02 ];
```

Result

Region	Year	Month	Hash160
abc	xyz	123	MA&5]6+3=:>;>G%S<U*S2I:`=X*

Region	Year	Month	Hash160
EU	2022	01	B40^K&[T@!;VB'XR]<5=//_F853
UK	2022	02	O5T;+1?[B&"F&1//MA[MN!T"FWZ
US	2022	02	C6@#]4#_G-(]J7EQY#KRW`@KF+W

Hash256

Hash256() returns a 256-bit hash of the combined input expression values. The result is a 43-character string.

Syntax:

```
Hash256(expr{, expression})
```

Return data type: string

Example: Chart expressions

Example	Result
Hash256 ('abc', 'xyz', '123')	Returns 'MA&5]6+3=:>;>G%S<U*S2I:`=X*A.IO*8N\%Y7Q;YEJ'.
Hash256 (Region, Year, Month) Note: Region, Year, and Month are table fields.	Returns 'G7*=6GKPJ(Z+)^KM?<$'AI.)?U$#X2RB [:0ZP=+Z`F:'.

Example: Load script

```
Hash_256:
Load *,
Hash256(Region, Year, Month) as Hash256;
Load * inline [
Region, Year, Month
abc, xyz, 123
EU, 2022, 01
UK, 2022, 02
US, 2022, 02 ];
```

Result

Region	Year	Month	Hash256
abc	xyz	123	MA&5]6+3=:>;>G%S<U*S2I:`=X*A.IO*8N\%Y7Q;YEJ
EU	2022	01	B40^K&[T@!;VB'XR]<5=//_F853?BE6'G&,YH*T'MF)
UK	2022	02	O5T;+1?[B&"F&1//MA[MN!T"FWZT=4\#V`M%6_\0C>4
US	2022	02	C6@#]4#_G-(]J7EQY#KRW`@KF+W-0]`[Z8R+#'")=+0

Index

Index() searches a string to find the starting position of the nth occurrence of a provided substring. An optional third argument provides the value of n, which is 1 if omitted. A negative value searches from the end of the string. The positions in the string are numbered from **1** and up.

Syntax:

```
Index(text, substring[, count])
```

Return data type: integer

Arguments:

Arguments

Argument	Description
text	The original string.
substring	A string of characters to search for in text.
count	Defines which occurrence of **substring** to search for. For example, a value of 2 searches for the second occurrence.

Examples and results:

Example	Result
Index('abcdefg', 'cd')	Returns 3
Index('abcdabcd', 'b', 2)	Returns 6 (the second occurrence of 'b')
Index('abcdabcd', 'b',-2)	Returns 2 (the second occurrence of 'b' starting from the end)
`Left( Date, Index( Date,'-') -1`) where **Date** = 1997-07-14	Returns 1997
`Mid( Date, Index( Date, '-', 2 ) -2, 2 )` where **Date** = 1997-07-14	Returns 07

Example: Script

```
T1:
Load
*,
index(String, 'cd') as Index_CD,              // returns 3  in Index_CD
index(String, 'b') as Index_B,               // returns 2 in Index_B
index(String, 'b', -1) as Index_B2;          // returns 2 or 6 in Index_B2
Load * inline [
String
```

```
abcdefg
abcdabcd ];
```

IsJson

IsJson() tests whether a specified string contains valid JSON (JavaScript Object Notation) data. You can also validate a specific JSON data type.

Syntax:

```
value IsJson(json [, type])
```

Return data type: dual

Arguments

Argument	Description
json	String to test. It can contain extra spaces or newlines.
type	Optional argument that specifies the JSON data type to test for. • 'value' (default) • 'object' • 'array' • 'string' • 'number' • 'Boolean' • 'null'

Example: Valid JSON and type

Example	Result
`IsJson('null')`	Returns -1 (true)
`IsJson('"abc"', 'value')`	Returns -1 (true)
`IsJson('"abc"', 'string')`	Returns -1 (true)
`IsJson(123, 'number')`	Returns -1 (true)

Example: Invalid JSON or type

Example	Result	Description
`IsJson('text')`	Returns 0 (false)	'text' is not a valid JSON value
`IsJson('"text"', 'number')`	Returns 0 (false)	'"text"' is not a valid JSON number
`IsJson('"text"', 'text')`	Returns 0 (false)	'text' is not a valid JSON type

JsonGet

JsonGet() returns the path of a JSON (JavaScript Object Notation) data string. The data must be valid JSON but can contain extra spaces or newlines.

Syntax:

```
value JsonGet(json, path)
```

Return data type: dual

Arguments

Argument	Description
json	String containing JSON data.
path	The path must be specified according to RFC 6901. This will allow lookup of properties inside JSON data without using complex substring or index functions.

Example: Valid JSON and path

Example	Result
`JsonGet('{"a":{"foo":"bar"},"b":[123,"abc","ABC"]}', '')`	Returns '{"a":{"foo":"bar"},"b":[123,"abc","ABC"]}'
`JsonGet('{"a":{"foo":"bar"},"b":[123,"abc","ABC"]}', '/a')`	Returns '{"foo":"bar"}'
`JsonGet('{"a":{"foo":"bar"},"b":[123,"abc","ABC"]}', '/a/foo')`	Returns '"bar"'
`JsonGet('{"a":{"foo":"bar"},"b":[123,"abc","ABC"]}', '/b')`	Returns '[123,"abc","ABC"]'
`JsonGet('{"a":{"foo":"bar"},"b":[123,"abc","ABC"]}', '/b/0')`	Returns '123'
`JsonGet('{"a":{"foo":"bar"},"b":[123,"abc","ABC"]}', '/b/1')`	Returns '"abc"'
`JsonGet('{"a":{"foo":"bar"},"b":[123,"abc","ABC"]}', '/b/2')`	Returns '"ABC"'

Example: Invalid JSON or path

Example	Result	Description
`JsonGet('{"a":"b"}','/b')`	Returns null	The path does not point to a valid part of the JSON data.
`JsonGet('{"a"}','/a')`	Returns null	The JSON data is not valid JSON (member "a" does not have a value).

JsonSet

JsonSet() modifies a string containing JSON (JavaScript Object Notation) data. It can set or insert a JSON value with the new location specified by the path. The data must be valid JSON but can contain extra spaces or newlines.

Syntax:

```
value JsonSet(json, path, value)
```

Return data type: dual

Arguments

Argument	Description
json	String containing JSON data.
path	The path must be specified according to RFC 6901. This allows buildup of properties inside JSON data without using complex substring or index functions and concatenation.
value	The new string value in JSON format.

Example: Valid JSON, path, and value

Example	Result
`JsonSet('{}','/a','"b"')`	Returns '{"a":"b"}'
`JsonSet('[]','/0','"x"')`	Returns '["x"]'
`JsonSet('"abc"','','123')`	Returns 123

Example: Invalid JSON, path, or value

Example	Result	Description
`JsonSet('"abc"','/x','123')`	Returns null	The path does not point to a valid part of the JSON data.
`JsonSet('{"a": {"b":"c"}}','a/b','"x"')`	Returns null	The path is invalid.
`JsonSet('{"a":"b"}','/a','abc')`	Returns null	The value is not valid JSON. A string must be enclosed in quotes.

KeepChar

KeepChar() returns a string consisting of the first string ,'text', less any of the characters NOT contained in the second string, "keep_chars".

Syntax:

```
KeepChar(text, keep_chars)
```

Return data type: string

Arguments:

Arguments

Argument	Description
text	The original string.
keep_chars	A string containing the characters in text to be kept.

Example: Chart expressions

Example	Result
`KeepChar ( 'a1b2c3','123' )`	Returns '123'.
`KeepChar ( 'a1b2c3','1234' )`	Returns '123'.
`KeepChar ( 'a1b22c3','1234' )`	Returns '1223'.
`KeepChar ( 'a1b2c3','312' )`	Returns '123'.

Example: Load script

```
T1:
Load
*,
keepchar(String1, String2) as KeepChar;
Load * inline [
String1, String2
'a1b2c3', '123'
];
```

Results

Qlik Sense table showing the output from using the *KeepChar* function in the load script.

String1	String2	KeepChar
a1b2c3	123	123

See also:

PurgeChar (page 1015)

Left

Left() returns a string consisting of the first (leftmost) characters of the input string, where the number of characters is determined by the second argument.

Syntax:

```
Left(text, count)
```

Return data type: string

Arguments:

Argument	Description
text	The original string.
count	Defines the number of characters to included from the left-hand part of the string **text**.

Example: Chart expression

Example	Result
`Left('abcdef', 3)`	Returns 'abc'

Example: Load script

```
T1:
Load
*,
left(Text,Start) as Left;
Load * inline [
Text, Start
'abcdef', 3
'2021-07-14', 4
'2021-07-14', 2
];
```

Result

Qlik Sense table showing the output from using the *Left* function in the load script.

Text	Start	Left
abcdef	3	abc
2021-07-14	4	2021
2021-07-14	2	20

See also *Index (page 1003)*, which allows more complex string analysis.

Len

Len() returns the length of the input string.

Syntax:

```
Len(text)
```

Return data type: integer

Example: Chart expression

Example	Result
`Len('Peter')`	Returns '5'

Example: Load script

```
T1:
Load String, First&Second as NewString;
Load *, mid(String,len(First)+1) as Second;
Load *, upper(left(String,1)) as First;
Load  * inline [
String
this is a sample text string
capitalize first letter only ];
```

Result

String	NewString
this is a sample text string	This is a sample text string
capitalize first letter only	Capitalize first letter only

LevenshteinDist

LevenshteinDist() returns the Levenshtein distance between two strings. It is defined as the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into the other. The function is useful for fuzzy string comparisons.

Syntax:

```
LevenshteinDist(text1, text2)
```

Return data type: integer

Example: Chart expression

Example	Result
`LevenshteinDist('Kitten','Sitting')`	Returns '3'

Example: Load script

Load script

```
T1:
Load *, recno() as ID;
Load 'Silver' as String_1,* inline [
String_2
```

```
Sliver
SSiver
SSiveer ];

T1:
Load *, recno()+3 as ID;
Load 'Gold' as String_1,* inline [
String_2
Bold
Bool
Bond ];

T1:
Load *, recno()+6 as ID;
Load 'Ove' as String_1,* inline [
String_2
Ove
Uve
Üve ];

T1:
Load *, recno()+9 as ID;
Load 'ABC' as String_1,* inline [
String_2
DEFG
abc
ピピピ ];

set nullinterpret = '<NULL>';
T1:
Load *, recno()+12 as ID;
Load 'X' as String_1,* inline [
String_2
''
<NULL>
1 ];

R1:
Load
ID,
String_1,
String_2,
LevenshteinDist(String_1, String_2) as LevenshteinDistance
resident T1;

Drop table T1;
```

Result

ID	String_1	String_2	LevenshteinDistance
1	Silver	Sliver	2

ID	String_1	String_2	LevenshteinDistance
2	Silver	SSiver	2
3	Silver	SSiveer	3
4	Gold	Bold	1
5	Gold	Bool	3
6	Gold	Bond	2
7	Ove	Ove	0
8	Ove	Uve	1
9	Ove	Üve	1
10	ABC	DEFG	4
11	ABC	abc	3
12	ABC	ビビビ	3
13	X		1
14	X	-	1
15	X	1	1

Lower

Lower() converts all the characters in the input string to lower case.

Syntax:

```
Lower(text)
```

Return data type: string

Example: Chart expression

Example	Result
`Lower('abcD')`	Returns 'abcd'

Example: Load script

```
Load
String,
Lower(String)
Inline
[String
rHode iSland
washingTon d.C.
new york];
```

Result

String	Lower(String)
rHode iSland	rhode island
washingTon d.C.	washington d.c.
new york	new york

LTrim

LTrim() returns the input string trimmed of any leading spaces.

Syntax:

```
LTrim(text)
```

Return data type: string

Example: Chart expressions

Example	Result
`LTrim( ' abc' )`	Returns 'abc'
`LTrim( 'abc ' )`	Returns 'abc '

Example: Load script

```
Set verbatim=1;
T1:

Load *,
len(LtrimString) as LtrimStringLength;
Load *,
ltrim(String) as LtrimString;
Load *,
len(String) as StringLength;
Load * Inline [
String
'   abc    '
' def  '];
```

The "Set verbatim=1" statement is included in the example to ensure that the spaces are not automatically trimmed before the demonstration of the ltrim function. See Verbatim (page 165) for more information.

Result

String	StringLength	LtrimStringLength
def	6	5
abc	10	7

See also:

RTrim (page 1018)

Mid

Mid() returns the part of the input string starting at the position of the character defined by the second argument, 'start', and returning the number of characters defined by the third argument, 'count'. If 'count' is omitted, the rest of the input string is returned. The first character in the input string is numbered 1.

Syntax:

```
Mid(text, start[, count])
```

Return data type: string

Arguments:

Arguments

Argument	Description
text	The original string.
start	Integer defining the position of the first character in text to include.
count	Defines the string length of the output string. If omitted, all characters from the position defined by **start** are included.

Example: Chart expressions

Example	Result
`Mid('abcdef',3 )`	Returns 'cdef'
`Mid('abcdef',3, 2 )`	Returns 'cd'

Example: Load script

```
T1:
Load *,
mid(Text,Start) as Mid1,
mid(Text,Start,Count) as Mid2;
Load * inline [
```

```
Text, Start, Count
'abcdef', 3, 2
'abcdef', 2, 3
'210714', 3, 2
'210714', 2, 3
];
```

Result

Qlik Sense table showing the output from using the *Mid* function in the load script.

Text	Start	Mid1	Count	Mid2
abcdef	2	bcdef	3	bcd
abcdef	3	cdef	2	cd
210714	2	10714	3	107
210714	3	0714	2	07

See also:

Index (page 1003)

Ord

Ord() returns the Unicode code point number of the first character of the input string.

Syntax:

```
Ord(text)
```

Return data type: integer

Examples and results:

Example: Chart expression

Example	Result
`Ord('A')`	Returns the integer 65.
`Ord('Ab')`	Returns the integer 65.

Example: Load script

```
//Guqin (Chinese: 古琴) - 7-stringed zithers
T2:
Load *,
ord(Chinese) as OrdUnicode,
ord(Western) as OrdASCII;
Load * inline [
Chinese, Western
```

```
古琴, Guqin ];
```

Result:

Chinese	Western	OrdASCII	OrdUnicode
古琴	Guqin	71	21476

PurgeChar

PurgeChar() returns a string consisting of the characters contained in the input string ('text'), excluding any that appear in the second argument ('remove_chars').

Syntax:

```
PurgeChar(text, remove_chars)
```

Return data type: string

Arguments:

Arguments

Argument	Description
text	The original string.
remove_chars	A string containing the characters in text to be removed.

Return data type: string

Example: Chart expressions

Example	Result
`PurgeChar ( 'a1b2c3','123' )`	Returns 'abc'.
`PurgeChar ( 'a1b2c3','312' )`	Returns 'abc'.

Example: Load script

```
T1:
Load
*,
purgechar(String1, String2) as PurgeChar;
Load * inline [
String1, String2
'a1b2c3', '123'
];
```

Results

Qlik Sense table showing the output from using the *PurgeChar* function in the load script.

String1	String2	PurgeChar
a1b2c3	123	abc

See also:

KeepChar (page 1006)

Repeat

Repeat() forms a string consisting of the input string repeated the number of times defined by the second argument.

Syntax:

```
Repeat(text[, repeat_count])
```

Return data type: string

Arguments:

Arguments

Argument	Description
text	The original string.
repeat_count	Defines the number of times the characters in the string **text** are to be repeated in the output string.

Example: Chart expression

Example	Result
`Repeat( ' * ', rating )` when **rating** = 4	Returns '****'

Example: Load script

```
T1:
Load *,
repeat(String,2) as Repeat;
Load * inline [
String
hello world!
hOw aRe you? ];
```

Result

String	Repeat
hello world!	hello world!hello world!
hOw aRe you?	hOw aRe you?hOw aRe you?

Replace

Replace() returns a string after replacing all occurrences of a given substring within the input string with another substring. The function is non-recursive and works from left to right.

Syntax:

```
Replace(text, from_str, to_str)
```

Return data type: string

Arguments:

Arguments

Argument	Description
text	The original string.
from_str	A string that may occur one or more times within the input string **text**.
to_str	The string that will replace all occurrences of **from_str** within the string **text**.

Examples and results:

Example	Result
`Replace('abccde','cc','xyz')`	Returns 'abxyzde'

See also:

Right

Right() returns a string consisting of the last (rightmost) characters of the input string, where the number of characters is determined by the second argument.

Syntax:

```
Right(text, count)
```

Return data type: string

Arguments:

Arguments

Argument	Description
text	The original string.
count	Defines the number of characters to be included from the rightmost part of the string **text**.

Example: Chart expression

Example	Result
`Right('abcdef', 3)`	Returns 'def'

Example: Load script

```
T1:
Load
*,
right(Text,Start) as Right;
Load * inline [
Text, Start
'abcdef', 3
'2021-07-14', 4
'2021-07-14', 2
];
```

Result

Qlik Sense table showing the output from using the *Right* function in the load script.

Text	Start	Right
abcdef	3	def
2021-07-14	4	7-14
2021-07-14	2	14

RTrim

RTrim() returns the input string trimmed of any trailing spaces.

Syntax:

```
RTrim(text)
```

Return data type: string

Example: Chart expressions

Example	Result
`RTrim( ' abc' )`	Returns ' abc'
`RTrim( 'abc ' )`	Returns 'abc'

Example: Load script

```
Set verbatim=1;

T1:

Load *, len(RtrimString) as RtrimStringLength;
Load *, rtrim(String) as RtrimString;
Load *, len(String) as StringLength;
Load * Inline [
String
'   abc    '
' def  '];
```

The "Set verbatim=1" statement is included in the example to ensure that the spaces are not automatically trimmed before the demonstration of the rtrim function. See Verbatim (page 165) for more information.

Result

String	StringLength	RtrimStringLength
def	6	4
abc	10	6

See also:

LTrim (page 1012)

SubField

SubField() is used to extract substring components from a parent string field, where the original record fields consist of two or more parts separated by a delimiter.

The **Subfield()** function can be used, for example, to extract first name and surname from a list of records consisting of full names, the component parts of a path name, or for extracting data from comma-separated tables.

If you use the **Subfield()** function in a **LOAD** statement with the optional field_no parameter left out, one full record will be generated for each substring. If several fields are loaded using **Subfield()** the Cartesian products of all combinations are created.

Syntax:

```
SubField(text, delimiter[, field_no ])
```

Return data type: string

Arguments:

Arguments

Argument	Description
text	The original string. This can be a hard-coded text, a variable, a dollar-sign expansion, or another expression.
delimiter	A character within the input **text** that divides the string into component parts.
field_no	The optional third argument is an integer that specifies which of the substrings of the parent string **text** is to be returned. Use the value 1 to return the first substring, 2 to return the second substring, and so on. • If **field_no** is a positive value, substrings are extracted from left to right. • If **field_no** is a negative value, substrings are extracted from right to left.

SubField() can be used instead of using complex combinations of functions such as Len(), Right(), Left (), Mid(), and other string functions.

Examples: Script and chart expressions using SubField

Examples - script and chart expressions

Basic examples

Example	Result
`SubField(S, ';' ,2)`	Returns 'cde' if **S** is 'abc;cde;efg'.
`SubField(S, ';' ,1)`	Returns an empty string if **S** is an empty string.
`SubField(S, ';' ,1)`	Returns an empty string if **S** is ';'.

Example	Result
Suppose you have a variable that holds a path name vMyPath, `Set vMyPath=\Users\ext_jrb\Documents\Qlik\Sense\Apps;.`	In a text & image chart, you can add a measure such as: `SubField(vMyPath, '\',-3)`, which results in 'Qlik', because it is the substring third from the right-hand end of the variable vMyPath.

Script example 1

Load script

Load the following script expressions and data in the data load editor.

```
FullName:

LOAD * inline [
Name
'Dave Owen'
'Joe Tem'
];

SepNames:

Load Name,
SubField(Name, ' ',1) as FirstName,
SubField(Name, ' ',-1) as Surname
Resident FullName;
Drop Table FullName;
```

Create a visualization

Create a table visualization in a Qlik Sense sheet with **Name**, **FirstName**, and **SurName** as dimensions.

Result

Name	FirstName	SurName
Dave Owen	Dave	Owen
Joe Tem	Joe	Tem

Explanation

The **SubField()** function extracts the first substring of **Name** by setting the **field_no** argument to 1. Since the value of **field_no** is positive, a left to right order is followed for extracting the subtring. A second function call extracts the second substring by setting the **field _no** argument to -1, which extracts the substring following a right to left order.

Script example 2

Load script

Load the following script expressions and data in the data load editor.

```
LOAD DISTINCT
Instrument,
SubField(Player,',') as Player,
SubField(Project,',') as Project;

Load * inline [
Instrument|Player|Project
Guitar|Neil,Mike|Music,Video
Guitar|Neil|Music,OST
Synth|Neil,Jen|Music,Video,OST
Synth|Jo|Music
Guitar|Neil,Mike|Music,OST
] (delimiter is '|');
```

Create a visualization

Create a table visualization in a Qlik Sense sheet with **Instrument**, **Player**, and **Project** as dimensions.

Result

Instrument	Player	Project
Guitar	Mike	Music
Guitar	Mike	Video
Guitar	Mike	OST
Guitar	Neil	Music
Guitar	Neil	Video
Guitar	Neil	OST
Synth	Jen	Music
Synth	Jen	Video
Synth	Jen	OST
Synth	Jo	Music
Synth	Neil	Music
Synth	Neil	Video
Synth	Neil	OST

Explanation

This example shows how using multiple instances of the **Subfield()** function, each with the field_no parameter left out, from within the same **LOAD** statement creates Cartesian products of all combinations. The **DISTINCT** option is used to avoid creating duplicate records.

SubStringCount

SubStringCount() returns the number of occurrences of the specified substring in the input string text. If there is no match, 0 is returned.

Syntax:

```
SubStringCount(text, sub_string)
```

Return data type: integer

Arguments:

Argument	Description
text	The original string.
sub_string	A string which may occur one or more times within the input string **text**.

Example: Chart expressions

Example	Result
`SubStringCount ( 'abcdefgcdxyz', 'cd' )`	Returns '2'
`SubStringCount ( 'abcdefgcdxyz', 'dc' )`	Returns '0'

Example: Load script

```
T1:
Load *,
substringcount(upper(Strings),'AB') as SubStringCount_AB;
Load * inline [
Strings
ABC:DEF:GHI:AB:CD:EF:GH
aB/cd/ef/gh/Abc/abandoned ];
```

Result

Strings	SubStringCount_AB
aB/cd/ef/gh/Abc/abandoned	3
ABC:DEF:GHI:AB:CD:EF:GH	2

TextBetween

TextBetween() returns the text in the input string that occurs between the characters specified as delimiters.

Syntax:

```
TextBetween(text, delimiter1, delimiter2[, n])
```

Return data type: string

Arguments:

Argument	Description
text	The original string.
delimiter1	Specifies the first delimiting character (or string) to search for in **text**.
delimiter2	Specifies the second delimiting character (or string) to search for in **text**.
n	Defines which occurrence of the delimiter pair to search between. For example, a value of 2 returns the characters between the second occurrence of delimiter1 and the second occurrence of delimiter2.

Example: Chart expressions

Example	Result
`TextBetween('<abc>', '<', '>')`	Returns 'abc'
`TextBetween('<abc><de>', '<', '>',2)`	Returns 'de'
`TextBetween('abc', '<', '>')` `TextBetween('<a<b', '<', '>')`	Both examples return NULL. If any of the delimiter is not found in the string, NULL is returned.
`TextBetween('<>', '<', '>')`	Returns a zero-length string.
`TextBetween('<abc>', '<', '>', 2)`	Returns NULL, as n is greater than the number of occurrences of the delimiters.

Example: Load script

```
Load *,
textbetween(Text,'<','>') as TextBetween,
textbetween(Text,'<','>',2) as SecondTextBetween;
Load * inline [
Text
<abc><de>
<def><ghi><jkl> ];
```

Result

Text	TextBetween	SecondTextBetween
<abc><de>	abc	de
<def><ghi><jkl>	def	ghi

Trim

Trim() returns the input string trimmed of any leading and trailing spaces.

Syntax:

```
Trim(text)
```

Return data type: string

Examples and results:

Example: Chart expression

Example	Result
`Trim( ' abc' )`	Returns 'abc'
`Trim( 'abc ' )`	Returns 'abc'
`Trim( ' abc ' )`	Returns 'abc'

Example: Load script

```
Set verbatim=1;

T1:
Load *, len(TrimString) as TrimStringLength;
Load *, trim(String) as TrimString;
Load *, len(String) as StringLength;
Load * inline [
String
'   abc    '
' def  ](delimiter is '\t');
```

The "Set verbatim=1" statement is included in the example to ensure that the spaces are not automatically trimmed before the demonstration of the trim function. See Verbatim (page 165) for more information.

Result:

String	StringLength	TrimStringLength
def	6	3
abc	10	3

Upper

Upper() converts all the characters in the input string to upper case for all text characters in the expression. Numbers and symbols are ignored.

Syntax:

```
Upper(text)
```

Return data type: string

Example: Chart expression

Example	Result
`Upper(' abcD')`	Returns 'ABCD'

Example: Load script

```
Load
String,Upper(String)
Inline
[String
rHode iSland
washingTon d.C.
new york];
```

Result

String	Upper(String)
rHode iSland	RHODE ISLAND
washingTon d.C.	WASHINGTON D.C.
new york	NEW YORK

5.25 System functions

System functions provide functions for accessing system, device and Qlik Sense app properties.

System functions overview

Some of the functions are described further after the overview. For those functions, you can click the function name in the syntax to immediately access the details for that specific function.

Author()

This function returns a string containing the author property of the current app. It can be used in both the data load script and in a chart expression.

> *Author property can not be set in the current version of Qlik Sense. If you migrate a QlikView document, the author property will be retained.*

ClientPlatform()

This function returns the user agent string of the client browser. It can be used in both the data load script and in a chart expression.

Example:

```
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/35.0.1916.114 Safari/537.36
```

ComputerName

This function returns a string containing the name of the computer as returned by the operating system. It can be used in both the data load script and in a chart expression.

> *If the name of the computer has more than 15 characters, the string will only contain the first 15 characters.*

```
ComputerName( )
```

DocumentName

This function returns a string containing the name of the current Qlik Sense app, without path but with extension. It can be used in both the data load script and in a chart expression.

```
DocumentName( )
```

DocumentPath

This function returns a string containing the full path to the current Qlik Sense app. It can be used in both the data load script and in a chart expression.

```
DocumentPath( )
```

> *This function is not supported in standard mode.*

DocumentTitle

This function returns a string containing the title of the current Qlik Sense app. It can be used in both the data load script and in a chart expression.

```
DocumentTitle( )
```

EngineVersion

This function returns the full Qlik Sense engine version as a string.

```
EngineVersion ()
```

GetCollationLocale

This script function returns the culture name of the collation locale that is used. If the variable CollationLocale has not been set, the actual user machine locale is returned.

```
GetCollationLocale( )
```

GetObjectField

GetObjectField() returns the name of the dimension. **Index** is an optional integer denoting the dimension that should be returned.

```
GetObjectField - chart function([index])
```

GetRegistryString

This function returns the value of a key in the Windows registry. It can be used in both the data load script and in a chart expression.

```
GetRegistryString(path, key)
```

This function is not supported in standard mode.

IsPartialReload

This function returns - 1 (True) if the current reload is partial, otherwise 0 (False).

```
IsPartialReload ()
```

OSUser

This function returns a string containing the name of the user that is currently connected. It can be used in both the data load script and in a chart expression.

```
OSUser( )
```

In Qlik Sense Desktop and Qlik Sense Mobile Client Managed, this function always returns 'Personal\Me'.

ProductVersion

This function returns the full Qlik Sense version and build number as a string.

This function is deprecated and replaced by **EngineVersion()**.

```
ProductVersion ()
```

ReloadTime

This function returns a timestamp for when the last data load finished. It can be used in both the data load script and in a chart expression.

```
ReloadTime( )
```

StateName

StateName() returns the name of the alternate state of the visualization in which it is used. StateName can be used, for example, to create visualizations with dynamic text and colors to reflect when the state of a visualization is changed. This function can be used in chart expressions, but cannot be used to determine the state that the expression refers to.

```
StateName - chart function()
```

EngineVersion

This function returns the full Qlik Sense engine version as a string.

Syntax:

```
EngineVersion()
```

IsPartialReload

This function returns - 1 (True) if the current reload is partial, otherwise 0 (False).

Syntax:

```
IsPartialReload()
```

ProductVersion

This function returns the full Qlik Sense version and build number as a string. This function is deprecated and replaced by **EngineVersion()**.

Syntax:

```
ProductVersion()
```

StateName - chart function

StateName() returns the name of the alternate state of the visualization in which it is used. StateName can be used, for example, to create visualizations with dynamic text and colors to reflect when the state of a visualization is changed. This function can be used in chart expressions, but cannot be used to determine the state that the expression refers to.

Syntax:

```
StateName ()
```

Example 1:

Dynamic Text

```
='Region - ' & if(StateName() = '$', 'Default', StateName())
```

Example 2:

Dynamic Colors

```
if(StateName() = 'Group 1', rgb(152, 171, 206),
      if(StateName() = 'Group 2', rgb(187, 200, 179),
            rgb(210, 210, 210)
      )
)
```

5.26 Table functions

The table functions return information about the data table which is currently being read. If no table name is specified and the function is used within a **LOAD** statement, the current table is assumed.

All functions can be used in the data load script, while only **NoOfRows** can be used in a chart expression.

Table functions overview

Some of the functions are described further after the overview. For those functions, you can click the function name in the syntax to immediately access the details for that specific function.

FieldName

The **FieldName** script function returns the name of the field with the specified number within a previously loaded table. If the function is used within a **LOAD** statement, it must not reference the table currently being loaded.

```
FieldName (field_number ,table_name)
```

FieldNumber

The **FieldNumber** script function returns the number of a specified field within a previously loaded table. If the function is used within a **LOAD** statement, it must not reference the table currently being loaded.

```
FieldNumber (field_name ,table_name)
```

NoOfFields

The **NoOfFields** script function returns the number of fields in a previously loaded table. If the function is used within a **LOAD** statement, it must not reference the table currently being loaded.

```
NoOfFields (table_name)
```

NoOfRows

The **NoOfRows** function returns the number of rows (records) in a previously loaded table. If the function is used within a **LOAD** statement, it must not reference the table currently being loaded.

```
NoOfRows (table_name)
```

NoOfTables

This script function returns the number of tables previously loaded.

```
NoOfTables()
```

TableName

This script function returns the name of the table with the specified number.

```
TableName(table_number)
```

TableNumber

This script function returns the number of the specified table. The first table has number 0.

If table_name does not exist, NULL is returned.

```
TableNumber(table_name)
```

Example:

In this example, we want to create a table with information about the tables and fields that have been loaded.

First we load some sample data. This creates the two tables that will be used to illustrate the table functions described in this section.

```
Characters:
Load Chr(RecNo()+Ord('A')-1) as Alpha, RecNo() as Num autogenerate 26;

ASCII:
Load
 if(RecNo()>=65 and RecNo()<=90,RecNo()-64) as Num,
 Chr(RecNo()) as AsciiAlpha,
 RecNo() as AsciiNum
autogenerate 255
 Where (RecNo()>=32 and RecNo()<=126) or RecNo()>=160 ;
```

Next, we iterate through the tables that have been loaded, using the **NoOfTables** function, and then through the fields of each table, using the **NoOfFields** function, and load information using the table functions.

```
//Iterate through the loaded tables
For t = 0 to NoOfTables() - 1

//Iterate through the fields of table
 For f = 1 to NoOfFields(TableName($(t)))
  Tables:
  Load
   TableName($(t)) as Table,
   TableNumber(TableName($(t))) as TableNo,
   NoOfRows(TableName($(t))) as TableRows,
   FieldName($(f),TableName($(t))) as Field,
   FieldNumber(FieldName($(f),TableName($(t))),TableName($(t))) as FieldNo
   Autogenerate 1;
 Next f
Next t;
```

The resulting table Tables will look like this:

Load table

Table	TableNo	TableRows	Field	FieldNo
Characters	0	26	Alpha	1
Characters	0	26	Num	2

Table	TableNo	TableRows	Field	FieldNo
ASCII	1	191	Num	1
ASCII	1	191	AsciiAlpha	2
ASCII	1	191	AsciiNum	3

FieldName

The **FieldName** script function returns the name of the field with the specified number within a previously loaded table. If the function is used within a **LOAD** statement, it must not reference the table currently being loaded.

Syntax:

```
FieldName(field_number ,table_name)
```

Arguments:

Arguments

Argument	Description
field_number	The field number of the field you want to reference.
table_name	The table containing the field you want to reference.

Example:

```
LET a = FieldName(4,'tab1');
```

FieldNumber

The **FieldNumber** script function returns the number of a specified field within a previously loaded table. If the function is used within a **LOAD** statement, it must not reference the table currently being loaded.

Syntax:

```
FieldNumber(field_name ,table_name)
```

Arguments:

Arguments

Argument	Description
field_name	The name of the field.
table_name	The name of the table containing the field.

If the field field_name does not exist in table_name, or table_name does not exist, the function returns 0.

Example:

```
LET a = FieldNumber('Customer','tab1');
```

NoOfFields

The **NoOfFields** script function returns the number of fields in a previously loaded table. If the function is used within a **LOAD** statement, it must not reference the table currently being loaded.

Syntax:

```
NoOfFields(table_name)
```

Arguments:

Arguments

Argument	Description
table_name	The name of the table.

Example:

```
LET a = NoOfFields('tab1');
```

NoOfRows

The **NoOfRows** function returns the number of rows (records) in a previously loaded table. If the function is used within a **LOAD** statement, it must not reference the table currently being loaded.

Syntax:

```
NoOfRows(table_name)
```

Arguments:

Arguments

Argument	Description
table_name	The name of the table.

Example:

```
LET a = NoOfRows('tab1');
```

5.27 Trigonometric and hyperbolic functions

This section describes functions for performing trigonometric and hyperbolic operations. In all of the functions, the arguments are expressions resolving to angles measured in radians, where **x** should be interpreted as a real number.

All angles are measured in radians.

All functions can be used in both the data load script and in chart expressions.

cos
Cosine of **x**. The result is a number between -1 and 1.

```
cos( x )
```

acos
Inverse cosine of **x**. The function is only defined if -1≤**x**≤1. The result is a number between 0 and π.

```
acos( x )
```

sin
Sine of **x**. The result is a number between -1 and 1.

```
sin( x )
```

asin
Inverse sine of **x**. The function is only defined if -1≤**x**≤1. The result is a number between - π/2 and π/2.

```
asin( x )
```

tan
Tangent of **x**. The result is a real number.

```
tan( x )
```

atan
Inverse tangent of **x**. The result is a number between - π/2 and π/2.

```
atan( x )
```

atan2
Two-dimensional generalization of the inverse tangent function. Returns the angle between the origin and the point represented by the coordinates **x** and **y**. The result is a number between - π and + π.

```
atan2( y,x )
```

cosh
Hyperbolic cosine of **x**. The result is a positive real number.

```
cosh( x )
```

sinh

Hyperbolic sine of **x**. The result is a real number.

```
sinh( x )
```

tanh

Hyperbolic tangent of **x**. The result is a real number.

```
tanh( x )
```

acosh

Inverse hyperbolic cosine of **x**. The result is a positive real number.

```
acosh( x )
```

asinh

Inverse hyperbolic sine of **x**. The result is a real number.

```
asinh( x )
```

atanh

Inverse hyperbolic tangent of **x**. The result is a real number.

```
atanh( x )
```

Examples:

The following script code loads a sample table, and then loads a table containing the calculated trigonometric and hyperbolic operations on the values.

```
SampleData:
LOAD * Inline
[Value
-1
0
1];

Results:
Load *,
cos(Value),
acos(Value),
sin(Value),
asin(Value),
tan(Value),
atan(Value),
atan2(Value, Value),
cosh(Value),
sinh(Value),
tanh(Value)
RESIDENT SampleData;

Drop Table SampleData;
```

6 File system access restriction

For security reasons, Qlik Sense in standard mode does not support paths in the data load script or functions and variables that expose the file system.

However, since file system paths were supported in QlikView, it is possible to disable standard mode and use legacy mode in order to reuse QlikView load scripts.

Disabling standard mode can create a security risk by exposing the file system.

Disabling standard mode (page 1041)

6.1 Security aspects when connecting to file based ODBC and OLE DB data connections

ODBC and OLE DB data connections using file-based drivers will expose the path to the connected data file in the connection string. The path can be exposed when the connection is edited, in the data selection dialog, or in certain SQL queries. This is the case both in standard mode and legacy mode.

If exposing the path to the data file is a concern, it is recommended to connect to the data file using a folder data connection if it is possible.

6.2 Limitations in standard mode

Several statements, variables and functions cannot be used or have limitations in standard mode. Using unsupported statements in the data load script produces an error when the load script runs. Error messages can be found in the script log file. Using unsupported variables and functions does not produce error messages or log file entries. Instead, the function returns NULL.

There is no indication that a variable, statement or function is unsupported when you are editing the data load script.

System variables

System variables

Variable	Standard mode	Legacy mode	Definition
Floppy	Not supported	Supported	Returns the drive letter of the first floppy drive found, normally *a:*.

Variable	Standard mode	Legacy mode	Definition
CD	Not supported	Supported	Returns the drive letter of the first CD-ROM drive found. If no CD-ROM is found, then *c:* is returned.
QvPath	Not supported	Supported	Returns the browse string to the Qlik Sense executable.
QvRoot	Not supported	Supported	Returns the root directory of the Qlik Sense executable.
QvWorkPath	Not supported	Supported	Returns the browse string to the current Qlik Sense app.
QvWorkRoot	Not supported	Supported	Returns the root directory of the current Qlik Sense app.
WinPath	Not supported	Supported	Returns the browse string to Windows.
WinRoot	Not supported	Supported	Returns the root directory of Windows.
$(include=...)	Supported input: Path using library connection	Supported input: Path using library connection or file system	The **Include/Must_Include** variable specifies a file that contains text that should be included in the script and evaluated as script code. It is not used to add data. You can store parts of your script code in a separate text file and reuse it in several apps. This is a user-defined variable.

Regular script statements

Regular script statements

Statement	Standard mode	Legacy mode	Definition
Binary	Supported input: Path using library connection	Supported input: Path using library connection or file system	The **binary** statement is used for loading data from another app.
Connect	Supported input: Path using library connection	Supported input: Path using library connection or file system	The **CONNECT** statement is used to define Qlik Sense access to a general database through the OLE DB/ODBC interface. For ODBC, the data source first needs to be specified using the ODBC administrator.
Directory	Supported input: Path using library connection	Supported input: Path using library connection or file system	The **Directory** statement defines which directory to look in for data files in subsequent **LOAD** statements, until a new **Directory** statement is made.
Execute	Not supported	Supported input: Path using library connection or file system	The **Execute** statement is used to run other programs while Qlik Sense is loading data. For example, to make conversions that are necessary.
LOAD from ...	Supported input: Path using library connection	Supported input: Path using library connection or file system	The **LOAD** statement loads fields from a file, from data defined in the script, from a previously loaded table, from a web page, from the result of a subsequent **SELECT** statement or by generating data automatically.

Statement	Standard mode	Legacy mode	Definition
Store into ...	Supported input: Path using library connection	Supported input: Path using library connection or file system	The **Store** statement creates a QVD, CSV, or text file.

Script control statements

Script control statements

Statement	Standard mode	Legacy mode	Definition
For each... filelist mask/dirlist mask	Supported input: Path using library connection Returned output: Library connection	Supported input: Path using library connection or file system Returned output: Library connection or file system path, depending on input	The filelist mask syntax produces a comma separated list of all files in the current directory matching the **filelist mask**. The **dirlist mask** syntax produces a comma separated list of all directories in the current directory matching the directory name mask.

File functions

File functions

Function	Standard mode	Legacy mode	Definition
Attribute()	Supported input: Path using library connection	Supported input: Path using library connection or file system	Returns the value of the meta tags of different media files as text.
ConnectString()	Returned output: Library connection name	Library connection name or actual connection, depending on input	Returns the active connect string for ODBC or OLE DB connections.
FileDir()	Returned output: Library connection	Returned output: Library connection or file system path, depending on input	The **FileDir** function returns a string containing the path to the directory of the table file currently being read.

Function	Standard mode	Legacy mode	Definition
FilePath()	Returned output: Library connection	Returned output: Library connection or file system path, depending on input	The **FilePath** function returns a string containing the full path to the table file currently being read.
FileSize()	Supported input: Path using library connection	Supported input: Path using library connection or file system	The **FileSize** function returns an integer containing the size in bytes of the file filename or, if no filename is specified, of the table file currently being read.
FileTime()	Supported input: Path using library connection	Supported input: Path using library connection or file system	The **FileTime** function returns a timestamp in UTC format of the last modification of a specified file. If a file is not specified, the function returns a timestamp in UTC of the last modification of the currently read table file.
GetFolderPath()	Not supported	Returned output: Absolute path	The **GetFolderPath** function returns the value of the Microsoft Windows *SHGetFolderPath* function. This function takes as input the name of a Microsoft Windows folder and returns the full path of the folder.
QvdCreateTime()	Supported input: Path using library connection	Supported input: Path using library connection or file system	This script function returns the XML-header time stamp from a QVD file, if any is present, otherwise it returns NULL. In the timestamp, time is provided in UTC.

Function	Standard mode	Legacy mode	Definition
QvdFieldName()	Supported input: Path using library connection	Supported input: Path using library connection or file system	This script function returns the name of field number **fieldno** in a QVD file. If the field does not exist NULL is returned.
QvdNoOfFields()	Supported input: Path using library connection	Supported input: Path using library connection or file system	This script function returns the number of fields in a QVD file.
QvdNoOfRecords()	Supported input: Path using library connection	Supported input: Path using library connection or file system	This script function returns the number of records currently in a QVD file.
QvdTableName()	Supported input: Path using library connection	Supported input: Path using library connection or file system	This script function returns the name of the table stored in a QVD file.

System functions

System functions

Function	Standard mode	Legacy mode	Definition
DocumentPath()	Not supported	Returned output: Absolute path	This function returns a string containing the full path to the current Qlik Sense app.
GetRegistryString()	Not supported	Supported	Returns the value of a named registry key with a given registry path. This function can be used in chart and script alike.

6.3 Disabling standard mode

You can disable standard mode, or in other words, set legacy mode, in order to reuse QlikView load scripts that refer to absolute or relative file paths as well as library connections.

Disabling standard mode can create a security risk by exposing the file system.

Qlik Sense

For Qlik Sense, standard mode can be disabled in QMC using the **Standard mode** property.

Qlik Sense Desktop

In Qlik Sense Desktop, you can set standard/legacy mode in *Settings.ini*.

If you installed Qlik Sense Desktop using the default installation location, *Settings.ini* is located in *C:\Users\{user}\Documents\Qlik\Sense\Settings.ini*. If you installed Qlik Sense Desktop to a folder that you selected, *Settings.ini* is located in the *Engine* folder of the installation path.

Do the following:

1. Open *Settings.ini* in a text editor.
2. Change *StandardReload=1* to *StandardReload=0*.
3. Save the file and start Qlik Sense Desktop.

Qlik Sense Desktop now runs in legacy mode.

Settings

The available settings for StandardReload are:

- 1 (standard mode)
- 0 (legacy mode)

6 Chart level scripting

When modifying chart data, you use a sub-set of the Qlik Sense script which consists of a number of statements. A statement can be either a regular script statement or a script control statement. Certain statements can be preceded by prefixes.

Regular statements are typically used for manipulating data in one way or another. These statements may be written over any number of lines in the script and must always be terminated by a semicolon, ";".

Control statements are typically used for controlling the flow of the script execution. Each clause of a control statement must be kept inside one script line and may be terminated by a semicolon or the end-of-line.

Prefixes may be applied to applicable regular statements but never to control statements.

All script keywords can be typed with any combination of lower case and upper case characters. Field and variable names used in the statements are however case sensitive.

In this section you can find an alphabetical listing of all script statements, control statements and prefixes available in the sub-set of the script used when modifying chart data.

6.4 Control statements

When modifying chart data, you use a sub-set of the Qlik Sense script which consists of a number of statements. A statement can be either a regular script statement or a script control statement.

Control statements are typically used for controlling the flow of the script execution. Each clause of a control statement must be kept inside one script line and may be terminated by semicolon or end-of-line.

Prefixes are never applied to control statements.

All script keywords can be typed with any combination of lower case and upper case characters.

Chart modifier control statements overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Call

The **call** control statement calls a subroutine which must be defined by a previous **sub** statement.

```
Call name ( [ paramlist ])
```

Do..loop

The **do..loop** control statement is a script iteration construct which executes one or several statements until a logical condition is met.

```
Do..loop [ ( while | until ) condition ] [statements]
[exit do [ ( when | unless ) condition ] [statements]
loop [ ( while | until ) condition ]
```

End

The **End** script keyword is used to close **If**, **Sub** and **Switch** clauses.

Exit

The **Exit** script keyword is part of the **Exit Script** statement, but can also be used to exit **Do**, **For** or **Sub** clauses.

Exit script

This control statement stops script execution. It may be inserted anywhere in the script.

```
Exit script[ (when | unless) condition ]
```

For..next

The **for..next** control statement is a script iteration construct with a counter. The statements inside the loop enclosed by **for** and **next** will be executed for each value of the counter variable between specified low and high limits.

```
For..next  counter = expr1 to expr2 [ stepexpr3 ]
[statements]
[exit for [ ( when | unless ) condition ]
[statements]
Next [counter]
```

For each ..next

The **for each..next** control statement is a script iteration construct which executes one or several statements for each value in a comma separated list. The statements inside the loop enclosed by **for** and **next** will be executed for each value of the list.

```
For each..next var in list
[statements]
[exit for [ ( when | unless ) condition ]
[statements]
next [var]
```

If..then

The **if..then** control statement is a script selection construct forcing the script execution to follow different paths depending on one or several logical conditions.

> *Since the **if..then** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its four possible clauses (**if..then**, **elseif..then**, **else** and **end if**) must not cross a line boundary.*

```
If..then..elseif..else..end if condition then
  [ statements ]
{ elseif condition then
  [ statements ] }
[ else
  [ statements ] ]
end if
```

Next

The **Next** script keyword is used to close **For** loops.

Sub

The **sub..end sub** control statement defines a subroutine which can be called upon from a **call** statement.

```
Sub..end sub name [ ( paramlist )] statements end sub
```

Switch

The **switch** control statement is a script selection construct forcing the script execution to follow different paths, depending on the value of an expression.

```
Switch..case..default..end switch expression {case valuelist [ statements ]}
[default statements] end switch
```

To

The **To** script keyword is used in several script statements.

Call

The **call** control statement calls a subroutine which must be defined by a previous **sub** statement.

Syntax:

```
Call name ( [ paramlist ])
```

Arguments:

Arguments

Argument	Description
name	The name of the subroutine.
paramlist	A comma separated list of the actual parameters to be sent to the subroutine. Each item in the list may be a field name, a variable or an arbitrary expression.

The subroutine called by a **call** statement must be defined by a **sub** encountered earlier during script execution.

Parameters are copied into the subroutine and, if the parameter in the **call** statement is a variable and not an expression, copied back out again upon exiting the subroutine.

Limitations:

- Since the **call** statement is a control statement and as such is ended with either a semicolon or end-of-line, it must not cross a line boundary.
- When you define a subroutine with `Sub..end sub` inside a control statement, for example `if..then`, you can only call the subroutine from within the same control statement.

Do..loop

The **do..loop** control statement is a script iteration construct which executes one or several statements until a logical condition is met.

Syntax:

```
Do [ ( while | until ) condition ] [statements]
[exit do [ ( when | unless ) condition ] [statements]
loop[ ( while | until ) condition ]
```

*Since the **do..loop** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its three possible clauses (**do**, **exit do** and **loop**) must not cross a line boundary.*

Arguments:

Arguments

Argument	Description
condition	A logical expression evaluating to True or False.
statements	Any group of one or more Qlik Sense script statements.
while / until	The **while** or **until** conditional clause must only appear once in any **do..loop** statement, i.e. either after **do** or after **loop**. Each condition is interpreted only the first time it is encountered but is evaluated for every time it encountered in the loop.
exit do	If an **exit do** clause is encountered inside the loop, the execution of the script will be transferred to the first statement after the **loop** clause denoting the end of the loop. An **exit do** clause can be made conditional by the optional use of a **when** or **unless** suffix.

End

The **End** script keyword is used to close **If**, **Sub** and **Switch** clauses.

Exit

The **Exit** script keyword is part of the **Exit Script** statement, but can also be used to exit **Do**, **For** or **Sub** clauses.

Exit script

This control statement stops script execution. It may be inserted anywhere in the script.

Syntax:

```
Exit Script [ (when | unless) condition ]
```

Since the **exit script** statement is a control statement and as such is ended with either a semicolon or end-of-line, it must not cross a line boundary.

Arguments:

Arguments

Argument	Description
condition	A logical expression evaluating to True or False.
when / unless	An **exit script** statement can be made conditional by the optional use of **when** or **unless** clause.

Examples:

```
//Exit script
Exit Script;

//Exit script when a condition is fulfilled
Exit Script when a=1
```

For..next

The **for..next** control statement is a script iteration construct with a counter. The statements inside the loop enclosed by **for** and **next** will be executed for each value of the counter variable between specified low and high limits.

Syntax:

```
For counter = expr1 to expr2 [ step expr3 ]
[statements]
[exit for [ ( when | unless ) condition ]
[statements]
Next [counter]
```

The expressions *expr1*, *expr2* and *expr3* are only evaluated the first time the loop is entered. The value of the counter variable may be changed by statements inside the loop, but this is not good programming practice.

If an **exit for** clause is encountered inside the loop, the execution of the script will be transferred to the first statement after the **next** clause denoting the end of the loop. An **exit for** clause can be made conditional by the optional use of a **when** or **unless** suffix.

*Since the **for..next** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its three possible clauses (**for..to..step**, **exit for** and **next**) must not cross a line boundary.*

Arguments:

Arguments

Argument	Description
counter	A variable name. If *counter* is specified after **next** it must be the same variable name as the one found after the corresponding **for**.
expr1	An expression which determines the first value of the *counter* variable for which the loop should be executed.
expr2	An expression which determines the last value of the *counter* variable for which the loop should be executed.
expr3	An expression which determines the value indicating the increment of the *counter* variable each time the loop has been executed.
condition	a logical expression evaluating to True or False.
statements	Any group of one or more Qlik Sense script statements.

For each..next

The **for each..next** control statement is a script iteration construct which executes one or several statements for each value in a comma separated list. The statements inside the loop enclosed by **for** and **next** will be executed for each value of the list.

Syntax:
Special syntax makes it possible to generate lists with file and directory names in the current directory.

```
for each var in list
[statements]
[exit for [ ( when | unless ) condition ]
[statements]
next [var]
```

Arguments:

Arguments

Argument	Description
var	A script variable name which will acquire a new value from list for each loop execution. If **var** is specified after **next** it must be the same variable name as the one found after the corresponding **for each**.

The value of the **var** variable may be changed by statements inside the loop, but this is not good programming practice.

If an **exit for** clause is encountered inside the loop, the execution of the script will be transferred to the first statement after the **next** clause denoting the end of the loop. An **exit for** clause can be made conditional by the optional use of a **when** or **unless** suffix.

> *Since the **for each..next** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its three possible clauses (**for each**, **exit for** and **next**) must not cross a line boundary.*

Syntax:

```
list := item { , item }
item := constant | (expression) | filelist mask | dirlist mask |
fieldvaluelist mask
```

Arguments

Argument	Description
constant	Any number or string. Note that a string written directly in the script must be enclosed by single quotes. A string without single quotes will be interpreted as a variable, and the value of the variable will be used. Numbers do not need to be enclosed by single quotes.
expression	An arbitrary expression.
mask	A filename or folder name mask which may include any valid filename characters as well as the standard wildcard characters, ***** and **?**. You can use absolute file paths or lib:// paths.
condition	A logical expression evaluating to True or False.
statements	Any group of one or more Qlik Sense script statements.
filelist mask	This syntax produces a comma separated list of all files in the current directory matching the filename mask. *This argument supports only library connections in standard mode.*
dirlist mask	This syntax produces a comma separated list of all folders in the current folder matching the folder name mask. *This argument supports only library connections in standard mode.*
fieldvaluelist mask	This syntax iterates through the values of a field already loaded into Qlik Sense.

*The Qlik Web Storage Provider Connectors and other DataFiles connections do not support filter masks that use wildcard (***** and **?**) characters.*

Example 1: Loading a list of files

```
// LOAD the files 1.csv, 3.csv, 7.csv and xyz.csv
for each a in 1,3,7,'xyz'
   LOAD * from file$(a).csv;
next
```

Example 2: Creating a list of files on disk

This example loads a list of all Qlik Sense related files in a folder.

```
sub DoDir (Root)
   for each Ext in 'qvw', 'qva', 'qvo', 'qvs', 'qvc', 'qvf', 'qvd'

      for each File in filelist (Root&'/*.' &Ext)

         LOAD
            '$(File)' as Name,
            FileSize( '$(File)' ) as Size,
            FileTime( '$(File)' ) as FileTime
         autogenerate 1;

      next File

   next Ext
   for each Dir in dirlist (Root&'/*' )

      call DoDir (Dir)

   next Dir

end sub

call DoDir ('lib://DataFiles')
```

Example 3: Iterating through a the values of a field

This example iterates through the list of loaded values of FIELD and generates a new field, NEWFIELD. For each value of FIELD, two NEWFIELD records will be created.

```
load * inline [
FIELD
one
two
three
];

FOR Each a in FieldValueList('FIELD')
```

```
LOAD '$(a)' &'-'&RecNo() as NEWFIELD AutoGenerate 2;
NEXT a
```

The resulting table looks like this:

Example table

NEWFIELD
one-1
one-2
two-1
two-2
three-1
three-2

If..then..elseif..else..end if

The **if..then** control statement is a script selection construct forcing the script execution to follow different paths depending on one or several logical conditions.

Control statements are typically used to control the flow of the script execution. In a chart expression, use the **if** conditional function instead.

Syntax:

```
If  condition then
  [ statements ]
{ elseif condition then
  [ statements ] }
[ else
  [ statements ] ]
end if
```

Since the **if..then** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its four possible clauses (**if..then**, **elseif..then**, **else** and **end if**) must not cross a line boundary.

Arguments:

Arguments

Argument	Description
condition	A logical expression which can be evaluated as True or False.
statements	Any group of one or more Qlik Sense script statements.

Example 1:

```
if a=1 then
```

```
      LOAD * from abc.csv;
      SQL SELECT e, f, g from tab1;
end if
```

Example 2:

```
if a=1 then; drop table xyz; end if;
```

Example 3:

```
if x>0 then
      LOAD * from pos.csv;
elseif x<0 then
      LOAD * from neg.csv;
else
      LOAD * from zero.txt;
end if
```

Next

The **Next** script keyword is used to close **For** loops.

Sub..end sub

The **sub..end sub** control statement defines a subroutine which can be called upon from a **call** statement.

Syntax:

```
Sub name [ ( paramlist )] statements end sub
```

Arguments are copied into the subroutine and, if the corresponding actual parameter in the **call** statement is a variable name, copied back out again upon exiting the subroutine.

If a subroutine has more formal parameters than actual parameters passed by a **call** statement, the extra parameters will be initialized to NULL and can be used as local variables within the subroutine.

Arguments:

Arguments

Argument	Description
name	The name of the subroutine.
paramlist	A comma separated list of variable names for the formal parameters of the subroutine. These can be used as any variable inside the subroutine.
statements	Any group of one or more Qlik Sense script statements.

Limitations:

- Since the **sub** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its two clauses (**sub** and **end sub**) must not cross a line boundary.
- When you define a subroutine with `Sub..end sub` inside a control statement, for example `if..then`, you can only call the subroutine from within the same control statement.

Example 1:

```
Sub INCR (I,J)
I = I + 1
Exit Sub when I < 10
J = J + 1
End Sub
Call INCR (X,Y)
```

Example 2: - parameter transfer

```
Sub ParTrans (A,B,C)
A=A+1
B=B+1
C=C+1
End Sub
A=1
X=1
C=1
Call ParTrans (A, (X+1)*2)
```

The result of the above will be that locally, inside the subroutine, A will be initialized to 1, B will be initialized to 4 and C will be initialized to NULL.

When exiting the subroutine, the global variable A will get 2 as value (copied back from subroutine). The second actual parameter "(X+1)*2" will not be copied back since it is not a variable. Finally, the global variable C will not be affected by the subroutine call.

Switch..case..default..end switch

The **switch** control statement is a script selection construct forcing the script execution to follow different paths, depending on the value of an expression.

Syntax:

```
Switch expression {case valuelist [ statements ]} [default statements] end switch
```

*Since the **switch** statement is a control statement and as such is ended with either a semicolon or end-of-line, each of its four possible clauses (**switch**, **case**, **default** and **end switch**) must not cross a line boundary.*

Arguments:

Arguments

Argument	Description
expression	An arbitrary expression.
valuelist	A comma separated list of values with which the value of expression will be compared. Execution of the script will continue with the statements in the first group encountered with a value in valuelist equal to the value in expression. Each value in valuelist may be an arbitrary expression. If no match is found in any **case** clause, the statements under the **default** clause, if specified, will be executed.
statements	Any group of one or more Qlik Sense script statements.

Example:

```
Switch I
Case 1
LOAD '$(I): CASE 1' as case autogenerate 1;
Case 2
LOAD '$(I): CASE 2' as case autogenerate 1;
Default
LOAD '$(I): DEFAULT' as case autogenerate 1;
End Switch
```

To

The **To** script keyword is used in several script statements.

6.5 Prefixes

Prefixes may be applied to applicable regular statements but never to control statements.

All script keywords can be typed with any combination of lower case and upper case characters. Field and variable names used in the statements are however case sensitive.

Chart modifier prefixes overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

Add

The **Add** prefix can be added to any **LOAD** or **SELECT** statement in the script to specify that it should add records to another table. It also specifies that this statement should be run in a partial reload. The **Add** prefix can also be used in a **Map** statement.

```
Add [only] [Concatenate[(tablename )]] (loadstatement | selectstatement)
Add [ Only ] mapstatement
```

Replace

The **Replace** prefix can be added to any **LOAD** or **SELECT** statement in the script to specify that the loaded table should replace another table. It also specifies that this statement should be run in a partial reload. The **Replace** prefix can also be used in a **Map** statement.

```
Replace [only] [Concatenate[(tablename) ]] (loadstatement | selectstatement)
Replace [only] mapstatement
```

Add

In a chart modifying context, the **Add** prefix is used with **LOAD** to append values to the *HC1* table, representing the hypercube computed by the Qlik associative engine. You can specify one or several columns. Missing values are automatically filled by the Qlik associative engine.

Syntax:

```
Add loadstatement
```

Example:

This example adds two rows to the columns *Dates* and *Sales* from the inline statement

```
Add Load
x as Dates,
y as Sales
Inline
[
Dates,Sales
2001/09/1,1000
2001/09/10,-300
]
```

Replace

In a chart modifying context, the **Replace** prefix changes all values of the *HC1* table with a computed value defined by the script.

Syntax:

```
Replace loadstatement
```

Example:

This example overwrites all values in column `z` with the sum of `x` and `y`.

```
Replace Load
x+y as z
Resident HC1;
```

6.6 Regular statements

Regular statements are typically used for manipulating data in one way or another. These statements may be written over any number of lines in the script and must always be terminated by a semicolon, ";".

All script keywords can be typed with any combination of lower case and upper case characters. Field and variable names used in the statements are however case sensitive.

Chart modifier regular statements overview

Each function is described further after the overview. You can also click the function name in the syntax to immediately access the details for that specific function.

LOAD

The **LOAD** statement loads fields from a file, from data defined in the script, from a previously loaded table, from a web page, from the result of a subsequent **SELECT** statement or by generating data automatically. It is also possible to load data from analytic connections.

```
Load [ distinct ] *fieldlist
[( from file [ format-spec ] |
from_field fieldassource [format-spec]
inline data [ format-spec ] |
resident table-label |
autogenerate size )]
[ where criterion | while criterion ]
[ group_by groupbyfieldlist ]
[order_by orderbyfieldlist ]
[extension pluginname.functionname(tabledescription)]
```

Let

The **let** statement is a complement to the **set** statement, used for defining script variables. The **let** statement, in opposition to the **set** statement, evaluates the expression on the right side of the ' =' at script run time before it is assigned to the variable.

```
Let variablename=expression
```

Set

The **set** statement is used for defining script variables. These can be used for substituting strings, paths, drives, and so on.

```
Set variablename=string
```

Put

The **Put** statement is used in...

HCValue

The **HCValue** statement is used in...

Load

In a chart modifying context, the **LOAD** statement loads additional data to the hypercube from data defined in the script, or from a previously loaded table. It is also possible to load data from analytic connections.

*The **LOAD** statement must have either **Replace** or **Add** prefix, or it will be rejected.*

Syntax:

```
Add | Replace LOAD fieldlist
(
inline data [ format-spec ] |
resident table-label
) | extension pluginname.functionname([script] tabledescription)]
[ where criterion | while criterion ]
[ group by groupbyfieldlist ]
[order by orderbyfieldlist ]
```

Arguments:

Arguments

Argument	Description
fieldlist	*fieldlist ::= (* | field {,* | field })* A list of the fields to be loaded. Using * as a field list indicates all fields in the table. *field ::= (fieldref | expression) [**as** aliasname]* The field definition must always contain a literal, a reference to an existing field, or an expression. *fieldref ::= (fieldname |@fieldnumber |@startpos:endpos [**I** | **U** | **R** | **B** | **T**])* *fieldname* is a text that is identical to a field name in the table. Note that the field name must be enclosed by straight double quotation marks or square brackets if it contains e.g. spaces. Sometimes field names are not explicitly available. Then a different notation is used: *@fieldnumber* represents the field number in a delimited table file. It must be a positive integer preceded by "@". The numbering is always made from 1 and up to the number of fields. *@startpos*:*endpos* represents the start and end positions of a field in a file with fixed length records. The positions must both be positive integers. The two numbers must be preceded by "@" and separated by a colon. The numbering is always made from 1 and up to the number of positions. In the last field, **n** is used as end position. • If *@startpos*:*endpos* is immediately followed by the characters **I** or **U**, the bytes read will be interpreted as a binary signed (**I**) or unsigned (**U**) integer (Intel byte order). The number of positions read must be 1, 2 or 4. • If *@startpos*:*endpos* is immediately followed by the character **R**, the bytes read will be interpreted as a binary real number (IEEE 32-bit or 64 bit floating point). The number of positions read must be 4 or 8. • If *@startpos*:*endpos* is immediately followed by the character **B**, the bytes read will be interpreted as a BCD (Binary Coded Decimal) numbers according to the COMP-3 standard. Any number of bytes may be specified. *expression* can be a numeric function or a string function based on one or several other fields in the same table. For further information, see the syntax of expressions. **as** is used for assigning a new name to the field.

Argument	Description
inline	**inline** is used if data should be typed within the script, and not loaded from a file. *data ::= [text]* Data entered through an **inline** clause must be enclosed by double quotation marks or by square brackets. The text between these is interpreted in the same way as the content of a file. Hence, where you would insert a new line in a text file, you should also do it in the text of an **inline** clause, i.e. by pressing the Enter key when typing the script. The number of columns are defined by the first line. *format-spec ::= (fspec-item {, fspec-item })* The format specification consists of a list of several format specification items, within brackets.
resident	**resident** is used if data should be loaded from a previously loaded table. *table label* is a label preceding the **LOAD** statement that created the original table. The label should be given with a colon at the end.
extension	You can load data from analytic connections. You need to use the **extension** clause to call a function defined in the server-side extension (SSE) plugin, or evaluate a script. You can send a single table to the SSE plugin, and a single data table is returned. If the plugin does not specify the names of the fields that are returned, the fields will be named Field1, Field2, and so on. `Extension pluginname.functionname( tabledescription );` • Loading data using a function in an SSE plugin *tabledescription ::= (table { ,tablefield})* If you do not state table fields, the fields will be used in load order. • Loading data by evaluating a script in an SSE plugin *tabledescription ::= (script, table { ,tablefield})* **Data type handling in the table field definition** Data types are automatically detected in analytic connections. If the data has no numeric values and at least one non-NULL text string, the field is considered as text. In any other case it is considered as numeric. You can force the data type by wrapping a field name with **String()** or **Mixed()**. • **String()** forces the field to be text. If the field is numeric, the text part of the dual value is extracted, there is no conversion performed. • **Mixed()** forces the field to be dual. **String()** or **Mixed()** cannot be used outside **extension** table field definitions, and you cannot use other Qlik Sense functions in a table field definition.

Argument	Description
where	**where** is a clause used for stating whether a record should be included in the selection or not. The selection is included if *criterion* is True. *criterion* is a logical expression.
while	**while** is a clause used for stating whether a record should be repeatedly read. The same record is read as long as *criterion* is True. In order to be useful, a **while** clause must typically include the **IterNo( )** function. *criterion* is a logical expression.
group by	**group by** is a clause used for defining over which fields the data should be aggregated (grouped). The aggregation fields should be included in some way in the expressions loaded. No other fields than the aggregation fields may be used outside aggregation functions in the loaded expressions. *groupbyfieldlist ::= (fieldname { ,fieldname })*
order by	**order by** is a clause used for sorting the records of a resident table before they are processed by the **load** statement. The resident table can be sorted by one or more fields in ascending or descending order. The sorting is made primarily by numeric value and secondarily by national collation order. This clause may only be used when the data source is a resident table. The ordering fields specify which field the resident table is sorted by. The field can be specified by its name or by its number in the resident table (the first field is number 1). orderbyfieldlist ::= fieldname [sortorder] { , fieldname [sortorder] } *sortorder* is either *asc* for ascending or *desc* for descending. If no *sortorder* is specified, *asc* is assumed. *fieldname*, *path*, *filename* and *aliasname* are text strings representing what the respective names imply. Any field in the source table can be used as *fieldname*. However, fields created through the as clause (*aliasname*) are out of scope and cannot be used inside the same **load** statement.

Let

The **let** statement is a complement to the **set** statement, used for defining script variables. The **let** statement, in opposition to the **set** statement, evaluates the expression on the right side of the ' =' at script run time before it is assigned to the variable.

Syntax:

```
Let variablename=expression
```

Examples and results:

Example	Result

`Set x=3+4;`
`Let y=3+4;`
`z=$(y)+1;`
	`$(x)` will be evaluated as ' 3+4 '
	
	`$(y)` will be evaluated as ' 7 '
	
	`$(z)` will be evaluated as ' 8 '
	
	Note the difference between the **Set** and **Let** statements. The **Set** statement assigns the string '3+4' to the variable, whereas the **Let** statement evaluates the string and assigns 7 to the variable.

Let T=now( );	`$(T)` will be given the value of the current time.

Set

The **set** statement is used for defining script variables. These can be used for substituting strings, paths, drives, and so on.

Syntax:

```
Set variablename=string
```

Example 1:

```
Set FileToUse=Data1.csv;
```

Example 2:

```
Set Constant="My string";
```

Example 3:

```
Set BudgetYear=2012;
```

Put

The **put** statement is used to set some numeric value in the hypercube.

Access to the columns can be done by labels. You can also access columns and rows by declaration order. See the examples below for more details.

Syntax:

```
put column(position)=value
```

Example 1:

Access to the columns can be done by labels.

This example will set a value of `1` in the first position of the column labeled *Sales*.

```
Put Sales(1) = 1;
```

Example 2:

You can access measure columns by declaration order using the `#hc1.measure` format for measures.

This example will set the value 1000 in the tenth position of the final sorted hypercube.

```
Put #hc1.measure.2(10) = 1000;
```

Example 3:

You can access the dimension rows by declaration order using the `#hc1.dimension` format for dimensions.

This example puts the value of the constant Pi in the fifth row of the third declared dimension.

```
Put #hc1.dimension.3(5) = Pi();
```

If there are no such dimensions or expressions, in value or labels, an error is returned indicating that the column was not found. If the index for the column is out of bounds, no error is displayed.

HCValue

The **HCValue** function it is used to retrieve values in a row of a specified column.

Syntax:

```
HCValue(column,position)
```

Example 1:

This example returns the value at the first position of the column with label 'Sales'.

```
HCValue(Sales,1)
```

Example 2:

This example returns the value at the tenth position of the sorted hypercube.

```
HCValue(#hc1.measure.2,10)
```

Example 3:

This example returns the value at the fifth row in the third dimension.

```
HCValue(#hc1.dimension.3,5)
```

If there are no such dimensions or expressions, in value or labels, an error is returned indicating that the column was not found. If the index for the column is out of bounds, NULL is returned.

7 QlikView functions and statements not supported in Qlik Sense

Most functions and statements that can be used in QlikView load scripts and chart expressions are also supported in Qlik Sense, but there are some exceptions, as described here.

7.1 Script statements not supported in Qlik Sense

QlikView script statements that are not supported in Qlik Sense

Statement	Comments
Command	Use **SQL** instead.
InputField	

7.2 Functions not supported in Qlik Sense

This list describes QlikView script and chart functions that are not supported in Qlik Sense.

- **GetCurrentField**
- **GetExtendedProperty**
- **Input**
- **InputAvg**
- **InputSum**
- **MsgBox**
- **NoOfReports**
- **ReportComment**
- **ReportId**
- **ReportName**
- **ReportNumber**

7.3 Prefixes not supported in Qlik Sense

This list describes QlikView prefixes that are not supported in Qlik Sense.

- **Bundle**
- **Image_Size**
- **Info**

8 Functions and statements not recommended in Qlik Sense

Most functions and statements that can be used in QlikView load scripts and chart expressions are also supported in Qlik Sense, but some of them are not recommended for use in Qlik Sense. There are also functions and statements available in previous versions of Qlik Sense that have been deprecated.

For compatibility reasons they will still work as intended, but it is advisable to update the code according to the recommendations in this section, as they may be removed in coming versions.

8.1 Script statements not recommended in Qlik Sense

This table contains script statements that are not recommended for use in Qlik Sense.

Script statements that are not recommended

Statement	Recommendation
Command	Use **SQL** instead.
CustomConnect	Use **Custom Connect** instead.

8.2 Script statement parameters not recommended in Qlik Sense

This table describes script statement parameters that are not recommended for use in Qlik Sense.

Script statement parameters that are not recommended

Statement	Parameters
Buffer	Use **Incremental** instead of: • **Inc** (not recommended) • **Incr** (not recommended)

Statement	Parameters
LOAD	The following parameter keywords are generated by QlikView file transformation wizards. Functionality is retained when data is reloaded, but Qlik Sense does not provide guided support/wizards for generating the statement with these parameters: • **Bottom** • **Cellvalue** • **Col** • **Colmatch** • **Colsplit** • **Colxtr** • **Compound** • **Contain** • **Equal** • **Every** • **Expand** • **Filters** • **Intarray** • **Interpret** • **Length** • **Longer** • **Numerical** • **Pos** • **Remove** • **Rotate** • **Row** • **Rowcnd** • **Shorter** • **Start** • **Strcnd** • **Top** • **Transpose** • **Unwrap** • **XML: XMLSAX and Pattern is Path**

8.3 Functions not recommended in Qlik Sense

This table describes script and chart functions that are not recommended for use in Qlik Sense.

Functions that are not recommended

Function	Recommendation
NumAvg **NumCount** **NumMax** **NumMin** **NumSum**	Use Range functions instead. *Range functions (page 901)*
Color() **QliktechBlue** **QliktechGray**	Use other color functions instead. **QliktechBlue()** can be replaced by **RGB(8, 18, 90)** and **QliktechGray** can be replaced by **RGB(158, 148, 137)** to get the same colors. *Color functions (page 465)*
QlikViewVersion	Use **EngineVersion** instead. *EngineVersion (page 1029)*
ProductVersion	Use **EngineVersion** instead. *EngineVersion (page 1029)*
QVUser	
Year2Date	Use **YearToDate** instead.
Vrank	Use **Rank** instead.
WildMatch5	Use **WildMatch** instead.

ALL qualifier

In QlikView, the **ALL** qualifier may occur before an expression. This is equivalent to using **{1} TOTAL**. In such a case the calculation will be made over all the values of the field in the document, disregarding the chart dimensions and current selections. The same value is always returned regardless of the logical state in the document. If the **ALL** qualifier is used, a set expression cannot be used, since the **ALL** qualifier defines a set by itself. For legacy reasons, the **ALL** qualifier will still work in this version of Qlik Sense, but may be removed in coming versions.