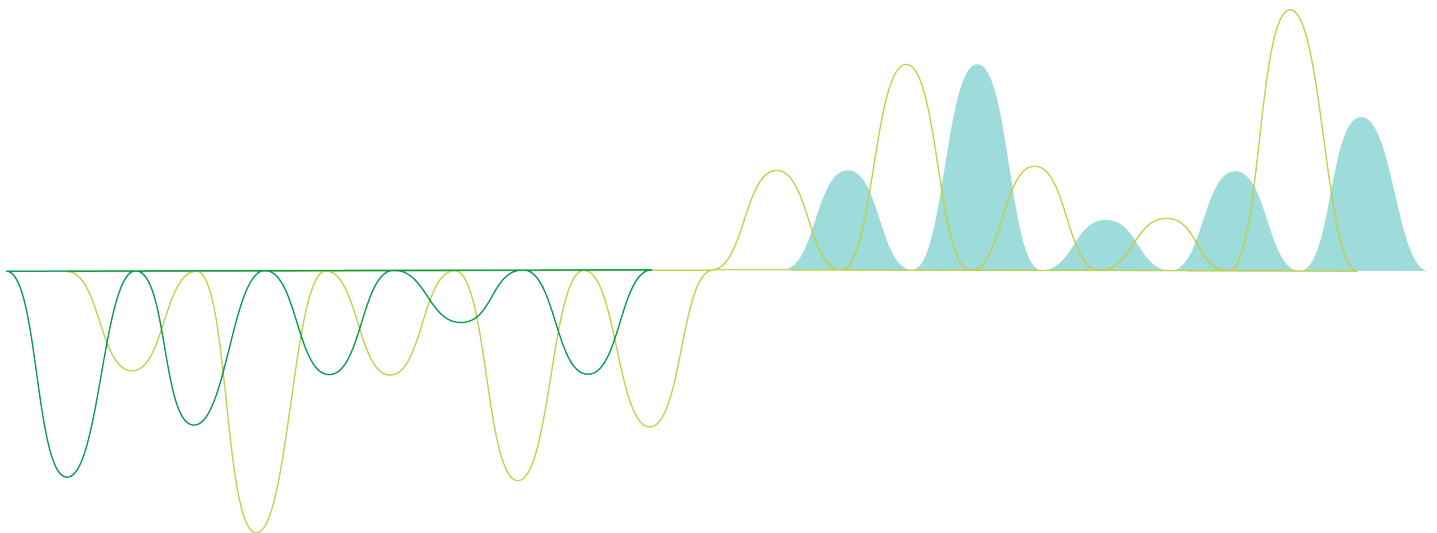


# Script syntax and chart functions

Qlik Sense®

May 2021

Copyright © 1993-2021 QlikTech International AB. All rights reserved.





---

<b>1 What is Qlik Sense?</b>	<b>15</b>
1.1 What can you do in Qlik Sense?	15
1.2 How does Qlik Sense work?	15
The app model	15
The associative experience	15
Collaboration and mobility	15
1.3 How can you deploy Qlik Sense?	15
Qlik Sense Desktop	16
Qlik Sense Enterprise	16
1.4 How to administer and manage a Qlik Sense site	16
1.5 Extend Qlik Sense and adapt it for your own purposes	16
Building extensions and mashups	16
Building clients	16
Building server tools	16
Connecting to other data sources	16
<b>2 Script syntax</b>	<b>17</b>
2.1 Introduction to script syntax	17
2.2 What is Backus-Naur formalism?	17
<b>2 Script statements and keywords</b>	<b>19</b>
2.3 Script control statements	19
Script control statements overview	19
Call	21
Do..loop	22
Exit script	23
For..next	24
For each..next	25
If..then..elseif..else..end if	28
Sub..end sub	29
Switch..case..default..end switch	30
2.4 Script prefixes	31
Script prefixes overview	31
Add	35
Buffer	36
Concatenate	38
Crosstable	38
First	39
Generic	40
Hierarchy	40
HierarchyBelongsTo	42
Inner	43
IntervalMatch	45
Join	48
Keep	49
Left	50

---

---

Mapping .....	51
Merge .....	53
NoConcatenate .....	56
Outer .....	56
Partial reload .....	57
Replace .....	58
Right .....	59
Sample .....	61
Semantic .....	61
Unless .....	62
When .....	62
2.5 Script regular statements .....	63
Script regular statements overview .....	63
Alias .....	70
AutoNumber .....	70
Binary .....	72
Comment field .....	74
Comment table .....	74
Connect .....	75
Declare .....	77
Derive .....	79
Direct Query .....	80
Directory .....	85
Disconnect .....	86
Drop field .....	87
Drop table .....	87
Execute .....	88
Field/Fields .....	89
FlushLog .....	89
Force .....	89
Load .....	91
Let .....	108
Loosen Table .....	108
Map .....	109
NullAsNull .....	110
NullAsValue .....	110
Qualify .....	111
Rem .....	112
Rename field .....	113
Rename table .....	114
Search .....	114
Section .....	115
Select .....	116
Set .....	118
Sleep .....	119
SQL .....	119

---

---

SQLColumns .....	120
SQLTables .....	121
SQLTypes .....	121
Star .....	122
Store .....	124
Tag .....	125
Trace .....	126
Unmap .....	127
Unqualify .....	127
Untag .....	128
2.6 Working directory .....	129
Qlik Sense Desktop working directory .....	129
Qlik Sense working directory .....	129
<b>2 Working with variables in the data load editor .....</b>	<b>130</b>
2.7 Overview .....	130
2.8 Defining a variable .....	130
2.9 Deleting a variable .....	131
2.10 Loading a variable value as a field value .....	131
2.11 Variable calculation .....	131
2.12 System variables .....	132
System variables overview .....	132
CreateSearchIndexOnReload .....	135
HidePrefix .....	135
HideSuffix .....	136
Include .....	136
OpenUrlTimeout .....	137
StripComments .....	137
Verbatim .....	138
2.13 Value handling variables .....	138
Value handling variables overview .....	138
NullDisplay .....	139
NullInterpret .....	139
NullValue .....	139
OtherSymbol .....	140
2.14 Number interpretation variables .....	140
Number interpretation variables overview .....	140
BrokenWeeks .....	143
DateFormat .....	143
DayNames .....	143
DecimalSep .....	144
FirstWeekDay .....	144
LongDayNames .....	144
LongMonthNames .....	145
MoneyDecimalSep .....	145
MoneyFormat .....	145

---

---

MoneyThousandSep .....	145
MonthNames .....	146
NumericalAbbreviation .....	146
ReferenceDay .....	146
ThousandSep .....	147
TimeFormat .....	147
TimestampFormat .....	147
2.15 Direct Discovery variables .....	150
Direct Discovery system variables .....	150
Teradata query banding variables .....	152
Direct Discovery character variables .....	152
Direct Discovery number interpretation variables .....	153
2.16 Error variables .....	154
Error variables overview .....	154
ErrorMode .....	155
ScriptError .....	155
ScriptErrorCount .....	156
ScriptErrorList .....	157
<b>2 Script expressions .....</b>	<b>158</b>
<b>3 Visualization expressions .....</b>	<b>159</b>
3.1 Defining the aggregation scope .....	159
3.2 Syntax for sets .....	161
3.3 Set modifiers .....	162
Based on another field .....	162
Based on element sets .....	162
Forced exclusion .....	163
Set modifiers with set operators .....	164
Set modifiers using assignments with implicit set operators .....	165
Set modifiers with advanced searches .....	165
Set modifiers with dollar-sign expansions .....	165
Set modifiers with implicit field value definitions .....	166
3.4 Visualization expression and aggregation syntax .....	167
General syntax for chart expressions .....	167
General syntax for aggregations .....	167
<b>4 Operators .....</b>	<b>169</b>
4.1 Bit operators .....	169
4.2 Logical operators .....	170
4.3 Numeric operators .....	170
4.4 Relational operators .....	171
4.5 String operators .....	172
& .....	173
like .....	173
<b>5 Functions in scripts and chart expressions .....</b>	<b>174</b>
5.1 Analytic connections for server-side extensions (SSE) .....	174

---

---

5.2 Aggregation functions .....	174
Using aggregation functions in a data load script .....	174
Using aggregation functions in chart expressions .....	174
Basic aggregation functions .....	175
Counter aggregation functions .....	197
Financial aggregation functions .....	214
Statistical aggregation functions .....	226
Statistical test functions .....	288
String aggregation functions .....	351
Synthetic dimension functions .....	363
Nested aggregations .....	366
5.3 Aggr - chart function .....	367
5.4 Color functions .....	372
Pre-defined color functions .....	374
ARGB .....	375
RGB .....	375
HSL .....	376
5.5 Conditional functions .....	377
Conditional functions overview .....	377
alt .....	378
class .....	379
coalesce .....	380
if .....	381
match .....	383
mixmatch .....	386
pick .....	389
wildmatch .....	389
5.6 Counter functions .....	392
Counter functions overview .....	392
autonumber .....	393
autonumberhash128 .....	396
autonumberhash256 .....	398
IterNo .....	400
RecNo .....	401
RowNo .....	402
RowNo - chart function .....	403
5.7 Date and time functions .....	405
Date and time functions overview .....	405
addmonths .....	414
addyears .....	415
age .....	415
converttolocaltime .....	417
day .....	420
dayend .....	420
daylightsaving .....	422

---

---

dayname .....	422
daynumberofquarter .....	424
daynumberofyear .....	426
daystart .....	427
firstworkdate .....	429
GMT .....	431
hour .....	431
inday .....	432
indaytotime .....	434
inlunarweek .....	436
inlunarweektodate .....	438
inmonth .....	440
inmonths .....	442
inmonthstodate .....	444
inmonthtodate .....	447
inquarter .....	448
inquartertodate .....	450
inweek .....	452
inweektodate .....	454
inyear .....	456
inyeartodate .....	459
lastworkdate .....	461
localtime .....	462
lunarweekend .....	463
lunarweekname .....	465
lunarweekstart .....	467
makedate .....	469
maketime .....	471
makeweekdate .....	472
minute .....	472
month .....	473
monthend .....	473
monthname .....	475
monthsend .....	478
monthsname .....	480
monthsstart .....	482
monthstart .....	484
networkdays .....	486
now .....	488
quarterend .....	489
quartername .....	491
quarterstart .....	493
second .....	495
setdateyear .....	495
setdateyearmonth .....	497
timezone .....	499

---



---

today .....	499
UTC .....	500
week .....	500
weekday .....	502
weekend .....	505
weekname .....	507
weekstart .....	509
weekyear .....	511
year .....	512
yearend .....	512
yearname .....	514
yearstart .....	516
yeartodate .....	518
5.8 Exponential and logarithmic functions .....	519
5.9 Field functions .....	521
Count functions .....	521
Field and selection functions .....	521
GetAlternativeCount - chart function .....	522
GetCurrentSelections - chart function .....	523
GetExcludedCount - chart function .....	525
GetFieldSelections - chart function .....	526
GetNotSelectedCount - chart function .....	527
GetObjectDimension - chart function .....	528
GetObjectField - chart function .....	529
GetObjectMeasure - chart function .....	529
GetPossibleCount - chart function .....	530
GetSelectedCount - chart function .....	532
5.10 File functions .....	533
File functions overview .....	533
Attribute .....	535
ConnectionString .....	542
FileName .....	542
FileDir .....	543
FileExtension .....	543
FileName .....	543
FilePath .....	544
FileSize .....	544
FileTime .....	545
GetFolderPath .....	546
QvdCreateTime .....	547
QvdFieldName .....	548
QvdNoOfFields .....	549
QvdNoOfRecords .....	550
QvdTableName .....	551
5.11 Financial functions .....	552

---

---

Financial functions overview .....	553
BlackAndSchole .....	553
FV .....	554
nPer .....	555
Pmt .....	556
PV .....	557
Rate .....	558
5.12 Formatting functions .....	559
Formatting functions overview .....	559
ApplyCodepage .....	560
Date .....	561
Dual .....	563
Interval .....	564
Money .....	565
Num .....	567
Time .....	570
Timestamp .....	571
5.13 General numeric functions .....	572
General numeric functions overview .....	572
Combination and permutation functions .....	573
Modulo functions .....	573
Parity functions .....	574
Rounding functions .....	574
BitCount .....	574
Ceil .....	575
Combin .....	576
Div .....	577
Even .....	577
Fabs .....	577
Fact .....	578
Floor .....	578
Fmod .....	579
Frac .....	580
Mod .....	581
Odd .....	582
Permut .....	582
Round .....	583
Sign .....	584
5.14 Geospatial functions .....	584
Geospatial functions overview .....	585
GeoAggrGeometry .....	586
GeoBoundingBox .....	587
GeoCountVertex .....	588
GeoGetBoundingBox .....	588
GeoGetPolygonCenter .....	589

---

---

GeoInvProjectGeometry .....	590
GeoMakePoint .....	590
GeoProject .....	591
GeoProjectGeometry .....	592
GeoReduceGeometry .....	592
5.15 Interpretation functions .....	593
Interpretation functions overview .....	594
Date# .....	595
Interval# .....	596
Money# .....	597
Num# .....	598
Text .....	599
Time# .....	599
Timestamp# .....	600
5.16 Inter-record functions .....	601
Row functions .....	602
Column functions .....	603
Field functions .....	603
Pivot table functions .....	603
Inter-record functions in the data load script .....	604
Above - chart function .....	605
Below - chart function .....	610
Bottom - chart function .....	613
Column - chart function .....	617
Dimensionality - chart function .....	619
Exists .....	619
FieldIndex .....	623
FieldValue .....	625
FieldValueCount .....	627
LookUp .....	628
NoOfRows - chart function .....	630
Peek .....	631
Previous .....	634
Top - chart function .....	635
SecondaryDimensionality - chart function .....	639
After - chart function .....	639
Before - chart function .....	640
First - chart function .....	641
Last - chart function .....	642
ColumnNo - chart function .....	643
NoOfColumns - chart function .....	644
5.17 Logical functions .....	644
5.18 Mapping functions .....	645
Mapping functions overview .....	645
ApplyMap .....	646

---

---

MapSubstring .....	647
5.19 Mathematical functions .....	649
5.20 NULL functions .....	650
NULL functions overview .....	650
EmptyIsNull .....	650
IsNull .....	651
NULL .....	652
5.21 Range functions .....	653
Basic range functions .....	653
Counter range functions .....	654
Statistical range functions .....	654
Financial range functions .....	655
RangeAvg .....	656
RangeCorrel .....	658
RangeCount .....	660
RangeFractile .....	662
RangeIRR .....	664
RangeKurtosis .....	665
RangeMax .....	666
RangeMaxString .....	668
RangeMin .....	670
RangeMinString .....	672
RangeMissingCount .....	673
RangeMode .....	675
RangeNPV .....	677
RangeNullCount .....	678
RangeNumericCount .....	679
RangeOnly .....	681
RangeSkew .....	681
RangeStdev .....	683
RangeSum .....	684
RangeTextCount .....	687
RangeXIRR .....	688
RangeXNPV .....	689
5.22 Ranking and clustering functions .....	690
Ranking functions in charts .....	690
Clustering functions in charts .....	691
Rank - chart function .....	692
HRank - chart function .....	695
KMeans2D - chart function .....	697
KMeansND - chart function .....	710
KMeansCentroid2D - chart function .....	723
KMeansCentroidND - chart function .....	724
5.23 Statistical distribution functions .....	725
Statistical distribution functions overview .....	725

---

---

CHIDIST .....	727
CHIINV .....	727
FDIST .....	728
FINV .....	729
NORMDIST .....	730
NORMINV .....	730
TDIST .....	731
TINV .....	732
5.24 String functions .....	733
String functions overview .....	733
Capitalize .....	736
Chr .....	736
Evaluate .....	737
FindOneOf .....	737
Hash128 .....	738
Hash160 .....	738
Hash256 .....	739
Index .....	739
KeepChar .....	740
Left .....	741
Len .....	741
LevenshteinDist .....	741
Lower .....	742
LTrim .....	742
Mid .....	743
Ord .....	743
PurgeChar .....	744
Repeat .....	744
Replace .....	745
Right .....	746
RTrim .....	746
SubField .....	747
SubStringCount .....	749
TextBetween .....	750
Trim .....	750
Upper .....	751
5.25 System functions .....	751
System functions overview .....	751
EngineVersion .....	754
IsPartialReload .....	754
ProductVersion .....	754
StateName - chart function .....	754
5.26 Table functions .....	755
Table functions overview .....	755
FieldName .....	757

---

---

FieldNumber .....	757
NoOfFields .....	758
NoOfRows .....	758
5.27 Trigonometric and hyperbolic functions .....	759
<b>6 File system access restriction .....</b>	<b>762</b>
6.1 Security aspects when connecting to file based ODBC and OLE DB data connections ..	762
6.2 Limitations in standard mode .....	762
System variables .....	762
Regular script statements .....	764
Script control statements .....	765
File functions .....	765
System functions .....	767
6.3 Disabling standard mode .....	767
Qlik Sense .....	767
Qlik Sense Desktop .....	767
<b>7 QlikView functions and statements not supported in Qlik Sense .....</b>	<b>769</b>
7.1 Script statements not supported in Qlik Sense .....	769
7.2 Functions not supported in Qlik Sense .....	769
7.3 Prefixes not supported in Qlik Sense .....	769
<b>8 Functions and statements not recommended in Qlik Sense .....</b>	<b>770</b>
8.1 Script statements not recommended in Qlik Sense .....	770
8.2 Script statement parameters not recommended in Qlik Sense .....	770
8.3 Functions not recommended in Qlik Sense .....	771
ALL qualifier .....	772

# 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

### 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 DB database 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.

Symbol	Description
::=	Marks the beginning of a block that defines a symbol.
<b>LOAD</b>	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.

## 2 Script statements and keywords

```
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.

## 2 Script statements and keywords

Argument	Description
statements	Any group of one or more Qlik Sense script statements.
while / until	The <b>while</b> or <b>until</b> conditional clause must only appear once in any <b>do..loop</b> statement, i.e. either after <b>do</b> or after <b>loop</b> . 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 <b>exit do</b> clause is encountered inside the loop, the execution of the script will be transferred to the first statement after the <b>loop</b> clause denoting the end of the loop. An <b>exit do</b> clause can be made conditional by the optional use of a <b>when</b> or <b>unless</b> 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
```

## 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 <b>exit script</b> statement can be made conditional by the optional use of <b>when</b> or <b>unless</b> 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 <i>counter</i> is specified after <b>next</b> it must be the same variable name as the one found after the corresponding <b>for</b> .
expr1	An expression which determines the first value of the <i>counter</i> variable for which the loop should be executed.
expr2	An expression which determines the last value of the <i>counter</i> variable for which the loop should be executed.
expr3	An expression which determines the value indicating the increment of the <i>counter</i> 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 <b>var</b> is specified after <b>next</b> it must be the same variable name as the one found after the corresponding <b>for each</b> .

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

## 2 Script statements and keywords

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. <div> <i>This argument supports only library connections in standard mode.</i></div>
dirlist mask	This syntax produces a comma separated list of all folders in the current folder matching the folder name mask. <div> <i>This argument supports only library connections in standard mode.</i></div>
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')
```

## 2 Script statements and keywords

```
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
```

## 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 <b>case</b> clause, the statements under the <b>default</b> 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
```

## 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
```

---

## 2 Script statements and keywords

---

### 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** prefix is used to turn a cross table into a straight table, that is, a wide table with many columns is turned into a tall table, with the column headings being placed into a single attribute column.

```
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 unpacking and loading of a generic database can be done with a **generic** prefix.

```
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)
```



---

## 2 Script statements and keywords

---

### 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** in order 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 explicit **Join** prefix can be preceded by the prefix **Outer** in order to specify an outer join. In an outer join, the resulting table will contain all values from both raw tables where the linking field values are represented in either 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 )
```

### 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.

## 2 Script statements and keywords

```
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.



*Partial reload is supported by using the Qlik Engine JSON API, or the **Reload** button.*

### 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.

## 2 Script statements and keywords

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;	<p>During normal reload, data is loaded from <i>Persons.csv</i> and stored in the Qlik Sense table Tab1. Data from <i>NewPersons.csv</i> is then concatenated to the same Qlik Sense table.</p> <p>During partial reload, data is loaded from <i>NewPersons.csv</i> and appended to the Qlik Sense table Tab1. No check for duplicates is made.</p>
Tab1: SQL SELECT Name, Number FROM Persons.csv; Add LOAD Name, Number FROM NewPersons.csv where not exists(Name);	<p>A check for duplicates is made by means of looking if Name exists in the previously loaded table data.</p> <p>During normal reload, data is loaded from <i>Persons.csv</i> and stored in the Qlik Sense table Tab1. Data from <i>NewPersons.csv</i> is then concatenated to the same Qlik Sense table.</p> <p>During partial reload, data is loaded from <i>NewPersons.csv</i> 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.</p>
Tab1: LOAD Name, Number FROM Persons.csv; Add Only LOAD Name, Number FROM NewPersons.csv where not exists(Name);	<p>During normal reload, data is loaded from <i>Persons.csv</i> and stored in the Qlik Sense table Tab1. The statement loading <i>NewPersons.csv</i> is disregarded.</p> <p>During partial reload, data is loaded from <i>NewPersons.csv</i> 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.</p>

## 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)]
```

---

## 2 Script statements and keywords

---

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. Note that the incremental option can only be used with <b>LOAD</b> statements and text files and that 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.

### Syntax:

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

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** prefix is used to turn a cross table into a straight table, that is, a wide table with many columns is turned into a tall table, with the column headings being placed into a single attribute column.

### Syntax:

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

### Arguments:

Arguments

Argument	Description
attribute field name	The field that contains the attribute values.
data field name	The field that contains the data values.
n	The number of qualifier fields preceding the table to be transformed to generic form. Default is 1.

A crosstable is a common type of table featuring a matrix of values between two or more orthogonal lists of header data, of which one is used as column headers. A typical example could be to have one column per month. The result of the **crosstable** prefix is that the column headers (for example month names) will be stored in one field, the attribute field, and the column data (month numbers) will be stored in a second field: the data field.

### Examples:

```
Crosstable (Month, Sales) LOAD * from ex1.csv;  
Crosstable (Month,Sales,2) LOAD * from ex2.csv;  
Crosstable (A,B) SELECT * from table3;
```

## 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.

### Syntax:

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

### Arguments:

Arguments

Argument	Description
n	An arbitrary expression that evaluates to an integer indicating the maximum number of records to be read.  <i>n</i> can be enclosed in parentheses, like ( <i>n</i> ), but this is not required.

### Examples:

First 10 LOAD \* from abc.csv;  
First (1) SQL SELECT \* from Orders;

## Generic

The unpacking and loading of a generic database can be done with a **generic** prefix.

### Syntax:

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

Tables loaded through a **generic** statement are not auto-concatenated.

### Examples:

Generic LOAD \* from abc.csv;  
Generic SQL SELECT \* from table1;

## 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.



## 2 Script statements and keywords

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 <b>ParentName</b> 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, <b>NodeName</b> will be used.
PathName	A string used to name the new <b>Path</b> 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 <b>Path</b> field. Optional parameter. If omitted, '\' will be used.
Depth	A string used to name the new <b>Depth</b> 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
];
```

Node ID	ParentID	NodeName	ParentName	NodeName	NodeName	NodeName	PathName	Depth
1	4	London	Europe	UK	London	UK	Europe\UK\London	3
2	3	Munich	Europe	Germany	Munich	Germany	Europe\Germany\Munich	3

## 2 Script statements and keywords

3	5	German y	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.

## 2 Script statements and keywords

Argument	Description
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 <b>DepthDiff</b> 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
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.

## 2 Script statements and keywords

### 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 <b>LOAD</b> or <b>SELECT</b> statement for the loaded table.

### Example 1:

Table1

Column1	Column2
A	B
1	aa
2	cc
3	ee

Table2

Column1	Column2
A	C
1	xx
4	yy

QVTable:

```
SQL SELECT * From table1;  
inner join SQL SELECT * From table2;
```

QVTable

Column1	Column2	Column1
A	B	C
1	aa	xx

### Example 2:

QVTab1:

```
SQL SELECT * From Table1;
```

QVTab2:

```
inner keep SQL SELECT * From Table2;
```

QVTab1

Column1	Column2
A	B
1	aa

QVTab2

Column1	Column2
A	C
1	xx

The two tables in the **keep** example are, of course, associated via A.

### 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 or selectstatement	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 <b>IntervalMatch</b> 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:

## 2 Script statements and keywords

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.

#### Syntax:

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

The join is a natural join made over all the common fields. The join statement may be preceded by one of the prefixes **inner**, **outer**, **left** or **right**.

#### Arguments:

Arguments

Argument	Description
tablename	The named table to be compared to the loaded table.
loadstatement or selectstatement	The <b>LOAD</b> or <b>SELECT</b> statement for the loaded table.



### Example:

```
Join LOAD * from abc.csv;

Join SELECT * from table1;

tab1:
LOAD * from file1.csv;
tab2:
LOAD * from file2.csv;
.. ..
join (tab1) LOAD * from file3.csv;
```

## 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 <b>LOAD</b> or <b>SELECT</b> 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 741)*

#### 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 <b>LOAD</b> or <b>SELECT</b> statement for the loaded table.

#### Example:

Example table 1

Column1	Column2
A	B
1	aa
2	cc
3	ee

Example table 2

Column1	Column2
A	C
1	xx
4	yy

qvTable:

## 2 Script statements and keywords

```
SELECT * From table1;  
Left Join Sselect * From table2;
```

QVTable

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

```
QVTab1:  
SELECT * From Table1;  
QVTab2:  
Left Keep SELECT * From Table2;
```

QVTab1

Column1	Column2
A	B
1	aa
2	cc
3	ee

QVTab2

Column1	Column2
A	C
1	xx

The two tables in the **keep** example are, of course, associated via A.

```
tab1:  
LOAD * From file1.csv;  
tab2:  
LOAD * From file2.csv;  
... ..  
Left Keep (tab1) LOAD * From file3.csv;
```

## 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 )
```

## 2 Script statements and keywords

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, Risttu] ;
// 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

## 2 Script statements and keywords

Salesperson	Country
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.



*Partial reload is supported by using the Qlik Engine JSON API, or the **Reload** button.*

#### Syntax:

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

During a normal (non-partial) reload, the **Merge LOAD** construction will work 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 will be 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.

If the **Concatenate** prefix is used, or if there already exists a table with the same set of fields, this will be the relevant table to modify. Otherwise, the result of the **Merge LOAD** construction will be stored in a new table.

If the **Concatenate** prefix is used, the resulting table may have more fields than the change log that is used as input to the **Merge**.

A partial reload will do the same as a full reload. The only difference is that the **Merge LOAD** construction will never create a new table. There always exists a relevant table from the previous script execution to modify.

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** will assign a new value to the **SequenceNoVar** with the maximum value seen in the **SequenceNoField** field.

### 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.
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.
ListOfKeys	A comma separated list of field names specifying the primary key.
operation	The first field of the load statement must contain the operation: Insert, Update, or Delete. 'i', 'u' and 'd' are also accepted.

### Example: Data load script

In this example, an inline table named Persons is loaded with four 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 will be set to the maximum value in the *ChangeDate* column after **Merge** is executed.

```
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
];
```

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

## 2 Script statements and keywords

Resulting table

Name	Number
Jake	3
Jill	2
Steven	3

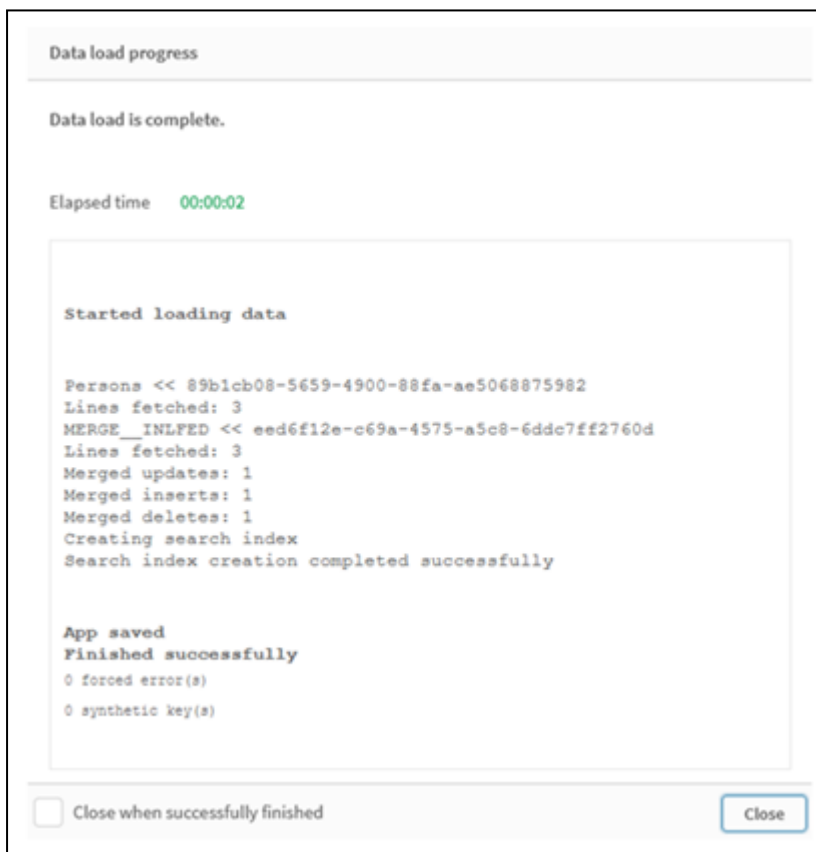
Following the **Merge** load, the table appears as follows:

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: Data load script - Partial reload

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

```
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;
```

### Outer

The explicit **Join** prefix can be preceded by the prefix **Outer** in order 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 explicit **Join** prefix can be preceded by the prefix **Outer** in order to specify an outer join. In an outer join, the resulting table will contain all values from both raw tables where the linking field values are represented in either 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 <b>LOAD</b> or <b>SELECT</b> statement for the loaded table.



## 2 Script statements and keywords

### Example:

Table1

Column1	Column2
A	B
1	aa
2	cc
3	ee

Table2

Column1	Column2
A	C
1	xx
4	yy

```
SQL SELECT * from table1;  
join SQL SELECT * from table2;  
OR
```

```
SQL SELECT * from table1;  
outer join SQL SELECT * from table2;
```

Joined table

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

### Partial reload

A normal (non-partial) 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 normal (non-partial) reloads.



*Partial reload is supported by using the Qlik Engine JSON API, or the **Reload** button.*

### 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.



*Partial reload is supported by using the Qlik Engine JSON API, or the **Reload** button.*

#### 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.

## 2 Script statements and keywords

Example	Result
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 746)*

#### 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 <b>LOAD</b> or <b>SELECT</b> statement for the loaded table.

## 2 Script statements and keywords

---

### Examples:

Example table 1

Column1	Column2
A	B
1	aa
2	cc
3	ee

Example table 2

Column1	Column2
A	C
1	xx
4	yy

QVTable:

```
SQL SELECT * from table1;  
right join SQL SELECT * from table2;
```

QVTable

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

QVTab1:

```
SQL SELECT * from Table1;
```

QVTab2:

```
right keep SQL SELECT * from Table2;
```

QVTab1

Column1	Column2
A	B
1	aa

QVTab2

Column1	Column2
A	C
1	xx
4	yy

The two tables in the **keep** example are, of course, associated via A.

```
tab1:
LOAD * from file1.csv;
tab2:
LOAD * from file2.csv;
... ..
right keep (tab1) LOAD * from file3.csv;
```

### 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 evaluates 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

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.

#### Syntax:

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

The semantic load will create semantic fields that can be displayed in filter panes to be used for navigation in the data.

Tables loaded through a **semantic** statement cannot be concatenated.

### Example:

```
Semantic LOAD * from abc.csv;  
Semantic SELECT Object1, Relation, Object2, InverseRelation from table1;
```

## 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 <b>exit for</b> , <b>exit do</b> or <b>exit sub</b> clause or an <b>exit script</b> 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.

## 2 Script statements and keywords

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

### Syntax:

#### Arguments

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

---

## 2 Script statements and keywords

---

### Binary

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

```
Binary file
file ::= [ path ] filename
```

### 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

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>:
```



## 2 Script statements and keywords

```
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
```

### Direct Query

The **DIRECT QUERY** statement allows you to access tables through an ODBC or OLE DB connection using the Direct Discovery function.

```
Direct Query [path]
```

### 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 '=' before it is assigned to the variable.

```
Let variablename=expression
```

---

## 2 Script statements and keywords

---

### 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.

---

## 2 Script statements and keywords

---

### 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

This script function creates a QVD or a CSV 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

## 2 Script statements and keywords

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 <b>SELECT</b> and <b>LOAD</b> statements. A new alias can be defined for a field name by a new <b>alias</b> 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 393) 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] ]
```

## 2 Script statements and keywords

### Arguments:

Arguments

Argument	Description
*fieldlist	A comma-separated list of the fields where the values should be replaced by a symbol key.  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	<b>Using</b> 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:

In this example we replace field values with symbol table keys using the **AutoNumber** statement 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 an **AutoNumber** statement with the Region, Year and Month fields.

```
RegionSales:
LOAD * INLINE
[ Region, Year, Month, Sales
North, 2014, May, 245
North, 2014, May, 347
North, 2014, June, 127
South, 2014, June, 645
```

## 2 Script statements and keywords

```
South, 2013, May, 367  
South, 2013, May, 221  
];
```

AutoNumber Region, Year, Month;

The resulting table would look like this:

Results table

Region	Year	Month	Sales
1	2	1	245
1	2	1	347
1	2	2	127
2	2	2	645
2	1	1	367
2	1	1	221

### 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.



*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
```

#### Syntax:

```
binary [path] filename
```



### Arguments:

#### Arguments

Argument	Description
path	<p>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.</p> <p><b>Example: 'lib://Table Files/'</b></p> <p>In legacy scripting mode, the following path formats are also supported:</p> <ul style="list-style-type: none"><li>absolute</li></ul> <p><b>Example: c:\data\</b></p> <ul style="list-style-type: none"><li>relative to the app containing this script line.</li></ul> <p><b>Example: data\</b></p>
filename	<p>The name of the file, including the file extension .qvw or .qvf.</p>

### 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 located in the <b>Folder</b> data connection. This may be, for example, a folder that your administrator creates on the Qlik Sense server. Click <b>Create new connection</b> in the data load editor and then select <b>Folder</b> under <b>File locations</b> .
Binary customer.qvf;	In this example, the file must be 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
<i>*fieldlist</i>	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.
<i>mapname</i>	The name of a mapping table previously read in a mapping <b>LOAD</b> or mapping <b>SELECT</b> statement.
<i>fieldname</i>	The name of the field that should be commented.
<i>comment</i>	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.

## 2 Script statements and keywords

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
<i>tablelist</i>	(table{,table})
<i>mapname</i>	The name of a mapping table previously read in a mapping <b>LOAD</b> or mapping <b>SELECT</b> statement.
<i>tablename</i>	The name of the table that should be commented.
<i>comment</i>	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 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	<p><code>connect-string ::= datasource { ; conn-spec-item }</code></p> <p>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.</p> <p><b>datasource</b> must be a defined ODBC data source or a string that defines an OLE DB provider.</p> <p><code>conn-spec-item ::= DBQ=database_specifier   DriverID=driver_specifier   UID=userid   PWD=password</code></p> <p>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.</p>
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	<p>Name of the field definition, ended with a colon.</p> <div> <i>Do not use autoCalendar as name for field definitions, as this name is reserved for auto-generated calendar templates.</i></div> <p><b>Example:</b></p> <p>calendar:</p>
tag_list	<p>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.</p> <p><b>Example:</b></p> <p>'\$date'Thank you for bringing this to our attention, and apologies for the inconvenience.</p>
parameter_list	<p>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.</p> <p><b>Example:</b></p> <p>first_month_of_year = 1</p>

## 2 Script statements and keywords

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> <b>As</b> field_name <b>tagged</b> tag. Use \$1 to reference the data field from which the derived fields should be generated.  <b>Example:</b>  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.  <b>Example:</b>  MyCalendar:

## 2 Script statements and keywords

Argument	Description
existing_definition	<p>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 <code>parameter_assignment</code> to change a value used in the field expressions.</p> <p><b>Example:</b></p> <p>Using Calendar</p>
parameter_assignment	<p>A comma separated list of parameter assignments. A parameter assignment is defined in the form <code>name=value</code> and overrides the parameter value that is set in the base field definition. Optional.</p> <p><b>Example:</b></p> <p>first_month_of_year = 4</p>

### 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.  <b>Example:</b> 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.  <b>Example:</b> 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.  <b>Example:</b> ('\$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;`

## Direct Query

The **DIRECT QUERY** statement allows you to access tables through an ODBC or OLE DB connection using the Direct Discovery function.

### Syntax:

```
DIRECT QUERY DIMENSION fieldlist [MEASURE fieldlist] [DETAIL fieldlist]
FROM tablelist
[WHERE where_clause]
```

The **DIMENSION**, **MEASURE**, and **DETAIL** keywords can be used in any order.

The **DIMENSION** and **FROM** keyword clauses are required on all **DIRECT QUERY** statements. The **FROM** keyword must appear after the **DIMENSION** keyword.



## 2 Script statements and keywords

The fields specified directly after the **DIMENSION** keyword are loaded in memory and can be used to create associations between in-memory and Direct Discovery data.



*The **DIRECT QUERY** statement cannot contain **DISTINCT** or **GROUP BY** clauses.*

Using the **MEASURE** keyword you can define fields that Qlik Sense is aware of on a “meta level”. The actual data of a measure field resides only in the database during the data load process, and is retrieved on an ad hoc basis driven by the chart expressions that are used in a visualization.

Typically, fields with discrete values that will be used as dimensions should be loaded with the **DIMENSION** keyword, whereas numbers that will be used in aggregations only should be selected with the **MEASURE** keyword.

**DETAIL** fields provide information or details, like comment fields, that a user may want to display in a drill-to-details table box. **DETAIL** fields cannot be used in chart expressions.

By design, the **DIRECT QUERY** statement is data-source neutral for data sources that support SQL. For that reason, the same **DIRECT QUERY** statement can be used for different SQL databases without change. Direct Discovery generates database-appropriate queries as needed.

Native data-source syntax can be used when the user knows the database to be queried and wants to exploit database-specific extensions to SQL. Native data-source syntax is supported:

- As field expressions in **DIMENSION** and **MEASURE** clauses
- As the content of the **WHERE** clause

Examples:

**DIRECT QUERY**

```
DIMENSION Dim1, Dim2
MEASURE
    NATIVE ('X % Y') AS X_MOD_Y
```

**FROM** TableName

**DIRECT QUERY**

```
DIMENSION Dim1, Dim2
MEASURE X, Y
FROM TableName
WHERE NATIVE ('EMAIL MATCHES "*"*.EDU"')
```



*The following terms are used as keywords and so cannot be used as column or field names without being quoted: and, as, detach, detail, dimension, distinct, from, in, is, like, measure, native, not, or, where*

### Arguments:

Argument	Description
fieldlist	A comma-separated list of field specifications, <i>fieldname</i> {, <i>fieldname</i> }. A field specification can be a field name, in which case the same name is used for the database column name and the Qlik Sense field name. Or a field specification can be a "field alias," in which case a database expression or column name is given a Qlik Sense field name.
tablelist	A list of the names of tables or views in the database from which data will be loaded. Typically, it will be views that contain a JOIN performed on the database.
where_ clause	<p>The full syntax of database <b>WHERE</b> clauses is not defined here, but most SQL "relational expressions" are allowed, including the use of function calls, the <b>LIKE</b> operator for strings, <b>IS NULL</b> and <b>IS NOT NULL</b>, and <b>IN</b>. <b>BETWEEN</b> is not included.</p> <p><b>NOT</b> is a unary operator, as opposed to a modifier on certain keywords.</p> <p>Examples:</p> <pre>WHERE x &gt; 100 AND "Region Code" IN ('south', 'west') WHERE Code IS NOT NULL and Code LIKE '%prospect' WHERE NOT x in (1,2,3)</pre> <p>The last example can not be written as:</p> <pre>WHERE x NOT in (1,2,3)</pre>

### Example:

In this example, a database table called TableName, containing fields Dim1, Dim2, Num1, Num2 and Num3, is used. Dim1 and Dim2 will be loaded into the Qlik Sense dataset.

```
DIRECT QUERY DIMENSION Dim1, Dim2 MEASURE Num1, Num2, Num3 FROM TableName ;
```

Dim1 and Dim2 will be available for use as dimensions. Num1, Num2 and Num3 will be available for aggregations. Dim1 and Dim2 are also available for aggregations. The type of aggregations for which Dim1 and Dim2 can be used depends on their data types. For example, in many cases **DIMENSION** fields contain string data such as names or account numbers. Those fields cannot be summed, but they can be counted: `count(Dim1)`.



**DIRECT QUERY** statements are written directly in the script editor. To simplify construction of **DIRECT QUERY** statements, you can generate a **SELECT** statement from a data connection, and then edit the generated script to change it into a **DIRECT QUERY** statement. For example, the **SELECT** statement:

```
SQL SELECT
  SalesOrderID,
  RevisionNumber,
  OrderDate,
  SubTotal,
  TaxAmt
FROM MyDB.Sales.SalesOrderHeader;
```

could be changed to the following **DIRECT QUERY** statement:

```
DIRECT QUERY
DIMENSION
  SalesOrderID,
  RevisionNumber

MEASURE
  SubTotal,
  TaxAmt

DETAIL
  OrderDate

FROM MyDB.Sales.SalesOrderHeader;
```

### Direct Discovery field lists

A field list is a comma-separated list of field specifications, *fieldname {, fieldname}*. A field specification can be a field name, in which case the same name is used for the database column name and the field name. Or a field specification can be a field alias, in which case a database expression or column name is given a Qlik Sense field name.

Field names can be either simple names or quoted names. A simple name begins with an alphabetic Unicode character and is followed by any combination of alphabetic or numeric characters or underscores. Quoted names begin with a double quotation mark and contain any sequence of characters. If a quoted name contains double quotation marks, those quotation marks are represented using two adjacent double quotation marks.

---

## 2 Script statements and keywords

---

Qlik Sense field names are case-sensitive. Database field names may or may not be case-sensitive, depending on the database. A Direct Discovery query preserves the case of all field identifiers and aliases. In the following example, the alias "MyState" is used internally to store the data from the database column "STATEID".

```
DIRECT QUERY Dimension STATEID as MyState Measure AMOUNT from SALES_TABLE;
```

This differs from the result of an **SQL Select** statement with an alias. If the alias is not explicitly quoted, the result contains the default case of column returned by the target database. In the following example, the **SQL Select** statement to an Oracle database creates "MYSTATE," with all upper case letters, as the internal Qlik Sense alias even though the alias is specified as mixed case. The **SQL Select** statement uses the column name returned by the database, which in the case of Oracle is all upper case.

```
SQL Select STATEID as MyState, STATENAME from STATE_TABLE;
```

To avoid this behavior, use the LOAD statement to specify the alias.

```
Load STATEID as MyState, STATENAME;  
SQL Select STATEID, STATEMENT from STATE_TABLE;
```

In this example, the "STATEID" column is stored internally by Qlik Sense as "MyState".

Most database scalar expressions are allowed as field specifications. Function calls can also be used in field specifications. Expressions can contain constants that are boolean, numeric, or strings contained in single quotation marks (embedded single quotation marks are represented by adjacent single quotation marks).

### Examples:

```
DIRECT QUERY  
  
  DIMENSION  
  
    SalesOrderID, RevisionNumber  
  
  MEASURE  
  
    SubTotal AS "Sub Total"  
  
FROM Adventureworks.Sales.SalesOrderHeader;  
  
DIRECT QUERY  
  
  DIMENSION  
  
    "SalesOrderID" AS "Sales Order ID"  
  
  MEASURE  
  
    SubTotal, TaxAmt, (SubTotal-TaxAmt) AS "Net Total"  
  
FROM Adventureworks.Sales.SalesOrderHeader;
```

```
DIRECT QUERY
```

```
DIMENSION
```

```
(2*Radius*3.14159) AS Circumference,
```

```
Molecules/6.02e23 AS Moles
```

```
MEASURE
```

```
Num1 AS numA
```

```
FROM TableName;
```

```
DIRECT QUERY
```

```
DIMENSION
```

```
concat(region, 'code') AS region_code
```

```
MEASURE
```

```
Num1 AS NumA
```

```
FROM TableName;
```

Direct Discovery does not support using aggregations in **LOAD** statements. If aggregations are used, the results are unpredictable. A **LOAD** statement such as the following should not be used:

```
DIRECT QUERY DIMENSION stateid, SUM(amount*7) AS MultiFirst MEASURE amount FROM sales_table;
```

The **SUM** should not be in the **LOAD** statement.

Direct Discovery also does not support Qlik Sense functions in **Direct Query** statements. For example, the following specification for a **DIMENSION** field results in a failure when the "Mth" field is used as a dimension in a visualization:

```
month(ModifiedDate) as Mth
```

## 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
<b>path</b>	<p>A text that can be interpreted as the path to the qvf file.</p> <p>The path is the path to the file, either:</p> <ul style="list-style-type: none"><li>• absolute <b>Example: <i>c:\data\</i></b></li><li>• relative to the Qlik Sense app working directory. <b>Example: <i>data\</i></b></li><li>• URL address (HTTP or FTP), pointing to a location on the Internet or an intranet. <b>Example: <i>http://www.qlik.com</i></b></li></ul>

### 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 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).

## 2 Script statements and keywords

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 <i>Tab2</i> is created, the table <i>Tab1</i> 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 statement is not supported in standard mode.*

#### Syntax:

```
execute commandline
```

#### Arguments:

##### Arguments

Argument	Description
<i>commandline</i>	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.

## 2 Script statements and keywords

### 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
<p>This example shows how to force capitalization.</p> <pre>FORCE Capitalization; Capitalization: LOAD * Inline [ ab Cd eF GH ];</pre>	<p>The <b>Capitalization</b> table contains the following values:</p> <p>Ab Cd Ef Gh All values are capitalized.</p>
<p>This example shows how to force case upper.</p> <pre>FORCE Case Upper; CaseUpper: LOAD * Inline [ ab Cd eF GH ];</pre>	<p>The <b>CaseUpper</b> table contains the following values:</p> <p>AB CD EF GH All values are upper case.</p>
<p>This example shows how to force case lower.</p> <pre>FORCE Case Lower; CaseLower: LOAD * Inline [ ab Cd eF GH ];</pre>	<p>The <b>CaseLower</b> table contains the following values:</p> <p>ab cd ef gh All values are lower case.</p>
<p>This example shows how to force case mixed.</p> <pre>FORCE Case Mixed; CaseMixed: LOAD * Inline [ ab Cd eF GH ];</pre>	<p>The <b>CaseMixed</b> table contains the following values:</p> <p>ab Cd eF GH All values are as they appear in the script.</p>

See also:

### 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 fieldsource [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	<p>You can use <b>distinct</b> as a predicate if you only want to load unique records. If there are duplicate records, the first instance will be loaded.</p> <p>If you are using preceding loads, you need to place <b>distinct</b> in the first load statement, as <b>distinct</b> only affects the destination table.</p>

## 2 Script statements and keywords

Argument	Description
fieldlist	<p><i>fieldlist</i> ::= ( *   <i>field</i> { , *   <i>field</i> } )</p> <p>A list of the fields to be loaded. Using * as a field list indicates all fields in the table.</p> <p><i>field</i> ::= ( <i>fieldref</i>   <i>expression</i> ) [<b>as</b> <i>aliasname</i> ]</p> <p>The field definition must always contain a literal, a reference to an existing field, or an expression.</p> <p><i>fieldref</i> ::= ( <i>fieldname</i>   @<i>fieldnumber</i>   @<i>startpos:endpos</i> [ <b>I</b>   <b>U</b>   <b>R</b>   <b>B</b>   <b>T</b> ] )</p> <p><i>fieldname</i> 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:</p> <p>@<i>fieldnumber</i> 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.</p> <p>@<i>startpos:endpos</i> 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, <b>n</b> is used as end position.</p> <ul style="list-style-type: none"> <li>• If @<i>startpos:endpos</i> is immediately followed by the characters <b>I</b> or <b>U</b>, the bytes read will be interpreted as a binary signed (<b>I</b>) or unsigned (<b>U</b>) integer (Intel byte order). The number of positions read must be 1, 2 or 4.</li> <li>• If @<i>startpos:endpos</i> is immediately followed by the character <b>R</b>, 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.</li> <li>• If @<i>startpos:endpos</i> is immediately followed by the character <b>B</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.</li> </ul> <p><i>expression</i> 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.</p> <p><b>as</b> is used for assigning a new name to the field.</p>

Argument	Description
from	<p><b>from</b> is used if data should be loaded from a file using a folder or a web file data connection.</p> <p><i>file ::= [ path ] filename</i></p> <p><b>Example: 'lib://Table Files/'</b></p> <p>If the path is omitted, Qlik Sense searches for the file in the directory specified by the <b>Directory</b> statement. If there is no <b>Directory</b> statement, Qlik Sense searches in the working directory, <i>C:\Users\{user}\Documents\Qlik\Sense\Apps</i>.</p> <div style="border: 1px solid #ccc; padding: 10px; margin: 10px 0;"> <p> <i>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.</i></p> </div> <p>The <i>filename</i> may contain the standard DOS wildcard characters ( * and ? ). This will cause all the matching files in the specified directory to be loaded.</p> <p><i>format-spec ::= ( fspec-item { , fspec-item } )</i></p> <p>The format specification consists of a list of several format specification items, within brackets.</p> <p><b>Legacy scripting mode</b></p> <p>In legacy scripting mode, the following path formats are also supported:</p> <ul style="list-style-type: none"> <li>absolute</li> </ul> <p><b>Example: c:\data\</b></p> <ul style="list-style-type: none"> <li>relative to the Qlik Sense app working directory.</li> </ul> <p><b>Example: data\</b></p> <ul style="list-style-type: none"> <li>URL address (HTTP or FTP), pointing to a location on the Internet or an intranet.</li> </ul> <p><b>Example: http://www.qlik.com</b></p>

## 2 Script statements and keywords

Argument	Description
from_field	<p><b>from_field</b> is used if data should be loaded from a previously loaded field. <i>fieldsource::=(tablename, fieldname)</i></p> <p>The field is the name of the previously loaded <i>tablename</i> and <i>fieldname</i>. <i>format-spec ::= ( fspec-item {, fspec-item } )</i> The format specification consists of a list of several format specification items, within brackets.</p>
inline	<p><b>inline</b> is used if data should be typed within the script, and not loaded from a file. <i>data ::= [ text ]</i></p> <p>Data entered through an <b>inline</b> 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 <b>inline</b> clause, i.e. by pressing the Enter key when typing the script. The number of columns are defined by the first line. <i>format-spec ::= ( fspec-item {, fspec-item } )</i> The format specification consists of a list of several format specification items, within brackets.</p>
resident	<p><b>resident</b> is used if data should be loaded from a previously loaded table. <i>table label</i> is a label preceding the <b>LOAD</b> or <b>SELECT</b> statement(s) that created the original table. The label should be given with a colon at the end.</p>
autogenerate	<p><b>autogenerate</b> is used if data should be automatically generated by Qlik Sense. <i>size ::= number</i></p> <p><i>Number</i> is an integer indicating the number of records to be generated.</p> <p>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 <b>Peek</b> function.</p>

## 2 Script statements and keywords

Argument	Description
extension	<p>You can load data from analytic connections. You need to use the <b>extension</b> clause to call a function defined in the server-side extension (SSE) plugin, or evaluate a script.</p> <p>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.</p> <pre>Extension pluginname.functionname( tabledescription );</pre> <ul style="list-style-type: none"><li>• Loading data using a function in an SSE plugin <i>tabledescription ::= (table { ,tablefield} )</i> If you do not state table fields, the fields will be used in load order.</li><li>• Loading data by evaluating a script in an SSE plugin <i>tabledescription ::= ( script, table { ,tablefield} )</i></li></ul> <p><b>Data type handling in the table field definition</b></p> <p>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.</p> <p>You can force the data type by wrapping a field name with <b>String()</b> or <b>Mixed()</b>.</p> <ul style="list-style-type: none"><li>• <b>String()</b> 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.</li><li>• <b>Mixed()</b> forces the field to be dual.</li></ul> <p><b>String()</b> or <b>Mixed()</b> cannot be used outside <b>extension</b> table field definitions, and you cannot use other Qlik Sense functions in a table field definition.</p> <p><b>More about analytic connections</b></p> <p>You need to configure analytic connections before you can use them.</p>
where	<p><b>where</b> is a clause used for stating whether a record should be included in the selection or not. The selection is included if <i>criterion</i> is True. <i>criterion</i> is a logical expression.</p>
while	<p><b>while</b> is a clause used for stating whether a record should be repeatedly read. The same record is read as long as <i>criterion</i> is True. In order to be useful, a <b>while</b> clause must typically include the <b>IterNo( )</b> function.</p> <p><i>criterion</i> is a logical expression.</p>

## 2 Script statements and keywords

Argument	Description
group by	<p><b>group by</b> 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.</p> <p><i>groupbyfieldlist ::= (fieldname { , fieldname } )</i></p>
order by	<p><b>order by</b> is a clause used for sorting the records of a resident table before they are processed by the <b>load</b> 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.</p> <p>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).</p> <p><i>orderbyfieldlist ::= fieldname [ sortorder ] { , fieldname [ sortorder ] }</i></p> <p><i>sortorder</i> is either <i>asc</i> for ascending or <i>desc</i> for descending. If no <i>sortorder</i> is specified, <i>asc</i> is assumed.</p> <p><i>fieldname</i>, <i>path</i>, <i>filename</i> and <i>aliasname</i> are text strings representing what the respective names imply. Any field in the source table can be used as <i>fieldname</i>. However, fields created through the <i>as</i> clause (<i>aliasname</i>) are out of scope and cannot be used inside the same <b>load</b> statement.</p>

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);
```



---

## 2 Script statements and keywords

---

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;
```

## 2 Script statements and keywords

---

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:

---

## 2 Script statements and keywords

---

```
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
<b>utf8</b>	UTF-8 character set
<b>unicode</b>	Unicode character set
<b>ansi</b>	Windows, codepage 1252
<b>oem</b>	DOS, OS/2, AS400 and others
<b>mac</b>	Codepage 10000
<b>codepage is</b>	With the <b>codepage</b> specifier, it is possible to use any Windows codepage as <i>N</i> .


### 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 \(page 91\)](#)

### 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	<p>In a fixed record file, each field is exactly a certain number of characters.</p> <p>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 <b>Record is</b>.</p> <div> <i>If the data contains multi-byte characters, field breaks can become misaligned as the format is based on a fixed length in bytes.</i></div>
dif	In a <b>.dif</b> 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 <b>biff</b> format (Binary Interchange File Format).
ooxml	Excel 2007 and later versions use the ooxml <b>.xlsx</b> 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 <b>qvd</b> is the proprietary QVD files format, exported from a Qlik Sense app.
qvx	<b>qvx</b> 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 \(page 91\)](#)

### 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 \(page 91\)](#)

## 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 \(page 91\)](#)

## 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 \(page 91\)](#)

### 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 \(page 91\)](#)

### 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	<p>Used to specify modern style quoting, allowing multi-line content in fields. Fields containing end-of-line characters must be enclosed within double quotes.</p> <p>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.</p>

### 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 \(page 91\)](#)

### 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 \(page 91\)](#)

### 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 \(page 91\)](#)

## 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 '=' before it is assigned to the variable.

### Syntax:

```
Let variablename=expression
```

The word **let** may be omitted, but the statement then becomes a control statement. Such a statement without the keyword **let** must be contained within a single script row and may be terminated either with a semicolon or end-of-line.

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 '
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 ...]
```

## 2 Script statements and keywords

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
<i>fieldlist</i>	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.
<i>mapname</i>	The name of a mapping table previously read in a <b>mapping load</b> or <b>mapping select</b> statement.

## 2 Script statements and keywords

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 <b>NullAsNull</b> 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.

## 2 Script statements and keywords

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 <b>NullAsValue</b> 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 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:
```

## 2 Script statements and keywords

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
```

## 2 Script statements and keywords

---

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.

#### 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	<b>distinct</b> is a predicate used if duplicate combinations of values in the selected fields only should be loaded once.
distinctrow	<b>distinctrow</b> is a predicate used if duplicate records in the source table only should be loaded once.
fieldlist	<p><b>fieldlist</b> ::= ( *   field ) { , field }</p> <p>A list of the fields to be selected. Using * as field list indicates all fields in the table.</p> <p><b>fieldlist</b> ::= field { , field }</p> <p>A list of one or more fields, separated by commas.</p> <p><b>field</b> ::= ( fieldref   expression ) [ as aliasname ]</p> <p>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: +, -, *, /, &amp; (string concatenation), sum(fieldname), count(fieldname), avg(fieldname)(average), month(fieldname), etc. See the documentation of the ODBC driver for more information.</p> <p><b>fieldref</b> ::= [ tablename. ] fieldname</p> <p>The <b>tablename</b> and the <b>fieldname</b> are text strings identical to what they imply. They must be enclosed by straight double quotation marks if they contain e.g. spaces.</p> <p>The <b>as</b> clause is used for assigning a new name to the field.</p>
from	<p><b>tablelist</b> ::= table { , table }</p> <p>The list of tables that the fields are to be selected from.</p> <p><b>table</b> ::= tablename [ [ as ] aliasname ]</p> <p>The <b>tablename</b> may or may not be put within quotes.</p>
where	<p><b>where</b> is a clause used for stating whether a record should be included in the selection or not.</p> <p><b>criterion</b> is a logical expression that can sometimes be very complex. Some of the operators accepted are: numeric operators and functions, =, &lt;&gt; or #(not equal), &gt;, &gt;=, &lt;, &lt;=, and, or, not, exists, some, all, in and also new <b>SELECT</b> statements. See the documentation of the ODBC driver or OLE DB provider for more information.</p>
group by	<b>group by</b> 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	<b>having</b> is a clause used for qualifying groups in a similar manner to how the <b>where</b> clause is used for qualifying records.

## 2 Script statements and keywords

Argument	Description
order by	<b>order by</b> is a clause used for stating the sort order of the resulting table of the <b>SELECT</b> statement.
join	<b>join</b> 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 <b>Connect</b> 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 <i>n</i> 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
<i>sql_command</i>	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
```

## 2 Script statements and keywords

---

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	<p>An arbitrary text. Note that the string must be enclosed by quotation marks if it contains blanks.</p> <p>If nothing is specified, <b>star is;</b> is assumed, i.e. there is no star symbol available unless explicitly specified. This definition is valid until a new <b>star</b> statement is made.</p>

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 *Starsign* is *\**.
- 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. The star sign *\** means all listed values, not all values of the field. This means that the user *USER5* cannot see the fields *SALES*, *WAREHOUSE* and *EMPLOYEES* but this user can see the field *ORDERS*.

## Store

This script function creates a QVD or a CSV 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
<i>fieldlist::= ( *   field ) { , field }</i>	A list of the fields to be selected. Using <i>*</i> as field list indicates all fields.  <i>field::= fieldname [as aliasname]</i>  <i>fieldname</i> is a text that is identical to a field name in <i>table</i> . (Note that the field name must be enclosed by straight double quotation marks or square brackets if it contains spaces or other non-standard characters.)  <i>aliasname</i> is an alternate name for the field to be used in the resulting QVD or CSV file.

## 2 Script statements and keywords

Argument	Description
<i>table</i>	A script label representing an already loaded table to be used as source for data.
<i>filename</i>	<p>The name of the target file including a valid path to an existing folder data connection.</p> <p><b>Example: 'lib://Table Files/target.qvd'</b></p> <p>In legacy scripting mode, the following path formats are also supported:</p> <ul style="list-style-type: none"><li>absolute</li></ul> <p><b>Example: c:\data\sales.qvd</b></p> <ul style="list-style-type: none"><li>relative to the Qlik Sense app working directory.</li></ul> <p><b>Example: data\sales.qvd</b></p> <p>If the path is omitted, Qlik Sense stores the file in the directory specified by the <b>Directory</b> statement. If there is no <b>Directory</b> statement, Qlik Sense stores the file in the working directory, <i>C:\Users\{user}\Documents\Qlik\Sense\Apps</i>.</p>
<i>format-spec ::= ( txt   qvd )</i>	The format specification consists of the text <b>txt</b> for text files, or the text <b>qvd</b> for qvd files. If the format specification is omitted, <b>qvd</b> 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.*

### 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.

## 2 Script statements and keywords

### 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 <b>mapping Load</b> or <b>mapping Select</b> 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:

```
Trace Main table loaded;
```

### Example 2:

```
Let MyMessage = NoOfRows('MainTable') & ' rows in Main Table';
Trace $(MyMessage);
```

### 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	<p>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.</p> <p>Refer to the documentation for the <b>Qualify</b> statement for further information.</p>

### 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 <b>LOAD</b> or mapping <b>SELECT</b> 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

When defining a variable, the syntax:

```
set variablename = string
```

or

```
let variable = expression
```

is used. The **Set** command assigns the text to the right of the equal sign to the variable, whereas the **Let** command evaluates the expression.

Variables are case sensitive.



*It is not recommended to name a variable identically to a field or a function in Qlik Sense.*

#### Examples:

```
set HidePrefix = $ ; // the variable will get the character '$' as value.
```

```
let vToday = Num(Today()); // returns the date serial number of today.
```

### 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 overview.

### 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.

## 2 Working with variables in the data load editor

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 154)*
- *Number interpretation variables (page 140)*
- *System variables (page 132)*
- *Value handling variables (page 138)*

### 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

## 2 Working with variables in the data load editor



*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. 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.*

## 2 Working with variables in the data load editor

---

### 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. 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, that is, they 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, and in Qlik Sense, it is according to the operating system of the server where Qlik Sense is installed.

The variables are included at the top of the script of the new Qlik Sense app and substitute operating system defaults for certain number formatting settings at the time of the script execution. They can be deleted, edited or duplicated freely.



*If you want to create an app for a certain locale, the easiest way is probably to use Qlik Sense Desktop on a computer with the desired locale setting in the operating system to create the app. The app will then contain the appropriate regional settings of that locale, and you can move it to a Qlik Sense server of choice for further development.*

## Number interpretation 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.

### Currency formatting

#### **MoneyDecimalSep**

The decimal separator defined replaces the decimal symbol for currency of the operating system (regional settings).

```
MoneyDecimalSep
```

#### **MoneyFormat**

The symbol defined replaces the currency symbol of the operating system (regional settings).

```
MoneyFormat
```

#### **MoneyThousandSep**

The thousands separator defined replaces the digit grouping symbol for currency of the operating system (regional settings).

```
MoneyThousandSep
```

### Number formatting

#### **DecimalSep**

The decimal separator defined replaces the decimal symbol of the operating system (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  $\mu$  for micro ( $10^{-6}$ ).

```
NumericalAbbreviation
```

### Time formatting

#### **DateFormat**

The format defined replaces the date format of the operating system (regional settings).

```
DateFormat
```

#### **TimeFormat**

The format defined replaces the time format of the operating system (regional settings).

```
TimeFormat
```

## 2 Working with variables in the data load editor

---

### 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 operating system (regional settings).

**MonthNames**

### LongMonthNames

The format defined replaces the long month names convention of the operating system (regional settings).

**LongMonthNames**

### DayNames

The format defined replaces the weekday names convention of the operating system (regional settings).

**DayNames**

### LongDayNames

The format defined replaces the long weekday names convention of the operating system (regional settings).

**LongDayNames**

### FirstWeekDay

Integer that defines which day to use as the first day of the week.

**FirstWeekDay**

### BrokenWeeks

The 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

The 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)

### Examples:

```
Set BrokenWeeks=0; //(use unbroken weeks)
Set BrokenWeeks=1; //(use broken weeks)
```

## DateFormat

The format defined replaces the date format of the operating system (regional settings).

### Syntax:

**DateFormat**

### Examples:

```
Set DateFormat='M/D/YY'; //(US format)
Set DateFormat='DD/MM/YY'; //(UK date format)
Set DateFormat='YYYY-MM-DD'; //(ISO date format)
```

## DayNames

The format defined replaces the weekday names convention of the operating system (regional settings).

### Syntax:

**DayNames**

### Example:

```
Set DayNames='Mon;Tue;Wed;Thu;Fri;Sat;Sun';
```

## DecimalSep

The decimal separator defined replaces the decimal symbol of the operating system (regional settings).

### Syntax:

**DecimalSep**

### Examples:

```
Set DecimalSep='.';
Set DecimalSep=',';
```

## FirstWeekDay

Integer that defines which day to use as the first day of the week.

### Syntax:

**FirstWeekDay**

By default, Qlik Sense functions use Monday as the first day of the week. The following values can be used:

- 0 (= Monday)
- 1 (= Tuesday)
- 2 (= Wednesday)
- 3 (= Thursday)
- 4 (= Friday)
- 5 (= Saturday)
- 6 (= Sunday)

### Examples:

```
Set FirstWeekDay=6; //(set Sunday as the first day of the week)
```

## LongDayNames

The format defined replaces the long weekday names convention of the operating system (regional settings).

### Syntax:

**LongDayNames**

### Example:

```
Set LongDayNames='Monday;Tuesday;Wednesday;Thursday;Friday;Saturday;Sunday';
```



### LongMonthNames

The format defined replaces the long month names convention of the operating system (regional settings).

#### Syntax:

```
LongMonthNames
```

#### Example:

```
Set  
LongMonthNames='January;February;March;April;May;June;July;August;September;October;November;December';
```

### MoneyDecimalSep

The decimal separator defined replaces the decimal symbol for currency of the operating system (regional settings).

#### Syntax:

```
MoneyDecimalSep
```

#### Example:

```
Set MoneyDecimalSep='.';
```

### MoneyFormat

The symbol defined replaces the currency symbol of the operating system (regional settings).

#### Syntax:

```
MoneyFormat
```

#### Example:

```
Set MoneyFormat='$ #,##0.00; ($ #,##0.00)';
```

### MoneyThousandSep

The thousands separator defined replaces the digit grouping symbol for currency of the operating system (regional settings).

#### Syntax:

```
MoneyThousandSep
```

#### Example:

```
Set MoneyThousandSep=',';
```

### MonthNames

The format defined replaces the month names convention of the operating system (regional settings).

#### Syntax:

**MonthNames**

#### Example:

```
Set MonthNames='Jan;Feb;Mar;Apr;May;Jun;Jul;Aug;Sep;Oct;Nov;Dec';
```

### 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  $\mu$  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: $\mu$ ;-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: $\mu$ ;-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.

#### Syntax:

**ReferenceDay**

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:

```
Set ReferenceDay=3; //(set January 3 as the reference day)
```

## 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)
Set ThousandSep=' ';
```

## 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
```

### Examples and results:

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.

## 2 Working with variables in the data load editor

### 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

### 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';
```

## 2 Working with variables in the data load editor

---

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, 0.3333333333333333, 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]';

## 2 Working with variables in the data load editor

---

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, 0.3333333333333333, 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).

## 2 Working with variables in the data load editor

---

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\sb*, this will be evaluated to `SET QUERY_BAND = 'who=WA\sb;' 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 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.

#### 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
<b>ErrorMode=1</b>	The default setting. The script execution will halt and the user will be prompted for action (non-batch mode).
<b>ErrorMode=0</b>	Qlik Sense will simply ignore the failure and continue script execution at the next script statement.
<b>ErrorMode=2</b>	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
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
13	BIFF error
14	BIFF error encrypted
15	BIFF error unsupported version
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 Visualization expressions

An 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	<p>Using the total qualifier inside your aggregation function disregards the dimensional value.</p> <p>The aggregation will be performed on all possible field values.</p> <p>The <b>TOTAL</b> 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.</p>
Set analysis	<p>Using set analysis inside your aggregation overrides the selection. The aggregation will be performed on all values split across the dimensions.</p>

Method	Description
TOTAL qualifier and set analysis	Using the <b>TOTAL</b> qualifier and set analysis inside your aggregation overrides the selection and disregards the dimensions.
ALL qualifier	Using the <b>ALL</b> qualifier inside your aggregation disregards the selection and the dimensions. The equivalent can be achieved with the {1} set analysis statement and the <b>TOTAL</b> 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%



Year	Quarter	Sum(Amount)	Sum({1} Amount)	Sum(Amount)/Sum({1} Amount)
2012	Q2	1700	1700	100%
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 Syntax for sets

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_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 Set modifiers

A set can be modified by an additional or a changed selection. Such a modification can be written in the set expression.

The modifier consists of one or several field names, each followed by a selection that should be made on the field, all enclosed by angled brackets: < >. For example: <year={2007,2008},Region={US}>. Field names and field values can be quoted as usual, for example: <[Sales Region]='west coast', 'South America'>.

A set modifier modifies the selection of the preceding set identifier. If no set identifier is referenced, the current selection state is implicit.

There are several ways to define the selection:

- Based on another field
- Based on element sets (a field value list in the modifier)
- Forced exclusion

These methods are described in the following subsections.

#### Based on another field

A simple case is a selection 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.

#### Based on element sets

The most common example of a set expression is one that is based on a list of field values enclosed in curly brackets. The values are separated by commas, for example <Year = {2007, 2008}>. The curly brackets define an element set, where the elements can be either explicit field values or searches of field values.

Unless the listed values contain blanks or special characters, quotes are not needed. The listed values will simply be matched with the field values. This comparison is case insensitive.

If the listed values contain blanks or special characters, or if you want to use wild cards, then you need to enclose the values in quotation marks. Single quotes should be used if the listed values are explicit field values. Then case sensitive matches between the listed values and the individual field values will be made.

Double quotes should be used for searches, i.e. strings that contain wild cards or start with a relational operator or an equals sign. For example, `<Ingredient = {"*Garlic*"}` will select all ingredients that contain the string 'Garlic'. Double quotes can be substituted with brackets, for example, `<Ingredient = {[*Garlic*]}`. Double quotes can also be substituted with grave accents, for example `<Ingredient = {`*Garlic*`}`. Searches are case-insensitive.



*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.*


### Forced exclusion

Finally, for fields in AND-mode, there is also the possibility of forced exclusion. If you want to force exclusion of specific field values, you will need to use “~” in front of the field name.



*AND mode is only supported using Qlik Engine API.*

### Examples and results:

Examples	Results
<code>sum( {1&lt;Region={USA} &gt;} Sales )</code>	Returns the sales for the region USA disregarding the current selection
<code>sum( {\$&lt;Region = &gt;} Sales )</code>	Returns the sales for the current selection, but with the selection in 'Region' removed
<code>sum( {&lt;Region = &gt;} Sales )</code>	Returns the same as the example immediately above. When the set to modify is omitted, \$ is assumed. <div><i>The syntax in the two previous examples is interpreted as “no selections” in 'Region', that is to say all regions given other selections will be possible. It is not equivalent to the syntax <code>&lt;Region = {}&gt;</code> (or any other text on the right side of the equal sign implicitly resulting in an empty element set) which is interpreted as no region.</i></div>
<code>sum( {\$&lt;Year = {2000}, Region = {US, SE, DE, UK, FR}&gt;} Sales )</code>	Returns the sales for current selection, but with new selections both in 'Year' and in 'Region'.

Examples	Results
sum( {\$<~Ingredient = {"*garlic*"}>} Sales )	The field <i>Ingredient</i> is in AND mode.  Returns the sales for current selection, but with a forced exclusion of all ingredients containing the string 'garlic'.
sum( {\$<Year = {"2*"}>} Sales )	Returns the sales for the current selection, but with all years beginning with the digit "2", i.e. most likely year 2000 and onwards, selected in the field 'Year'.
sum( {\$<Year = {"2*", "198*"}>} Sales )	As above, but now also the 1980:s are included in the selection.
sum( {\$<Year = {">1978<2004"}>} Sales )	Returns the sales for the current selections, but with a numeric search used to scope the range of years to sum the sales across.

### Set modifiers with set operators

The selection within a field can be defined using set operators working on different element sets. For example the modifier `<Year = {"20*", 1997} - {2000}>` will select all years beginning with "20" in addition to "1997", except for "2000".

#### Examples and results:

##### Operator examples

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.
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 using assignments with implicit set operators

This notation defines new selections, disregarding the current selection in the field. However, if you want to base your selection on the current selection in the field and add field values, for example you may want a modifier `<Year = Year + {2007, 2008}>`. A short and equivalent way to write this is `<Year += {2007, 2008}>`, that is, the assignment operator implicitly defines a union. Also implicit intersections, exclusions and symmetric differences can be defined using `"*="`, `"-="` and `"/="`.

#### Examples and results:

Operator examples

Examples	Results
<code>sum( {\$&lt;Product += {OurProduct1, OurProduct2} &gt;} Sales )</code>	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.
<code>sum( {\$&lt;Year += {"20*", 1997} - {2000} &gt;} Sales )</code>	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 <code>&lt;Year=Year + ({"20*", 1997}-{2000})&gt;</code> .
<code>sum( {\$&lt;Product *= {OurProduct1} &gt;} Sales )</code>	Returns the sales for the current selection, but only for the intersection of currently selected products and the product OurProduct1.

### Set modifiers with advanced searches

Advanced searches using wild cards and aggregations can be used to define sets.

#### Examples and results:

Set modifier examples

Examples	Results
<code>sum( {\$-1&lt;Product = {"*Internal*", "*Domestic*"}&gt;} Sales )</code>	Returns the sales for current selection, excluding transactions pertaining to products with the string 'Internal' or 'Domestic' in the product name.
<code>sum( {\$&lt;Customer = {"=Sum ({1&lt;Year = {2007}&gt;} Sales ) &gt; 1000000"}&gt;} Sales )</code>	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

Variables and other dollar-sign expansions can be used in set expressions.

### Examples and results:

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 implicit field value definitions

The following describes how to define a set of field values using a nested set definition.

In such cases, the element functions P() and E() must be used, representing the element set of possible values and the excluded values of a field, respectively. Inside the parentheses, it is possible to specify one set expression and one field, for example P({1} Customer). These functions cannot be used in other expressions.



*The element functions, P() and E(), can only be used on a natural set. 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 selection, and is therefore, not a natural set. Using these functions on non-natural sets can give rise to unexpected results.*

### Examples and results:

#### Field value examples

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.

Examples	Results
<code>sum( {\$&lt;Customer = P({1&lt;Product= 'Shoe'})&gt; Supplier}&gt; Sales )</code>	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.
<code>sum( {\$&lt;Customer = E({1&lt;Product= 'Shoe'})&gt;}&gt; Sales )</code>	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.

### 3.4 Visualization expression and aggregation syntax

The syntax used for visualization (chart) expressions and aggregations is described in the following sections.

#### 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 ::= aggexpression { , aggexpression }
```

The number and types of parameters are not arbitrary. They depend on the function used.

#### General syntax for aggregations

The following general syntax structure can be used for aggregations, with many optional parameters:

```
aggexpression ::= ( fieldref | operator1 aggexpression | aggexpression operator2 aggexpression |
functioninaggr | ( aggexpression ) )
```

**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.  <b>Example:</b>  bitnot 17 returns -18
bitand	Bit and.	The operation returns the logical AND of the operands performed bit by bit.  <b>Example:</b>  17 bitand 7 returns 1
bitor	Bit or.	The operation returns the logical OR of the operands performed bit by bit.  <b>Example:</b>  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.  <b>Example:</b>  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.  <b>Example:</b>  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.  <b>Example:</b>  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
<b>precedes</b>	<p>Unlike the &lt; 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.</p> <p><b>Example:</b></p> <p>'1 ' precedes ' 2' returns FALSE</p> <p>' 1' precedes ' 2' returns TRUE</p> <p>as the ASCII value of a space (' ') is of less value than the ASCII value of a number.</p> <p>Compare this to:</p> <p>'1 ' &lt; ' 2' returns TRUE</p> <p>' 1' &lt; ' 2' returns TRUE</p>
<b>follows</b>	<p>Unlike the &gt; 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.</p> <p><b>Example:</b></p> <p>' 2' follows '1' returns FALSE</p> <p>'2' follows ' 1' returns TRUE</p> <p>as the ASCII value of a space (' ') is of less value than the ASCII value of a number.</p> <p>Compare this to:</p> <p>' 2' &gt; ' 1' returns TRUE</p> <p>' 2' &gt; '1 ' returns TRUE</p>

## 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 Functions in scripts and chart expressions

This section describes functions that can be used in Qlik Sense data load scripts and chart expressions to transform and aggregate data.

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, where the aggregation is defined by a chart dimension or a **group by** clause in the script.

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.

### Using aggregation functions in a data load script

Aggregation functions can only be used inside **LOAD** statements..

### Using aggregation functions in chart expressions

The expression 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 calculated dimensions.

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.

### 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 )
```

## 5 Functions in scripts and chart expressions

---

### 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. DataCustomerProductUnitSalesUnitPriceAstridaAA416AstridaAA1015AstridaBB99BetacabBB510BetacabCC220BetacabDD-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
```



---

## 5 Functions in scripts and chart expressions

---

```
(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|BB|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\_**

---

## 5 Functions in scripts and chart expressions

---

**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 <b>value</b> that corresponds to the result of sorting <b>sort_weight</b> .
sort-weight Expression	The expression containing the data to be sorted. The first (lowest) value of <b>sort_weight</b> is found, from which the corresponding value of the <b>value</b> expression is determined. If you place a minus sign in front of <b>sort_weight</b> , the function returns the last (highest) sorted value instead.
rank Expression	By stating a <b>rank</b> "n" larger than 1, you get the nth sorted value.
distinct	If the word <b>DISTINCT</b> 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.

## 5 Functions in scripts and chart expressions

### Scripting examples

Example	Result
<p>Temp:</p> <pre>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 ' ');</pre> <p>FirstSortedValue:</p> <pre>LOAD Customer,FirstSortedValue(Product, UnitSales) as MyProductWithSmallestOrderByCustomer Resident Temp Group By Customer;</pre>	<p>Customer MyProductWithSmallestOrderByCustomer Astrida CC Betacab AA Canutility AA Divadip DD</p> <p>The function sorts UnitSales from smallest to largest, looking for the value of Customer with the smallest value of UnitSales, the smallest order.</p> <p>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..</p>
<p>Given that the <b>Temp</b> table is loaded as in the previous example:</p> <pre>LOAD Customer,FirstSortedValue(Product, - UnitSales) as MyProductWithLargestOrderByCustomer Resident Temp Group By Customer;</pre>	<p>Customer MyProductWithLargestOrderByCustomer Astrida AA Betacab DD Canutility CC Divadip -</p> <p>A minus sign precedes the sort_weight argument, so the function sorts the largest first.</p> <p>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.</p>
<p>Given that the <b>Temp</b> table is loaded as in the previous example:</p> <pre>LOAD Customer,FirstSortedValue(distinct Product, -UnitSales) as MyProductWithSmallestOrderByCustomer Resident Temp Group By Customer;</pre>	<p>Customer MyProductWithLargestOrderByCustomer Astrida AA Betacab DD Canutility CC Divadip AA</p> <p>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.</p>

### 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 <b>value</b> that corresponds to the result of sorting <b>sort_weight</b> .
sort_weight	Input field. The expression containing the data to be sorted. The first (lowest) value of <b>sort_weight</b> is found, from which the corresponding value of the <b>value</b> expression is determined. If you place a minus sign in front of <b>sort_weight</b> , the function returns the last (highest) sorted value instead.
rank	By stating a <b>rank</b> "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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {,fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

## 5 Functions in scripts and chart expressions

### 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 same rank (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
```

## 5 Functions in scripts and chart expressions

```
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 <b>rank</b> is 1, which corresponds to the highest value. By specifying <b>rank</b> as 2, the second highest value is returned. If <b>rank</b> 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;

## 5 Functions in scripts and chart expressions

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.
rank	The default value of <b>rank</b> is 1, which corresponds to the highest value. By specifying <b>rank</b> as 2, the second highest value is returned. If <b>rank</b> 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.

## 5 Functions in scripts and chart expressions

Argument	Description
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### 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.
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.





## 5 Functions in scripts and chart expressions

Examples	Results
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 180)*
-  *RangeMax (page 666)*

### 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 <b>rank</b> is 1, which corresponds to the lowest value. By specifying <b>rank</b> as 2, the second lowest value is returned. If <b>rank</b> is 3, the third lowest value is returned, and so on.

## 5 Functions in scripts and chart expressions

### 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.

## 5 Functions in scripts and chart expressions



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 <b>rank</b> is 1, which corresponds to the lowest value. By specifying <b>rank</b> as 2, the second lowest value is returned. If <b>rank</b> 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	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {,fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### 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

## 5 Functions in scripts and chart expressions

Customer	Product	UnitSales	UnitPrice
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 \(page 180\)](#)
-  [RangeMin \(page 670\)](#)

### 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.

## 5 Functions in scripts and chart expressions

### Scripting examples

Example	Result
<pre>Temp: LOAD * inline [ Customer Product OrderNumber UnitsSales 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;</pre>	<p>MyMostOftenSoldProduct</p> <p>AA</p> <p>because AA is the only product sold more than once.</p>

### 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	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {,fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

## 5 Functions in scripts and chart expressions

### 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
```

## 5 Functions in scripts and chart expressions

```
Betacab|CC|2|20
Betacab|DD||25
Canutility|AA|8|15
Canutility|CC||19
] (delimiter is '|');
```

### See also:

-  [Avg - chart function \(page 234\)](#)
-  [Median - chart function \(page 271\)](#)

### 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;
```



## 5 Functions in scripts and chart expressions

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	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {,fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>



*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

## 5 Functions in scripts and chart expressions

Customer	Product	UnitSales	UnitPrice
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
<code>Only({&lt;UnitPrice={9}&gt;} Product)</code>	BB, because this is the only Product that has a UnitPrice of '9'.
<code>Only({&lt;Product={DD}&gt;} Customer)</code>	B, because the only Customer selling a Product called 'DD'.
<code>Only({&lt;UnitPrice={20}&gt;} UnitSales)</code>	The number of UnitSales where UnitPrice is 20 is 2, because there is only one value of UnitSales where the UnitPrice = 20.
<code>Only({&lt;UnitPrice={15}&gt;} UnitSales)</code>	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 <b>distinct</b> 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 )
```

## 5 Functions in scripts and chart expressions

**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	<p>If the word <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.</p> <div> <i>Although the <b>DISTINCT</b> 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.</i></div>
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

**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

## 5 Functions in scripts and chart expressions

### Examples and results

Examples	Results
<code>Sum(UnitSales)</code>	38. The total of the values in unitSales.
<code>Sum(UnitSales*UnitPrice)</code>	505. The total of unitPrice multiplied by unitSales aggregated.
<code>Sum (TOTAL UnitSales*UnitPrice)</code>	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.  <code>Sum({1} TOTAL UnitSales*UnitPrice)</code>	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)
```

---

## 5 Functions in scripts and chart expressions

---

### 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.

---

## 5 Functions in scripts and chart expressions

---

```
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 <b>distinct</b> 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.

## 5 Functions in scripts and chart expressions

### Scripting examples

Example	Result
<pre>Temp: LOAD * inline [ Customer Product OrderNumber UnitsSales 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;</pre>	<pre>Customer OrdersByCustomer Astrida 3 Betacab 3 Canutility 2 Divadip 2</pre> <p>As long as the dimension Customer is included in the table on the sheet, otherwise the result for OrdersByCustomer is 3, 2.</p>
<p>Given that the <b>Temp</b> table is loaded as in the previous example:</p> <pre>LOAD Count(OrderNumber) as TotalOrderNumber Resident Temp;</pre>	<pre>TotalOrderNumber 10</pre>
<p>Given that the <b>Temp</b> table is loaded as in the first example:</p> <pre>LOAD Count(distinct OrderNumber) as TotalOrderNumber Resident Temp;</pre>	<pre>TotalOrderNumber 8</pre> <p>Because there are two values of OrderNumber with the same value, 1, and one null value.</p>

### 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.



## 5 Functions in scripts and chart expressions

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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>


### 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.

## 5 Functions in scripts and chart expressions

### 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. <div> "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.</div>
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.
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)
```

## 5 Functions in scripts and chart expressions

**Return data type:** integer

**Arguments:**

Arguments

Argument	Description
expr Expression	The expression or field containing the data to be measured.
distinct	If the word <b>distinct</b> 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
<pre>Temp: LOAD * inline [ Customer Product OrderNumber UnitsSales 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;</pre>	<p>Customer MissingOrdersByCustomer Astrida 0 Betacab 1 Canutility 2 Divadip 0</p> <p>The second statement gives:</p> <p>TotalMissingCount 3 in a table with that dimension.</p>
<p>Given that the <b>Temp</b> table is loaded as in the previous example:</p> <pre>LOAD MissingCount(distinct OrderNumber) as TotalMissingCountDistinct Resident Temp;</pre>	<p>TotalMissingCountDistinct 1 Because there is only oneOrderNumber one missing value.</p>

### 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.
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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {,fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>


#### 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

## 5 Functions in scripts and chart expressions

Customer	Product	OrderNumber	UnitSales	Unit Price
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 <div> "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.</div>
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)
```

## 5 Functions in scripts and chart expressions

**Return data type:** integer

**Arguments:**

Arguments

Argument	Description
expr Expression	The expression or field containing the data to be measured.
distinct	If the word <b>distinct</b> 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
<pre>Set NULLINTERPRET = NULL; Temp: LOAD * inline [ Customer Product OrderNumber UnitsSales 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;</pre>	<p>Customer NullOrdersByCustomer Astrida 0 Betacab 0 Canutility 1</p> <p>The second statement gives:</p> <p>TotalNullCount 1 in a table with that dimension, because only one record contains a null value.</p>

### 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)
```

## 5 Functions in scripts and chart expressions

**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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

**Examples and results:**

Examples and results

Example	Result
NullCount ([OrderNumber])	1 because we have introduced a null value using NullInterpret in the inline <b>LOAD</b> 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 <b>distinct</b> 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 <b>Temp</b> 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
```



## 5 Functions in scripts and chart expressions

```
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(LineNumber) 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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {,fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>


## 5 Functions in scripts and chart expressions

### 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

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. <div> "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.</div>
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-numeric, 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 <b>distinct</b> 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
```

## 5 Functions in scripts and chart expressions

```
] (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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.


## 5 Functions in scripts and chart expressions

Argument	Description
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### 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])	<p>10 because all of the 10 fields in Product are text.</p> <div> <i>"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.</i></div>
TextCount ([OrderNumber])	<p>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.</p>

## 5 Functions in scripts and chart expressions

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

---

## 5 Functions in scripts and chart expressions

---

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 222) ([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.

## 5 Functions in scripts and chart expressions

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. Then add the fields listed in the results column to a sheet in your app to see the result.

### Examples and results:

Examples and results

Example	Year	IRR2013
<pre>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;</pre>	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.



## 5 Functions in scripts and chart expressions

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	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {,fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>


### Limitations:

The expression 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 calculated dimensions.

Text values, NULL values and missing values are disregarded.

### Examples and results:

#### Examples and results

Example	Result
IRR (Payments)	<p>0.1634</p> <p>The payments are assumed to be periodic in nature, for example monthly.</p> <div> <i>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.</i></div>

Data used in examples:

CashFlow:



## 5 Functions in scripts and chart expressions

---

```
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 \(page 222\)](#)
-  [Aggr - chart function \(page 367\)](#)

### 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	<b>discount_rate</b> is the rate of discount over the length of the period.
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. Then add the fields listed in the results column to a sheet in your app to see the result.

## 5 Functions in scripts and chart expressions

Examples and results

Example	Year	NPV1_2013
<pre>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;</pre>	2013	-\$540.12

Examples and results

Example	Year	Discount	NPV2_2013
Given that the <b>Cashflow</b> table is loaded as in the previous example:	2013	0.1	-
<pre>LOAD Year,NPV(Discount, Payments) as NPV2_2013 Resident Cashflow Group By Year, Discount;</pre>	2013	0.2	\$3456.05
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.			\$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)
```

## 5 Functions in scripts and chart expressions

**Return data type:** numeric The result has a default number format of money.

### Arguments:

Arguments

Argument	Description
discount_rate	<b>discount_rate</b> is the rate of discount over the length of the period.
value	The expression or field containing the data to be measured.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p> <p>The <b>TOTAL</b> 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.</p>

### 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 calculated dimensions.

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
```

## 5 Functions in scripts and chart expressions

```
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 \(page 224\)](#)
-  [Aggr - chart function \(page 367\)](#)

### 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 <b>date</b> .
date	The expression or field containing the schedule of dates corresponding to the cash flow payments given in <b>pmt</b> .

### 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. Then add the fields listed in the results column to a sheet in your app to see the result.

## 5 Functions in scripts and chart expressions

### Examples and results

Example	Year	XIRR2013
<pre>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;</pre>	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 <b>date</b> .
date	The expression or field containing the schedule of dates corresponding to the cash flow payments given in <b>pmt</b> .
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {,fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

#### 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 calculated dimensions.

## 5 Functions in scripts and chart expressions

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 \(page 216\)](#)
-  [Aggr - chart function \(page 367\)](#)

## 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	<b>discount_rate</b> is the rate of discount over the length of the period.
pmt	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 <b>pmt</b> .

## 5 Functions in scripts and chart expressions

### 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. Then add the fields listed in the results column to a sheet in your app to see the result.

Examples and results

Example	Year	XNPV1_2013
<pre>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;</pre>	2013	\$2104.37

Examples and results

Example	Year	Discount	XNPV2_2013
Given that the <b>Cashflow</b> table is loaded as in the previous example: <pre>LOAD Year,XNPV(Discount, Payments, Date) as XNPV2_2013 Resident CashFlow Group By Year, Discount;</pre> 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)
```



## 5 Functions in scripts and chart expressions

**Return data type:** numeric The result has a default number format of money.

### Arguments:

Arguments

Argument	Description
discount_rate	<b>discount_rate</b> is the rate of discount over the length of the period.
pmt	Payments. The expression or field containing the cash flows corresponding to the payment schedule given in <b>date</b> .
date	The expression or field containing the schedule of dates corresponding to the cash flow payments given in <b>pmt</b> .
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### 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 calculated dimensions.

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 \(page 219\)](#)
-  [Aggr - chart function \(page 367\)](#)

## 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

---

## 5 Functions in scripts and chart expressions

---

$y=mx+b$  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

---

## 5 Functions in scripts and chart expressions

---

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]])
```

---

## 5 Functions in scripts and chart expressions

---

### 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 ( $r^2/(1-r^2)$ ) of a linear regression defined by the equation  $y=mx+b$  for a series of coordinates represented by paired numbers in 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 ]])
```

---

## 5 Functions in scripts and chart expressions

---

### 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 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.	
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.	

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

## 5 Functions in scripts and chart expressions

---

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 expression 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 calculated dimensions. 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` functions is `avg([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)
```

### 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 <b>distinct</b> occurs before the expression, all duplicates will be disregarded.

#### Examples and results:

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

##### Resulting data

Example	Result
<pre>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;</pre>	<pre>Customer MyAverageSalesByCustomer Astrida 48.916667 Betacab 44.916667 Canutility 56.916667 Divadip 63.083333</pre> <p>This can be checked in the sheet by creating a table including the measure: <code>Sum(Sales)/12</code></p>
<p>Given that the <b>Temp</b> table is loaded as in the previous example:</p> <pre>LOAD Customer,Avg(DISTINCT Sales) as MyAvgSalesDistinct Resident Temp Group By Customer;</pre>	<pre>Customer MyAverageSalesByCustomer Astrida 43.1 Betacab 43.909091 Canutility 55.909091 Divadip 61</pre> <p>Only the distinct values are counted. Divide the total by the number of non-duplicate values.</p>

### 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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {, fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

#### Limitations:

The expression 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 calculated dimensions.

#### 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

## 5 Functions in scripts and chart expressions

Function examples

Example	Result
Avg(Sales)	For a table including the dimension Customer and the measure Avg([Sales]), if <b>Totals</b> are shown, the result is 2566.
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 \(page 367\)](#)

### 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)
```

## 5 Functions in scripts and chart expressions

**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. Then add the fields listed in the results column to a sheet in your app to see the result.

Resulting data

Example	Result
<pre>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;</pre>	In a table with the dimension <code>correl_salary</code> , the result of the <code>Correl()</code> 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.

## 5 Functions in scripts and chart expressions

### 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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {, fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

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
<code>Correl(Age, Salary)</code>	For a table including the dimension Employee name and the measure <code>Correl(Age, Salary)</code> , the result is 0.9270611. The result is only displayed for the totals cell.




## 5 Functions in scripts and chart expressions

Example	Result
<code>Correl (TOTAL Age, Salary))</code>	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.
<code>Correl({1} TOTAL Age, Salary))</code>	0.927. Independent of selections. This is because the set expression {1} disregards all selections and dimensions.
<code>Correl (TOTAL &lt;Gender&gt; Age, Salary))</code>	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 \(page 367\)](#)
-  [Avg - chart function \(page 234\)](#)
-  [RangeCorrel \(page 658\)](#)

### 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 242) to calculate the exclusive fractile.

Syntax:

```
Fractile(expr, fraction)
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** numeric

The function returns the value corresponding to the rank as defined by  $\text{rank} = \text{fraction} * (N-1) + 1$  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. Then add the fields listed in the results column to a sheet in your app to see the result.

## 5 Functions in scripts and chart expressions

Resulting data	
Example	Result
<pre>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;</pre>	<p>In a table with the dimensions Type and MyFractile, the results of the Fractile() calculations in the data load script are:</p> <p>Type MyFractile Comparison 27.5 Observation 36</p>

### 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 244) 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  $\text{rank} = \text{fraction} * (N-1) + 1$  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.



## 5 Functions in scripts and chart expressions

### 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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

### 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

## 5 Functions in scripts and chart expressions

### Function examples

Example	Result
Fractile (Sales, 0.75)	For a table including the dimension Customer and the measure Fractile([Sales]), if <b>Totals</b> 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 \(page 367\)](#)

### 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 238) 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  $\text{rank} = \text{fraction} * (N+1)$  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. Then add the fields listed in the results column to a sheet in your app to see the result.

## 5 Functions in scripts and chart expressions

Resulting data	
Example	Result
<pre>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;</pre>	<p>In a table with the dimensions Type and MyFractile, the results of the FractileExc() calculations in the data load script are:</p> <p>Type MyFractile Comparison 28.5 Observation 38</p>

### 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 240) 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  $\text{rank} = \text{fraction} * (N+1)$  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.

## 5 Functions in scripts and chart expressions

### 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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

### 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

## 5 Functions in scripts and chart expressions

Function examples

Example	Result
FractileExc (Sales, 0.75)	For a table including the dimension Customer and the measure FractileExc([Sales]), if <b>Totals</b> 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.
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 \(page 367\)](#)

### 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 <b>distinct</b> occurs before the expression, all duplicates will be disregarded.

**Examples and results:**

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

## 5 Functions in scripts and chart expressions

Resulting data	
Example	Result
<pre>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;</pre>	<p>In a table with the dimensions Type, MyKurtosis1, and MyKurtosis2, the results of the Kurtosis() calculations in the data load script are:</p> <pre>Type MyKurtosis1 MyKurtosis2 Comparison -1.1612957 -1.4982366 Observation -1.1148768 -0.93540144</pre>

### 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)
```



## 5 Functions in scripts and chart expressions

**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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

**Limitations:**

The expression 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 calculated dimensions.

**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	19	39	19	25

## 5 Functions in scripts and chart expressions

Function examples

Example	Result
kurtosis (value)	For a table including the dimension type and the measure kurtosis(value), if <b>Totals</b> 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 \(page 234\)](#)

### 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 ]])
```

## 5 Functions in scripts and chart expressions

**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)	<p>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.</p> <p>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.</p>

**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:**

 [An example of how to use linest functions \(page 285\)](#)

### 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.

## 5 Functions in scripts and chart expressions



Argument	Description
y0_const, x0_const	<p>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.</p> <div> <i>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.</i></div>
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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld }&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

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:

-  [An example of how to use lineest functions \(page 285\)](#)
-  [Avg - chart function \(page 234\)](#)

## 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 ]])
```

## 5 Functions in scripts and chart expressions

---

**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)	<p>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.</p> <p>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.</p>

**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:**

 *An example of how to use linest functions (page 285)*

### 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.

## 5 Functions in scripts and chart expressions



Argument	Description
y0, x0	<p>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.</p> <div> <i>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.</i></div>
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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

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:

-  [An example of how to use lineest functions \(page 285\)](#)
-  [Avg - chart function \(page 234\)](#)

## 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 ]])
```

## 5 Functions in scripts and chart expressions

---

**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)	<p>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.</p> <p>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.</p>

**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:**

 [An example of how to use linest functions \(page 285\)](#)

### LINEST\_F - chart function

**LINEST\_F()** 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 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.

## 5 Functions in scripts and chart expressions



Argument	Description
y0, x0	<p>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.</p> <div> <i>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.</i></div>
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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld }&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

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:

-  [An example of how to use lineest functions \(page 285\)](#)
-  [Avg - chart function \(page 234\)](#)

## 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 ]])
```



## 5 Functions in scripts and chart expressions

**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)	<p>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.</p> <p>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.</p>

**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:**

 *An example of how to use linest functions (page 285)*

### 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.

## 5 Functions in scripts and chart expressions



Argument	Description
y0, x0	<p>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.</p> <div> <i>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.</i></div>
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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

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:

-  [An example of how to use line functions \(page 285\)](#)
-  [Avg - chart function \(page 234\)](#)

## 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 ]])
```

## 5 Functions in scripts and chart expressions

---

**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)	<p>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.</p> <p>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.</p>

**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:**

 *An example of how to use linest functions (page 285)*

### 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.

## 5 Functions in scripts and chart expressions



Argument	Description
y0, x0	<p>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.</p> <div> <i>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.</i></div>
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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld }&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

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:

-  [An example of how to use lineest functions \(page 285\)](#)
-  [Avg - chart function \(page 234\)](#)

## 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 ]])
```

## 5 Functions in scripts and chart expressions

---

**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)	<p>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.</p> <p>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.</p>

**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:**

 *An example of how to use linest functions (page 285)*

### 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.

## 5 Functions in scripts and chart expressions



Argument	Description
y0, x0	<p>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.</p> <div> <i>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.</i></div>
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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld }&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

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:

-  [An example of how to use linest functions \(page 285\)](#)
-  [Avg - chart function \(page 234\)](#)

## 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)	<p>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.</p> <p>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.</p>

**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:**

 [An example of how to use linest functions \(page 285\)](#)

### 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.

## 5 Functions in scripts and chart expressions



Argument	Description
y0, x0	<p>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.</p> <div> <i>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.</i></div>
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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld }&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

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:

-  [An example of how to use lineest functions \(page 285\)](#)
-  [Avg - chart function \(page 234\)](#)

## 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)	<p>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.</p> <p>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.</p>

**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:**

 [An example of how to use linest functions \(page 285\)](#)

### 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.

## 5 Functions in scripts and chart expressions



Argument	Description
y0, x0	<p>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.</p> <div> <i>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.</i></div>
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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld }&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

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:

-  [An example of how to use lineest functions \(page 285\)](#)
-  [Avg - chart function \(page 234\)](#)

## 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 ]])
```

## 5 Functions in scripts and chart expressions

---

**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)	<p>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.</p> <p>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.</p>

**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:**

 *An example of how to use linest functions (page 285)*

### 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.

## 5 Functions in scripts and chart expressions



Argument	Description
y0, x0	<p>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.</p> <div> <i>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.</i></div>
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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld }&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

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:

-  [An example of how to use lineest functions \(page 285\)](#)
-  [Avg - chart function \(page 234\)](#)

## 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 ]])
```

## 5 Functions in scripts and chart expressions

---

**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)	<p>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.</p> <p>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.</p>

**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:**

 *An example of how to use linest functions (page 285)*

### 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.

## 5 Functions in scripts and chart expressions

Argument	Description
y0, x0	<p>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.</p> <div> <i>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.</i></div>
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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>



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 expression 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 calculated dimensions.

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:

-  [An example of how to use line functions \(page 285\)](#)
-  [Avg - chart function \(page 234\)](#)

## 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.

**Examples and results:**

Add the example script to your app and run it. Then build a straight table with `Type` and `MyMedian` as dimensions.

Resulting data

Example	Result
<pre>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 ' ');  Median1: LOAD Type, Median(Value) as MyMedian Resident Table1 Group By Type;</pre>	<p>The results of the Median() calculation are:</p> <ul style="list-style-type: none"><li>• Type is MyMedian</li><li>• Comparison is 2.5</li><li>• observation is 26.5</li></ul>

### Median - chart function

**Median()** returns the median value of the range of values aggregated in the expression iterated over the chart dimensions.

## 5 Functions in scripts and chart expressions

---

### 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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {, fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### Limitations:

The expression 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 calculated dimensions.

### Examples and results:

Add the example script to your app and run it. Then build a straight table with `Type` as dimension and `Median (Value)` as measure.

`TOTALs` should be enabled in the properties of the table.



Example	Result
<pre>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 ' ');</pre>	<p>The median values for:</p> <ul style="list-style-type: none"><li>• Totals is 19</li><li>• Comparison is 2.5</li><li>• observation is 26.5</li></ul>

### See also:

 [Avg - chart function \(page 234\)](#)

### 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 <b>distinct</b> 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
<pre>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;</pre>	<p>The results of the Skew() calculation are:</p> <ul style="list-style-type: none"><li>• <code>Type</code> is MySkew</li><li>• <code>Comparison</code> is 0.86414768</li><li>• <code>observation</code> is 0.32625351</li></ul>

### 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)
```

---

## 5 Functions in scripts and chart expressions

---

**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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

**Limitations:**

The expression 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 calculated dimensions.

**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.

`total`s should be enabled in the properties of the table.

## 5 Functions in scripts and chart expressions

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: <ul style="list-style-type: none"><li>• Total is 0.23522195</li><li>• Comparison is 0.86414768</li><li>• observation is 0.32625351</li></ul>

### See also:

 [Avg - chart function \(page 234\)](#)

### 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 <b>distinct</b> 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
<pre>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;</pre>	<p>The results of the Stdev() calculation are:</p> <ul style="list-style-type: none"><li>• <code>Type</code> is <code>MyStdev</code></li><li>• <code>Comparison</code> is 14.61245</li><li>• <code>observation</code> is 12.507997</li></ul>

### 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)
```

---

## 5 Functions in scripts and chart expressions

---

**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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

**Limitations:**

The expression 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 calculated dimensions.

**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.

`total`s should be enabled in the properties of the table.

Example	Result
<pre>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 ' ');</pre>	<p>The results of the Stdev(Value) calculation are:</p> <ul style="list-style-type: none"><li>• Total is 15.47529</li><li>• Comparison is 14.61245</li><li>• observation is 12.507997</li></ul>

### See also:

- 📄 [Avg - chart function \(page 234\)](#)
- 📄 [STEYX - chart function \(page 283\)](#)

### Sterr

**Sterr()** returns the aggregated standard error ( $\text{stdev}/\sqrt{n}$ ) 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 <b>distinct</b> 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. Then add the fields listed in the results column to a sheet in your app to see the result.

Resulting data	
Example	Result
<pre>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;</pre>	<p>In a table with the dimensions Type and MySterr, the results of the Sterr() calculation in the data load script are:</p> <pre>Type MySterr Comparison 3.2674431 Observation 2.7968733</pre>

### Sterr - chart function

**Sterr()** finds the value of the standard error of the mean, ( $\text{stdev}/\sqrt{n}$ ), for the series of values aggregated in the expression iterated over the chart dimensions.

### Syntax:

```
Sterr ([{SetExpression}] [DISTINCT] [TOTAL [<fld{, fld}>]] expr)
```



---

## 5 Functions in scripts and chart expressions

---

**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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

**Limitations:**

The expression 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 calculated dimensions.

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.

`total1s` should be enabled in the properties of the table.

Example	Result
<pre>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 ' ');</pre>	<p>The results of the Sterr(Value) calculation are:</p> <ul style="list-style-type: none"><li>• Total is 2.4468583</li><li>• Comparison is 3.2674431</li><li>• Observation is 2.7968733</li></ul>

### See also:

- 📄 [Avg - chart function \(page 234\)](#)
- 📄 [STEYX - chart function \(page 283\)](#)

## 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. Then add the fields listed in the results column to a sheet in your app to see the result.

Resulting data

Example	Result
<pre>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;</pre>	<p>In a table with the dimension <code>MySTEYX</code>, the result of the <code>STEYX()</code> calculation in the data load script is 2.0714764.</p>

### 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)
```

## 5 Functions in scripts and chart expressions

---

**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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

**Limitations:**

The expression 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 calculated dimensions.

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
<pre>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 ' ');</pre>	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 \(page 234\)](#)
-  [Sterr - chart function \(page 280\)](#)

### An example 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.

Please refer to the individual linest chart function and script function topics for descriptions of syntax and arguments.


### Loading the sample data

Do the following:



1. Create a new app.
2. In the data load editor, enter the following:

```
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;
```

3. Click  to load the data.

### Displaying the results from the data load script calculations

1. Do the following:  
In the data load editor, click  to go to the app view, create a new sheet and open it.
2. Click  **Edit sheet** to edit the sheet.
3. From **Charts** add a table, and from Fields add the following as columns:
  - Linest\_B
  - Linest\_DF
  - Linest\_F
  - Linest\_M
  - Linest\_R2
  - Linest\_SEB
  - Linest\_SEM
  - Linest\_SEY
  - Linest\_SSREG
  - Linest\_SSRESID

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



## 5 Functions in scripts and chart expressions

Results table

Linest_SEM	Linest_SEY	Linest_SSREG	Linest_SSRESID
1.887	48.666	49235.014	42631.186

### Creating the linest chart function visualizations

Do the following:

1. In the data load editor, click  to go to the app view, create a new sheet and open it.
2. Click  **Edit sheet** to edit the sheet.
3. From **Charts** add a line chart, and from **Fields** add X as a dimension and Sum(Y) as a measure.  
A line chart is create that represents the graph of X plotted against Y, from which the linest functions are calculated.
4. From **Charts** add a table with the following as a dimension:  
ValueList('Linest\_b', 'Linest\_df', 'Linest\_f', 'Linest\_m', 'Linest\_r2', 'Linest\_SEB', 'Linest\_SEM', 'Linest\_SEY', 'Linest\_SSREG', 'Linest\_SSRESID')  
This 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.
5. 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 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.

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

### Statistical test functions

This section describes functions for statistical tests, which are divided into three categories. The 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.

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<sup>2</sup>-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<sup>2</sup>-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])





## 5 Functions in scripts and chart expressions

Chi2Test\_p

**Chi2Test\_p()** returns the aggregated chi<sup>2</sup>-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 \(page 292\)](#)
-  [Z-test functions \(page 326\)](#)

Chi2Test\_chi2

**Chi2Test\_chi2()** returns the aggregated chi<sup>2</sup>-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<sup>2</sup>-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 <b>col</b> and <b>row</b> .
expected_value	The expected value for the distribution at the specified <b>col</b> and <b>row</b> .



**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 \(page 341\)](#)
-  [Examples of how to use chi2-test functions in the data load script \(page 344\)](#)

### 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 <b>col</b> and <b>row</b> .
expected_value	The expected value for the distribution at the specified <b>col</b> and <b>row</b> .



### 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 \(page 341\)](#)
-  [Examples of how to use chi2-test functions in the data load script \(page 344\)](#)

### 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 <b>col</b> and <b>row</b> .
expected_value	The expected value for the distribution at the specified <b>col</b> and <b>row</b> .

### 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  \$\chi^2\$ -test functions in charts \(page 341\)](#)
- [Examples of how to use  \$\chi^2\$ -test functions in the data load script \(page 344\)](#)

### 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 346)*

#### 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.

---

## 5 Functions in scripts and chart expressions

---

**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)

## 5 Functions in scripts and chart expressions

---

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])
```

## 5 Functions in scripts and chart expressions

---

`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.

---

## 5 Functions in scripts and chart expressions

---

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.



## 5 Functions in scripts and chart expressions

**TTestlw\_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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> is set to 0.025, resulting in a 95% confidence interval.
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

### 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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

### TTest\_dif

**TTest\_dif()** is a numeric function that returns the aggregated student's t-test mean difference for two independent series of values.

## 5 Functions in scripts and chart expressions

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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

### 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.

## 5 Functions in scripts and chart expressions

---

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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> is set to 0.025, resulting in a 95% confidence interval.
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

### 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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

### 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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

**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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

**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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> is set to 0.025, resulting in a 95% confidence interval.
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

**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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
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 <b>Type</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> is set to 0.025, resulting in a 95% confidence interval.
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

**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.

## 5 Functions in scripts and chart expressions

---

### Syntax:

```
TTestw_df (weight, grp, value [, eq_var])
```

**Return data type:** numeric

### Arguments:

#### Arguments

Argument	Description
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
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 <b>Type</b> .
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

### 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 <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
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 <b>Type</b> .
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .

### 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 \(page 346\)](#)

### 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 <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
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 <b>Type</b> .
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> is set to 0.025, resulting in a 95% confidence interval.
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 346)*

**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 <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
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 <b>Type</b> .
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

### 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 <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
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 <b>Type</b> .
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

### 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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
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 <b>Type</b> .
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

### 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.

## 5 Functions in scripts and chart expressions

---

### Syntax:

```
TTestw_upper (weight, grp, value [, sig [, eq_var]])
```

**Return data type:** numeric

### Arguments:

#### Arguments

Argument	Description
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
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 <b>Type</b> .
value	The sample values to be evaluated. The sample values must be logically grouped as specified by exactly two values in <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> is set to 0.025, resulting in a 95% confidence interval.
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 346\)](#)

### 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 <b>Value</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> 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 \(page 346\)](#)

### 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 <b>Value</b> .

**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 \(page 346\)](#)

**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 <b>Value</b> .

### 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 \(page 346\)](#)

### 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 <b>Value</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> 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 )
```


## 5 Functions in scripts and chart expressions

---

`TTest1_lower( value, 0.005 )`

---

### See also:

 [Creating a typical t-test report \(page 346\)](#)

### 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 <b>Value</b> .

### 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 \(page 346\)](#)

### 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.

---

## 5 Functions in scripts and chart expressions

---

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 <b>Value</b> .

### 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 \(page 346\)](#)

### 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 <b>Value</b> .

**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 \(page 346\)](#)

**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 <b>Value</b> .

## 5 Functions in scripts and chart expressions

Argument	Description
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> 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 \(page 346\)](#)

### 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 <b>Value</b> .
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> 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 \(page 346\)](#)

### 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 <b>Value</b> .
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .

### 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 \(page 346\)](#)

### 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 <b>Value</b> .
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .

### 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 \(page 346\)](#)

### 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 <b>Value</b> .
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> 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 \(page 346\)](#)

### TTest1w\_sig

**TTest1w\_sig()** returns the aggregated student's t-test 2-tailed level of significance for a series of values.

## 5 Functions in scripts and chart expressions

---

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 <b>Value</b> .
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .

### 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 \(page 346\)](#)

### 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 <b>Value</b> .
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .


**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 \(page 346\)](#)

**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 <b>Value</b> .
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .

**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 \(page 346\)](#)

**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 <b>Value</b> .
weight	Each value in <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> 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 346)*

### 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 349)*

### One column format functions

The following functions apply to z-tests with simple input data series.

## 5 Functions in scripts and chart expressions

---

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.

## 5 Functions in scripts and chart expressions

---

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.



## 5 Functions in scripts and chart expressions

---

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 <b>sigma</b> . If <b>sigma</b> 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 \(page 349\)](#)

### 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])
```

## 5 Functions in scripts and chart expressions

---

**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 <b>sigma</b> . If <b>sigma</b> 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 \(page 349\)](#)

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.

## 5 Functions in scripts and chart expressions

Argument	Description
sigma	If known, the standard deviation can be stated in <b>sigma</b> . If <b>sigma</b> 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 \(page 349\)](#)

### 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 <b>sigma</b> . If <b>sigma</b> 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 \(page 349\)](#)

### 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 <b>sigma</b> . If <b>sigma</b> is omitted the actual sample standard deviation will be used.
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> 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 \(page 349\)](#)

### 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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> is set to 0.025, resulting in a 95% confidence interval.
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 349\)](#)

### 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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> is set to 0.025, resulting in a 95% confidence interval.
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 349\)](#)

### 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 <b>value</b> . 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 <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
sigma	If known, the standard deviation can be stated in <b>sigma</b> . If <b>sigma</b> 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 \(page 349\)](#)

### 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.

## 5 Functions in scripts and chart expressions

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 <b>value</b> . 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 <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
sigma	If known, the standard deviation can be stated in <b>sigma</b> . If <b>sigma</b> 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 \(page 349\)](#)

### 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 <b>value</b> . 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 <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
sigma	If known, the standard deviation can be stated in <b>sigma</b> . If <b>sigma</b> 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 \(page 349\)](#)

**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 <b>value</b> . 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 <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
sigma	If known, the standard deviation can be stated in <b>sigma</b> . If <b>sigma</b> 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 \(page 349\)](#)

**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 <b>value</b> can be counted one or more times according to a corresponding weight value in <b>weight</b> .
sigma	If known, the standard deviation can be stated in <b>sigma</b> . If <b>sigma</b> is omitted the actual sample standard deviation will be used.
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> 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 \(page 349\)](#)

**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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> is set to 0.025, resulting in a 95% confidence interval.
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 349\)](#)

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 <b>group</b> . If a field name for the sample values is not provided in the load script, the field will automatically be named <b>Value</b> .
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 <b>Type</b> .
sig	The two-tailed level of significance can be specified in <b>sig</b> . If omitted, <b>sig</b> is set to 0.025, resulting in a 95% confidence interval.
eq_var	If <b>eq_var</b> is specified as False (0), separate variances of the two samples will be assumed. If <b>eq_var</b> 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 \(page 349\)](#)

### 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 ',');
```

## 5 Functions in scripts and chart expressions



// 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 **3 Significant 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

## 5 Functions in scripts and chart expressions

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 Sense t-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

## 5 Functions in scripts and chart expressions

### 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	<code>if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_conf(Type, Value),TTest_conf(Type, Value, 0))</code>
t	<code>if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_t(Type, Value),TTest_t(Type, Value, 0))</code>
df	<code>if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_df(Type, Value),TTest_df(Type, Value, 0))</code>
Sig. (2-tailed)	<code>if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_sig(Type, Value),TTest_sig(Type, Value, 0))</code>
Mean Difference	<code>TTest_dif(Type, Value)</code>
Standard Error Difference	<code>if(ValueList (Dual('Equal Variance not Assumed', 0), Dual('Equal Variance Assumed', 1)),TTest_sterr(Type, Value),TTest_sterr(Type, Value, 0))</code>
95% Confidence Interval of the Difference (Lower)	<code>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))</code>
95% Confidence Interval of the Difference (Upper)	<code>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))</code>

**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:**

## 5 Functions in scripts and chart expressions

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 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 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 the text 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 the text 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]])
```



## 5 Functions in scripts and chart expressions

**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 <b>sort-weight</b> , if present, with the string corresponding to the lowest value appearing first in the concatenation.
distinct	If the word <b>distinct</b> occurs before the expression, all duplicates are disregarded.

**Examples and results:**

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

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 <b>TeamData</b> 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

## 5 Functions in scripts and chart expressions

Example	Result	Results once added to a sheet
Given that the <b>TeamData</b> table is loaded as in the previous example. Because the argument for <b>sort-weight</b> 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 <b>sort-weight</b> , 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 <b>DISTINCT</b> occurs before the function arguments, duplicates resulting from the evaluation of the function arguments are disregarded.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {, fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

## 5 Functions in scripts and chart expressions

### 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 <b>sort-weight</b> : 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
```

## 5 Functions in scripts and chart expressions

```
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. Then add the fields listed in the results column to a sheet in your app to see the result.

## 5 Functions in scripts and chart expressions

Resulting data

Example	Result	Results on a sheet
<pre>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;</pre>	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.

## 5 Functions in scripts and chart expressions

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
<pre>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;</pre>	SalesGroup  East  West	LastTeamLoaded  Beta  Theta

### MaxString

**MaxString()** finds string values in the expression and returns the last text value sorted 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. Then add the fields listed in the results column to a sheet in your app to see the result.

## 5 Functions in scripts and chart expressions

Example	Result	
<b>TeamData:</b> 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 ' ');  <b>Concat1:</b> LOAD SalesGroup,MaxString(Team) as MaxString1 Resident TeamData Group By SalesGroup;	SalesGroup	MaxString1
	East	Gamma
	West	Zeta
 Given that the <b>TeamData</b> 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	MaxString2
	East	01/11/2013
	West	01/12/2013

### MaxString - chart function

**MaxString()** finds string values in the expression or field and returns the last text value in the text 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.

## 5 Functions in scripts and chart expressions

Argument	Description
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld { .fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

### 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.
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 <code>SET DateFormat='YYYY-MM-DD';</code>

### 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
```



## 5 Functions in scripts and chart expressions

```
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 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. Then add the fields listed in the results column to a sheet in your app to see the result.

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 ' ');	SalesGroup	MinString1
	East	Alpha
	West	Epsilon
Concat1: LOAD SalesGroup,MinString(Team) as MinString1 Resident TeamData Group By SalesGroup;		

## 5 Functions in scripts and chart expressions

Example	Result	
Given that the <b>TeamData</b> table is loaded as in the previous example, and your data load script has the SET statement: <pre>SET DateFormat='DD/MM/YYYY';</pre> <pre>LOAD SalesGroup,MinString(Date) as MinString2 Resident TeamData Group By SalesGroup;</pre>	SalesGroup	MinString2
	East	01/05/2013
	West	01062/2013

### MinString - chart function

**MinString()** finds string values in the expression or field and returns the first text value in the text 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.
TOTAL	<p>If the word <b>TOTAL</b> 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.</p> <p>By using <b>TOTAL [&lt;fld {, fld}&gt;]</b>, where the <b>TOTAL</b> 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.</p>

#### 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

## 5 Functions in scripts and chart expressions

SalesGroup	Amount	MinString(Team)	MinString(Date)
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 <code>SET DateFormat='YYYY-MM-DD';</code>

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.

## 5 Functions in scripts and chart expressions

**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.

## 5 Functions in scripts and chart expressions

Example	Result																																				
<code>=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) ))</code>	<p>This expression takes the values from the created dimension and references them in a nested IF statement as input to three aggregation functions:</p> <table><tr><th colspan="4">ValueList()</th></tr><tr><th>Created dimension</th><th>Year</th><th>Added expression</th><th></th></tr><tr><td></td><td></td><td></td><td>522.00</td></tr><tr><td>Number of Orders</td><td>2012</td><td></td><td>5.00</td></tr><tr><td>Number of Orders</td><td>2013</td><td></td><td>7.00</td></tr><tr><td>Average Order Size</td><td>2012</td><td></td><td>13.20</td></tr><tr><td>Average Order Size</td><td>2013</td><td></td><td>15.43</td></tr><tr><td>Total Amount</td><td>2012</td><td></td><td>66.00</td></tr><tr><td>Total Amount</td><td>2013</td><td></td><td>108.00</td></tr></table>	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
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.

As a general rule, it is not allowed to nest aggregations in a Qlik Sense chart expression. Nesting is only allowed 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))
```

The inclusion of the **TOTAL** qualifier is absolutely necessary for this kind of nesting to be accepted by Qlik Sense, but then again also necessary for the desired comparison. This type of nesting need is quite common and is a good practice.

---

#### See also:

 [Aggr - chart function \(page 367\)](#)

## 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 advanced aggregations, in which the **Aggr** function is enclosed in another 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.

## 5 Functions in scripts and chart expressions

Argument	Description
StructuredParameter	<p>StructuredParameter consists of a dimension and optionally, sorting criteria in the format: (Dimension(Sort-type, Ordering))</p> <p>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.</p> <p>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.</p> <p>For details of how to use sorting criteria, see <a href="#">Adding sorting criteria to the dimension in the structured parameter</a>.</p>
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 <b>distinct</b> 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 <b>nodistinct</b> 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 <b>aggr</b> 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.

## 5 Functions in scripts and chart expressions

---

### Examples and results:

#### Examples and results

Example	Result
<code>Avg(Aggr(Sum (UnitSales*UnitPrice), Customer))</code>	<p>The expression <code>Aggr(Sum(UnitSales*UnitPrice), Customer)</code> finds the total value of sales by <b>Customer</b>, and returns an array of values: 295, 715, and 120 for the three <b>Customer</b> values.</p> <p>Effectively, we have built a temporary list of values without having to create an explicit table or column containing those values.</p> <p>These values are used as input to the <b>Avg()</b> function to find the average value of sales, 376.6667. (You must have <b>Totals</b> selected under <b>Presentation</b> in the properties panel).</p>
<code>Aggr(NODISTINCT Max (UnitPrice), Customer)</code>	<p>An array of values: 16, 16, 16, 25, 25, 25, 19, and 19. The <b>nodistinct</b> qualifier means that the array contains one element for each row in the source data: each is the maximum <b>UnitPrice</b> for each <b>Customer</b> and <b>Product</b>.</p>

### Data used in examples:

Create a table with Customer, Product, UnitPrice, and UnitSales as dimensions. Add the expression to the table, as a measure.

```
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||19
] (delimiter is '|');
```

### Examples and results: Structured parameters

Examples and results: Structured parameters	
Example	Result
<code>Sum(Aggr( Rangesum(Above(Sum(Sales),0,12)), (Year, (Numeric, Ascending)), (Month, (Numeric, Ascending)) ))</code>	<p>This measure calculates the year-to-date sales for every month using sorting criteria in the structured parameter argument in the expression.</p> <p>Without sorting criteria, the result of the expression <code>Sum(Aggr( Rangesum(Above(Sum(Sales),0,12)), (Year), (Month) ))</code> depends on how the dimensions Year and Month are sorted. We may not get the result we want.</p> <p>By adding values for sort type and ordering type to the dimension, we give sorting criteria to the structured parameter: <code>(Year, (Numeric, Ascending)), (Month, (Numeric, Ascending))</code>. The sort type NUMERIC and ordering ASCENDING determine that Year and Month are sorted in ascending numerical order.</p>

Data used in examples:

The following load script generates a table of orders with order lines, to be used in the example for structured parameters.

```
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;
```

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)) ))`

The second measure gives the correct year-to-date sales for each month.

### See also:

 [Basic aggregation functions \(page 175\)](#)

## 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 **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()** 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 values between 0 and 255.

```
RGB (r, g, b)
```

### Color

**Color()** is used in expressions to return the color representation of color number **n** in the chart palette shown in the chart properties. The color representation is a dual value where the text representation comes in the form of 'RGB(r, g, b)' where **r**, **g** and **b** are numbers between 0 and 255 representing the red, green and blue color value respectively. The number representation is an integer representing the red, green and blue components.

```
Color (n)
```

### 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.

---

## 5 Functions in scripts and chart expressions

---

**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  $0 < \text{Value} < 1$  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  $-1 < \text{Value} < 1$  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** factor. 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()** 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 values between 0 and 255.

### Syntax:

```
RGB(r, g, b)
```

## 5 Functions in scripts and chart expressions

**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** factor. 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**.

### 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  $a \leq x < b$  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 ])
```

## 5 Functions in scripts and chart expressions

### 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 and results:

Scripting 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).

## 5 Functions in scripts and chart expressions

Example	Result
<code>alt(Sales,0) + alt(Margin,0)</code>	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  $a \leq x < b$  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.
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 and results:

##### Scripting examples

Example	Result
<code>class( var,10 ) with var = 23</code>	returns '20<=x<30'
<code>class( var,5,'value' ) with var = 23</code>	returns '20<= value <25'
<code>class( var,10,'x',5 ) with var = 23</code>	returns '15<=x<25'

#### Example data 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 source table looks like this:

##### Results table

Name	Age
John	25
Karen	42
Yoshi	53

## 5 Functions in scripts and chart expressions

To add the age group classification field, you can add a preceding load statement using the **class** function. In this example, we load the source table using inline data.

```
LOAD *,
class(Age, 10, 'age') As Agegroup;
```

```
LOAD * INLINE
[ Age, Name
25, John
42, Karen
53, Yoshi];
```

The resulting data that is loaded looks like this:

Results table

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 and results:

Scripting examples

Example	Result
	This expression changes all the NULL values of a field to 'N/A'.

## 5 Functions in scripts and chart expressions

Example	Result
<code>Coalesce(ProductDescription, ProductName, ProductCode, 'no description available')</code>	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'.
<code>Coalesce(TextBetween(FileName, '"', '"'), FileName)</code>	This expression will trim potential enclosing quotes from the field <i>FileName</i> . If the <i>FileName</i> given is quoted, these are removed, and the enclosed, unquoted <i>FileName</i> is returned. If the <i>TextBetween</i> function doesn't find the delimiters it returns null, which the <b>Coalesce</b> rejects, returning instead the raw <i>FileName</i> .

### 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])
```

The if function has three parameters, *condition*, *then* and *else*, which are all expressions. The two other ones, *then* and *else*, can be of any type.

#### Arguments:

##### Arguments

Argument	Description
condition	Expression that is interpreted logically.
then	Expression that can be of any type. If the <i>condition</i> is True, then the if function returns the value of the <i>then</i> expression.
else	<p>Expression that can be of any type. If the <i>condition</i> is False, then the if function returns the value of the <i>else</i> expression.</p> <p>This parameter is optional. If the <i>condition</i> is False, NULL is returned if you have not specified else.</p>

#### Example: Chart expression

Qlik Sense table showing an example of the *if* function.

Example	Result
<code>if( Amount &gt;= 0, 'OK', 'Alarm' )</code>	This expression will test 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.

### Chart expression

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

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:

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
```

## 5 Functions in scripts and chart expressions

[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

### 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: Chart expression

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	<b>match</b> ( Cities,'Toronto','Boston','Beijing','Zurich')	<b>match</b> ( Cities,'Toronto','Boston','Beijing','Stockholm','zurich')
Beijing	3	3
Boston	2	2
Stockholm	0	4

## 5 Functions in scripts and chart expressions

Cities	<code>match(Cities,'Toronto','Boston','Beijing','Zurich')</code>	<code>match(Cities,'Toronto','Boston','Beijing','Stockholm','zurich')</code>
Toronto	1	1
zurich	0	5

Example: Chart expression

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.



## 5 Functions in scripts and chart expressions

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

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.

In the **Data load editor**, create a new section called Load, and then add the following:

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 statement 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

### 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 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: Chart expression

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	<b>mixmatch</b> ( Cities,'Toronto','Boston','Beijing','Zurich')	<b>mixmatch</b> ( Cities,'Toronto','Boston','Beijing','Stockholm','Zurich')
Beijing	3	3
Boston	2	2
Stockholm	0	4
Toronto	1	1
zurich	4	5

#### Example: Chart expression

You can use mixmatch to perform a custom sort for an expression.

## 5 Functions in scripts and chart expressions

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:  
`=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

---

## 5 Functions in scripts and chart expressions

---

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.

In the **Data load editor**, create a new section called Load, and then add the following:

```
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

Color Code Black, Blue, blue	Customer
blue	5646471

### pick

The pick function returns the *n*:th expression in the list.

#### Syntax:

```
pick(n, expr1[ , expr2,...exprN])
```

#### Arguments:

##### Arguments

Argument	Description
n	<i>n</i> is an integer between 1 and N.

#### Examples and results:

##### Scripting examples

Example	Result
<code>pick( N, 'A','B',4, 6 )</code>	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: Chart expression

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.

## 5 Functions in scripts and chart expressions

Qlik Sense table showing examples of the *wildmatch* function in a chart expression

Cities	wildmatch( Cities,'Tor*','?ton','Beijing','*urich') h')	wildmatch( Cities,'Tor*','???ton','Beijing','Stockholm','*urich') h')
Beijing	3	3
Boston	0	2
Stockholm	0	4
Toronto	1	1
zurich	4	5

Example: Chart expression

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

## 5 Functions in scripts and chart expressions

Cities
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

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.

In the **Data load editor**, create a new section called Load, and then add the following:

```
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', 2 for 'Blue', 3 for 'blue', 4 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,'B|*', '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

## 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})
```



## 5 Functions in scripts and chart expressions

### 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 function 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 <b>autonumber</b> function is used on different keys within the script, the optional parameter <i>AutoID</i> 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:

## 5 Functions in scripts and chart expressions

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
```

## 5 Functions in scripts and chart expressions

```
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

## 5 Functions in scripts and chart expressions

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;
```

## 5 Functions in scripts and chart expressions

---

```
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:

## 5 Functions in scripts and chart expressions

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



Day	Date
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( ),
```

## 5 Functions in scripts and chart expressions

```
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.



*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:

Arguments	
Argument	Description
TOTAL	If the table is one-dimensional or if the qualifier <b>TOTAL</b> is used as argument, the current column segment is always equal to the entire column.

## 5 Functions in scripts and chart expressions

### Examples and results:

Data			
Customer	UnitSales	Row in Segment	Row Number
Astrida	4	1	1
Astrida	10	2	2
Astrida	9	3	3
Betacab	5	1	4
Betacab	2	2	5
Betacab	25	3	6
Canutility	8	1	7
Canutility		2	8
Divadip	4	1	9
Divadip		2	10

### Examples and results

Examples	Results
Create a visualization consisting of a table with the dimensions <b>Customer</b> , <b>UnitSales</b> , and add <code>RowNo( )</code> and <code>RowNo(TOTAL)</code> as measures labeled <b>Row in Segment</b> and <b>Row Number</b> .	The <b>Row in Segment</b> 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 <b>Row Number</b> column disregards the dimensions can be used to count the rows in the table.
Add the expression: <code>IF( RowNo( )=1, 0, Unitsales / Above( Unitsales ) )</code> as a measure.	This expression returns 0 for the first row in each column segment, so the column will show:  0, 2.25, 1.1111111, 0, 2.5, 5, 0, 2.375, 0, and 4.

### 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 '|');
```

---

### See also:

 [Above - chart function \(page 605\)](#)

## 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.

---

## 5 Functions in scripts and chart expressions

---

**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 from the system clock 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 system clock and Windows time 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**



---

## 5 Functions in scripts and chart expressions

---

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, tertial, 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, tertial, 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.

---

## 5 Functions in scripts and chart expressions

---

```
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]])
```

---

## 5 Functions in scripts and chart expressions

---

### 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, tertial, or half-year containing a base date. It is also possible to find the timestamp for a previous or following time period.

---

## 5 Functions in scripts and chart expressions

---

```
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, tertial, 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, tertial, 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

---

## 5 Functions in scripts and chart expressions

---

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})
```

## 5 Functions in scripts and chart expressions

### 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.

```
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'

## 5 Functions in scripts and chart expressions

Example	Result
<code>addmonths ('2003-01-29',3,0)</code>	returns '2003-04-29'
<code>addmonths ('2003-01-29',3,1)</code>	returns '2003-04-28'
<code>addmonths ('2003-01-29',1,0)</code>	returns '2003-02-28'
<code>addmonths ('2003-01-29',1,1)</code>	returns '2003-02-26'
<code>addmonths ('2003-02-28',1,0)</code>	returns '2003-03-28'
<code>addmonths ('2003-02-28',1,1)</code>	returns '2003-03-31'
<code>addmonths ('2003-01-29',-3)</code>	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
<code>addyears ('2010-01-29',3)</code>	returns '2013-01-29'
<code>addyears ('2010-01-29',-1)</code>	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.

## 5 Functions in scripts and chart expressions

---

**Return data type:** numeric

**Arguments:**

Arguments

Argument	Description
<b>timestamp</b>	The timestamp, or expression resolving to a timestamp, up to which to calculate the completed number of years.
<b>date_of_birth</b>	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
<code>age('25/01/2014', '29/10/2012')</code>	Returns 1.
<code>age('29/10/2014', '29/10/2012')</code>	Returns 2.

**Example:**

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

```
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;
```



## 5 Functions in scripts and chart expressions

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
<b>timestamp</b>	The timestamp, or expression resolving to a timestamp, to convert.

## 5 Functions in scripts and chart expressions

Argument	Description
<b>place</b>	<p>A place or timezone from the table of valid places and timezones below. Alternatively, you can use GMT or UTC to define the local time. The following values and time offset ranges are valid:</p> <ul style="list-style-type: none"><li>• GMT</li><li>• GMT-12:00 - GMT-01:00</li><li>• GMT+01:00 - GMT+14:00</li><li>• UTC</li><li>• UTC-12:00 - UTC-01:00</li><li>• UTC+01:00 - UTC+14:00</li></ul> <div> <i>You can only use standard time offsets. It's not possible to use an arbitrary time offset, for example, GMT-04:27.</i></div>
<b>ignore_dst</b>	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

## 5 Functions in scripts and chart expressions

A-C	D-K	L-R	S-Z
Bangkok	Harare	Monrovia	Stockholm
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	-	-	-

## 5 Functions in scripts and chart expressions

Examples and results:

Scripting examples

Example	Result
<code>ConvertToLocalTime('2007-11-10 23:59:00', 'Paris')</code>	Returns '2007-11-11 00:59:00' and the corresponding internal timestamp representation.
<code>ConvertToLocalTime(UTC(), 'GMT-05:00')</code>	Returns the time for the North American east coast, for example, New York.
<code>ConvertToLocalTime(UTC(), 'GMT-05:00', True)</code>	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.

**Syntax:**

```
day (expression)
```

**Return data type:** integer

Examples and results:

Scripting examples

Example	Result
<code>day( '1971-10-12' )</code>	returns 12
<code>day( '35648' )</code>	returns 6, because 35648 = 1997-08-06

### 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]])
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>time</b>	The timestamp to evaluate.
<b>period_no</b>	<b>period_no</b> is an integer, or expression that resolves to an integer, where the value 0 indicates the day that contains <b>time</b> . Negative values in <b>period_no</b> indicate preceding days and positive values indicate succeeding days.
<b>day_start</b>	To specify days not starting at midnight, indicate an offset as a fraction of a day in <b>day_start</b> . For example, 0.125 to denote 3 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
dayend('25/01/2013 16:45:00')	Returns 25/01/2013 23:59:59.
dayend('25/01/2013 16:45:00', -1)	Returns '24/01/2013 23:59:59.
dayend('25/01/2013 16:45:00', 0, 0.5)	Returns 26/01/2013 11:59:59.

**Example:**

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

This example finds the timestamp that marks the end 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
```

## 5 Functions in scripts and chart expressions

---

```
];
```

```
InvoiceData:
LOAD *,
DayEnd(InvDate, 1) AS DEnd
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the dayend() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	DEnd
28/03/2012	29/03/2012 23:59:59
10/12/2012	11/12/2012 23:59:59
5/2/2013	07/02/2013 23:59:59
31/3/2013	01/04/2013 23:59:59
19/5/2013	20/05/2013 23:59:59
15/9/2013	16/09/2013 23:59:59
11/12/2013	12/12/2013 23:59:59
2/3/2014	03/03/2014 23:59:59
14/5/2014	15/05/2014 23:59:59
13/6/2014	14/06/2014 23:59:59
7/7/2014	08/07/2014 23:59:59
4/8/2014	05/08/2014 23:59:59

### 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**.

## 5 Functions in scripts and chart expressions

### Syntax:

```
DayName (time[, period_no [, day_start]])
```

**Return data type:** dual

### Arguments:

#### Arguments

Argument	Description
<b>time</b>	The timestamp to evaluate.
<b>period_no</b>	<b>period_no</b> is an integer, or expression that resolves to an integer, where the value 0 indicates the day that contains <b>time</b> . Negative values in <b>period_no</b> indicate preceding days and positive values indicate succeeding days.
<b>day_start</b>	To specify days not starting at midnight, indicate an offset as a fraction of a day in <b>day_start</b> . For example, 0.125 to denote 3 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. Then add the fields listed in the results column to a sheet in your app to see the result.

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 Functions in scripts and chart expressions

---

```
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
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.

#### Syntax:

```
DayNumberOfQuarter (timestamp[, start_month])
```

**Return data type:** integer

The function always uses years based on 366 days.



## 5 Functions in scripts and chart expressions

---

### Arguments:

Arguments

Argument	Description
<b>timestamp</b>	The date to evaluate.
<b>start_month</b>	By specifying a <b>start_month</b> 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 <b>start_month</b> = 3.

### 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
<code>DayNumberOfQuarter('12/09/2014')</code>	Returns 74, the day number of the current quarter.
<code>DayNumberOfQuarter('12/09/2014',3)</code>	Returns 12, the day number of the current quarter. In this case, the first quarter starts with March (because <b>start_month</b> is specified as 3). This means that the current quarter is the third quarter, which started on September 1.

### Example:

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

```
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 *,
DayNumberOfQuarter(StartDate,4) As DayNrQtr
Resident ProjectTable;
Drop table ProjectTable;
```

The resulting table shows the returned values of `DayNumberOfQuarter` for each of the records in the table.

## 5 Functions in scripts and chart expressions

Results table

InvID	StartDate	DayNrQtr
1	28/03/2014	88
2	10/12/2014	71
3	5/2/2015	36
4	31/3/2015	91
5	19/5/2015	49
6	15/9/2015	77

### 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

The function always uses years based on 366 days.

#### Arguments:

Arguments

Argument	Description
<b>timestamp</b>	The date to evaluate.
<b>start_month</b>	By specifying a <b>start_month</b> 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 <b>start_month</b> = 3.

#### 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
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:

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

```
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 *,
DayNumberOfYear(StartDate,4) As DayNrYear
Resident ProjectTable;
Drop table ProjectTable;
```

The resulting table shows the returned values of DayNumberOfYear for each of the records in the table.

Results table

InvID	StartDate	DayNrYear
1	28/03/2014	363
2	10/12/2014	254
3	5/2/2015	311
4	31/3/2015	366
5	19/5/2015	49
6	15/9/2015	168

### 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]])
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>time</b>	The timestamp to evaluate.
<b>period_no</b>	<b>period_no</b> is an integer, or expression that resolves to an integer, where the value 0 indicates the day that contains <b>time</b> . Negative values in <b>period_no</b> indicate preceding days and positive values indicate succeeding days.
<b>day_start</b>	To specify days not starting at midnight, indicate an offset as a fraction of a day in <b>day_start</b> . For example, 0.125 to denote 3 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. Then add the fields listed in the results column to a sheet in your app to see the result.

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
```

## 5 Functions in scripts and chart expressions

---

```
];
```

```
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} )
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** integer

**Arguments:**

Arguments

Argument	Description
<b>end_date</b>	The timestamp of end date to evaluate.
<b>no_of_workdays</b>	The number of working days to achieve.
<b>holiday</b>	<p>Holiday periods to exclude from working days. A holiday period is stated as a start date and an end date, separated by commas.</p> <p><b>Example:</b> '25/12/2013', '26/12/2013'</p> <p>You can specify more than one holiday period, separated by commas.</p> <p><b>Example:</b> '25/12/2013', '26/12/2013', '31/12/2013', '01/01/2014'</p>

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. Then add the fields listed in the results column to a sheet in your app to see the result.

```
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 *,
```

## 5 Functions in scripts and chart expressions

---

```
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 system clock and Windows time 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)
```

## 5 Functions in scripts and chart expressions

**Return data type:** integer

Examples and results:

Scripting examples

Example	Result
hour( '09:14:36' )	returns 9
hour( '0.5555' )	returns 13 ( Because 0.5555 = 13:19:55 )

### inday

This function returns True if **timestamp** lies inside the day containing **base\_timestamp**.

**Syntax:**

**InDay** (timestamp, base\_timestamp, period\_no[, day\_start])

**Return data type:** Boolean

**Arguments:**

Arguments

Argument	Description
<b>timestamp</b>	The date and time that you want to compare with <b>base_timestamp</b> .
<b>base_timestamp</b>	Date and time that is used to evaluate the timestamp.
<b>period_no</b>	The day can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the day which contains <b>base_timestamp</b> . Negative values in <b>period_no</b> indicate preceding days and positive values indicate succeeding days.
<b>day_start</b>	If you want to work with days not starting midnight, indicate an offset as a fraction of a day in <b>day_start</b> , For example, 0.125 to denote 3 AM.

Examples and results:

Scripting examples

Example	Result
inday ( '12/01/2006 12:23:00', '12/01/2006 00:00:00', 0)	Returns True
inday ( '12/01/2006 12:23:00', '13/01/2006 00:00:00', 0)	Returns False
inday ( '12/01/2006 12:23:00', '12/01/2006 00:00:00', -1)	Returns False
inday ( '11/01/2006 12:23:00', '12/01/2006 00:00:00', -1)	Returns True



## 5 Functions in scripts and chart expressions

Example	Result
<code>inDay ('12/01/2006 12:23:00', '12/01/2006 00:00:00', 0, 0.5)</code>	Returns False
<code>inDay ('12/01/2006 11:23:00', '12/01/2006 00:00:00', 0, 0.5)</code>	Returns True

### Example:

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

This example checks if an invoice date falls at any time in the day starting with the `base_timestamp`.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvTime
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 *,
InDay(InvTime, '28/03/2012 00:00:00', 0) AS InDayEx
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the `inDay()` function.

Results table

InvTime	InDayEx
28/03/2012	-1 (True)
10/12/2012	0 (False)
5/2/2013	0 (False)
31/3/2013	0 (False)
19/5/2013	0 (False)
15/9/2013	0 (False)
11/12/2013	0 (False)

## 5 Functions in scripts and chart expressions

InvTime	InDayEx
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)

### 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])
```

**Return data type:** Boolean

#### Arguments:

##### Arguments

Argument	Description
<b>timestamp</b>	The date and time that you want to compare with <b>base_timestamp</b> .
<b>base_timestamp</b>	Date and time that is used to evaluate the timestamp.
<b>period_no</b>	The day can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the day which contains <b>base_timestamp</b> . Negative values in <b>period_no</b> indicate preceding days and positive values indicate succeeding days.
<b>day_start</b>	(optional) If you want to work with days not starting midnight, indicate an offset as a fraction of a day in <b>day_start</b> . For example, 0.125 to denote 3 AM.

Examples and results:

##### Scripting examples

Example	Result
<code>indaytotime ('12/01/2006 12:23:00', '12/01/2006 23:59:00', 0)</code>	Returns True
<code>indaytotime ('12/01/2006 12:23:00', '12/01/2006 00:00:00', 0)</code>	Returns False
<code>indaytotime ('11/01/2006 12:23:00', '12/01/2006 23:59:00', -1)</code>	Returns True

## 5 Functions in scripts and chart expressions

---

### Example:

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

This example checks if an invoice timestamp falls before 17:00:00 on the day starting with the base\_timestamp.

```
TempTable:
LOAD RecNo() as InvID, * Inline [
InvTime
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 *,
InDayToTime(InvTime, '28/03/2012 17:00:00', 0) AS InDayExTT
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the indaytotime() function.

Results table

InvTime	InDayExTT
28/03/2012	-1 (True)
10/12/2012	0 (False)
5/2/2013	0 (False)
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)

## 5 Functions in scripts and chart expressions

InvTime	InDayExTT
7/7/2014	0 (False)
4/8/2014	0 (False)

### 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
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the lunar week.
<b>period_no</b>	The lunar week can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the lunar week which contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding lunar weeks and positive values indicate succeeding lunar weeks.
<b>first_week_day</b>	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
<code>inlunarweek ('12/01/2013', '14/01/2013', 0)</code>	Returns True. Because the value of timestamp, 12/01/2013 falls in the week 08/01/2013 to 14/01/2013.
<code>inlunarweek ('12/01/2013', '07/01/2013', 0)</code>	Returns False. Because the base_date 07/01/2013 is in the lunar week defined as 01/01/2013 to 07/01/2013.

## 5 Functions in scripts and chart expressions

Example	Result
<code>inlunarweek ( '12/01/2013', '14/01/2013', -1)</code>	Returns False. Because specifying a value of <code>period_no</code> as -1 shifts the week to the previous week, 01/01/2013 to 07/01/2013.
<code>inlunarweek ( '07/01/2013', '14/01/2013', -1)</code>	Returns True. In comparison with the previous example, the timestamp is in the week after taking into account the shift backwards.
<code>inlunarweek ( '11/01/2006', '08/01/2006', 0, 3)</code>	Returns False. Because specifying a value for <code>first_week_day</code> as 3 means the start of the year is calculated from 04/01/2013, and so the value of <code>base_date</code> falls in the first week, and the value of <code>timestamp</code> falls in the week 11/01/2013 to 17/01/2013.

### Example:

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

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 `InvDate` 5/2/2013 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.

## 5 Functions in scripts and chart expressions

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)

### inlunarweektoday

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])
```

**Return data type:** Boolean

#### Arguments:

Arguments

Argument	Description
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the lunar week.
<b>period_no</b>	The lunar week can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the lunar week which contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding lunar weeks and positive values indicate succeeding lunar weeks.
<b>first_week_day</b>	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.

## 5 Functions in scripts and chart expressions

Examples and results:

### Scripting examples

Example	Result
<code>inlunarweektodate('12/01/2013', '13/01/2013', 0)</code>	Returns True. Because the value of timestamp, 12/01/2013 falls in the part of the week 08/01/2013 to 13/01/2013.
<code>inlunarweektodate('12/01/2013', '11/01/2013', 0)</code>	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.
<code>inlunarweektodate('12/01/2006', '05/01/2006', 1)</code>	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. Then add the fields listed in the results column to a sheet in your app to see the result.

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 InLWeek2DP1us4  
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.

## 5 Functions in scripts and chart expressions

---

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[, first_month_of_year])
```

**Return data type:** Boolean

#### Arguments:

Arguments

Argument	Description
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the month.
<b>period_no</b>	The month can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the month which contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding months and positive values indicate succeeding months.
<b>first_month_of_year</b>	The <b>first_month_of_year</b> parameter is disabled and reserved for future use.



## 5 Functions in scripts and chart expressions

Examples and results:

Scripting examples

Example	Result
<code>inmonth ('25/01/2013', '01/01/2013', 0 )</code>	Returns True
<code>inmonth('25/01/2013', '01/04/2013', 0)</code>	Returns False
<code>inmonth ('25/01/2013', '01/01/2013', -1)</code>	Returns False
<code>inmonth ('25/12/2012', '01/01/2013', -1)</code>	Returns True

### Example:

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

This example checks if an invoice date falls at any time in the fourth month after the month 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 *,  
  InMonth(InvDate, '31/01/2013', 4) AS InMthPlus4  
Resident TempTable;  
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the `inmonth()` function.

Results table

InvDate	InMthPlus4
28/03/2012	0 (False)
10/12/2012	0 (False)
5/2/2013	0 (False)

## 5 Functions in scripts and chart expressions

InvDate	InMthPlus4
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)

### inmonths

This function finds if a timestamp falls within the same month, bi-month, quarter, tertial, 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])
```

**Return data type:** Boolean

#### Arguments:

##### Arguments

Argument	Description
<b>n_months</b>	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 (tertial), or 6 (half year).
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the period.
<b>period_no</b>	The period can be offset by <b>period_no</b> , an integer, or expression resolving to an integer, where the value 0 indicates the period that contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding periods and positive values indicate succeeding periods.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

## 5 Functions in scripts and chart expressions

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
<code>inmonths(4, '25/01/2013', '25/04/2013', 0)</code>	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.
<code>inmonths(4, '25/05/2013', '25/04/2013', 0)</code>	Returns False. Because 25/05/2013 is outside the same period as the previous example.
<code>inmonths(4, '25/11/2012', '01/02/2013', -1 )</code>	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.
<code>inmonths( 4, '25/05/2006', '01/03/2006', 0, 3)</code>	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. Then add the fields listed in the results column to a sheet in your app to see the result.

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
```

## 5 Functions in scripts and chart expressions

```
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)
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, tertial, 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
])
```

## 5 Functions in scripts and chart expressions

**Return data type:** Boolean

**Arguments:**

Arguments

Argument	Description
<b>n_months</b>	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 <code>inmonth()</code> function), 2 (bi-month), 3 (equivalent to the <code>inquarter()</code> function), 4 (terial), or 6 (half year).
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the period.
<b>period_no</b>	The period can be offset by <b>period_no</b> , an integer, or expression resolving to an integer, where the value 0 indicates the period that contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding periods and positive values indicate succeeding periods.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

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
<code>inmonthstodate(4, '25/01/2013', '25/04/2013', 0)</code>	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 <b>base_date</b> , 25/04/2013 lies.
<code>inmonthstodate(4, '26/04/2013', '25/04/2006', 0)</code>	Returns False. Because 26/04/2013 is outside the same period as the previous example.
<code>inmonthstodate(4, '25/09/2005', '01/02/2006', -1)</code>	Returns True. Because the value of <b>period_no</b> , -1, shifts the search period back one period of four months (the value of <b>n_months</b> ), which makes the search period 01/09/2005 to 01/02/2006.
<code>inmonthstodate(4, '25/04/2006', '01/06/2006', 0, 3)</code>	Returns True. Because the value of <b>first_month_of_year</b> 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.

## 5 Functions in scripts and chart expressions

---

### Example:

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

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)

## 5 Functions in scripts and chart expressions

InvDate	InMths2DPlus4
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:

##### Arguments

Argument	Description
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the month.
<b>period_no</b>	The month can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the month which contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding months and positive values indicate succeeding months.

#### Examples and results:

##### Scripting examples

Example	Result
<code>inmonthtodate ('25/01/2013', '25/01/2013', 0)</code>	Returns True
<code>inmonthtodate ('25/01/2013', '24/01/2013', 0)</code>	Returns False
<code>inmonthtodate ('25/01/2013', '28/02/2013', -1)</code>	Returns True

#### Example:

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

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**.

## 5 Functions in scripts and chart expressions

---

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', 0, 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])
```



## 5 Functions in scripts and chart expressions

---

**Return data type:** Boolean

**Arguments:**

Arguments

Argument	Description
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the quarter.
<b>period_no</b>	The quarter can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the quarter which contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding quarters and positive values indicate succeeding quarters.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

**Examples and results:**

Scripting examples

Example	Result
<code>inquarter ('25/01/2013', '01/01/2013', 0)</code>	Returns True
<code>inquarter ('25/01/2013', '01/04/2013', 0)</code>	Returns False
<code>inquarter ('25/01/2013', '01/01/2013', -1)</code>	Returns False
<code>inquarter ('25/12/2012', '01/01/2013', -1)</code>	Returns True
<code>inquarter ('25/01/2013', '01/03/2013', 0, 3)</code>	Returns False
<code>inquarter ('25/03/2013', '01/03/2013', 0, 3)</code>	Returns True

**Example:**

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

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
```

## 5 Functions in scripts and chart expressions

---

```
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])
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** Boolean

**Arguments:**

Arguments

Argument	Description
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the quarter.
<b>period_no</b>	The quarter can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the quarter which contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding quarters and positive values indicate succeeding quarters.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

**Examples and results:**

Scripting examples

Example	Result
<code>inquartertodate ('25/01/2013', '25/01/2013', 0)</code>	Returns True
<code>inquartertodate ( '25/01/2013', '24/01/2013', 0)</code>	Returns False
<code>inquartertodate ('25/01/2012', '01/02/2013', -1)</code>	Returns True

**Example:**

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

This example checks if an invoice date falls in a fiscal year specified by setting the value of `first_month_of_year` to 4, and in the fourth quarter, before the end of 28/02/2013.

```
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
```

## 5 Functions in scripts and chart expressions

---

```
7/7/2014  
4/8/2014  
];
```

```
InvoiceData:  
LOAD *,  
InQuarterToDate(InvDate, '28/02/2013', 0, 4) AS Qtr42Date  
Resident TempTable;  
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the `inquartertodate()` function.

Results table

InvDate	Qtr42Date
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)

### inweek

This function returns True if **timestamp** lies inside the week containing **base\_date**.

#### Syntax:

```
InWeek (timestamp, base_date, period_no[, first_week_day])
```

## 5 Functions in scripts and chart expressions

**Return data type:** Boolean

**Arguments:**

Arguments

Argument	Description
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the week.
<b>period_no</b>	The week can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the week which contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding weeks and positive values indicate succeeding weeks.
<b>first_week_day</b>	By default, the first day of the week is Monday, starting at midnight between Sunday and Monday. To indicate the week starting on another day, specify an offset in <b>first_week_day</b> . This may be given as a whole number of days and/or fractions of a day.

**Examples and results:**

Scripting examples

Example	Result
<code>inweek ('12/01/2006', '14/01/2006', 0)</code>	Returns True
<code>inweek ('12/01/2006', '20/01/2006', 0 )</code>	Returns False
<code>inweek ('12/01/2006', '14/01/2006', -1 )</code>	Returns False
<code>inweek ('07/01/2006', '14/01/2006', -1)</code>	Returns True
<code>inweek ('12/01/2006', '09/01/2006', 0, 3)</code>	Returns False Because <b>first_week_day</b> 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. Then add the fields listed in the results column to a sheet in your app to see the result.

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
```

## 5 Functions in scripts and chart expressions

```
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 `InvDate` 5/2/2013 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)

### inweektoday

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])
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** Boolean

**Arguments:**

Arguments

Argument	Description
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the week.
<b>period_no</b>	The week can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the week which contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding weeks and positive values indicate succeeding weeks.
<b>first_week_day</b>	By default, the first day of the week is Monday, starting at midnight between Sunday and Monday. To indicate the week starting on another day, specify an offset in <b>first_week_day</b> . This may be given as a whole number of days and/or fractions of a day.

**Examples and results:**

Scripting examples

Example	Result
<code>inweektodate ('12/01/2006', '12/01/2006', 0)</code>	Returns True
<code>inweektodate ('12/01/2006', '11/01/2006', 0)</code>	Returns False
<code>inweektodate ('12/01/2006', '18/01/2006', -1)</code>	Returns False Because <b>period_no</b> is specified as -1, the effective data that timestamp is measured against is 11/01/2006.
<code>inweektodate ('11/01/2006', '12/01/2006', 0, 3 )</code>	Returns False Because <b>first_week_day</b> is specified as 3 (Thursday), which makes 12/01/2006 the first day of the week following the week containing 12/01/2006.

**Example:**

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

This example checks if an invoice date falls during the fourth week after the week in **base\_date**, by specifying **period\_no** as 4, but before the value of **base\_date**.

TempTable:

## 5 Functions in scripts and chart expressions

---

```
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 *,  
  InWeekToDate(InvDate, '11/01/2013', 4) AS InWeek2DPlus4  
Resident TempTable;  
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the inweek() function.

Results table

InvDate	InWeek2DPlus4
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)

### 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])
```



## 5 Functions in scripts and chart expressions

---

**Return data type:** Boolean

**Arguments:**

Arguments

Argument	Description
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the year.
<b>period_no</b>	The year can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the year that contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding years, and positive values indicate succeeding years.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

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
<code>inyear ('25/01/2013', '01/01/2013', 0 )</code>	Returns True
<code>inyear ('25/01/2012', '01/01/2013', 0)</code>	Returns False
<code>inyear ('25/01/2013', '01/01/2013', -1)</code>	Returns False
<code>inyear ('25/01/2012', '01/01/2013', -1 )</code>	Returns True
<code>inyear ('25/01/2013', '01/01/2013', 0, 3)</code>	Returns True  The value of <b>base_date</b> and <b>first_month_of_year</b> specify that <b>timestamp</b> must fall within 01/03/2012 and 28/02/2013
<code>inyear ('25/03/2013', '01/07/2013', 0, 3 )</code>	Returns True

**Example:**

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

## 5 Functions in scripts and chart expressions

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)
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

#### Arguments:

##### Arguments

Argument	Description
<b>timestamp</b>	The date that you want to compare with <b>base_date</b> .
<b>base_date</b>	Date that is used to evaluate the year.
<b>period_no</b>	The year can be offset by <b>period_no</b> . <b>period_no</b> is an integer, where the value 0 indicates the year that contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding years, and positive values indicate succeeding years.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

#### Examples and results:

##### Scripting examples

Example	Result
<code>inyeartodate ('2013/01/25', '2013/02/01', 0)</code>	Returns True
<code>inyeartodate ('2012/01/25', '2013/01/01', 0)</code>	Returns False
<code>inyeartodate ('2012/01/25', '2013/02/01', -1)</code>	Returns True
<code>inyeartodate ('2012/11/25', '2013/01/31', 0, 4)</code>	Returns True The value of timestamp falls inside the fiscal year beginning in the fourth month and before the value of base_date.

## 5 Functions in scripts and chart expressions

Example	Result
<code>inyeartodate ( '2013/3/31', '2013/01/31', 0, 4 )</code>	Returns False Compared with the previous example, the value of timestamp is still inside the fiscal year, but it is after the value of <code>base_date</code> , so it falls outside the part of the year.

### Example:

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

This example checks if an invoice date falls in a fiscal year specified by setting the value of `first_month_of_year` to 4, and in the part of the year before the end of 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 *,
InYearToDate(InvDate, '31/01/2013', 0, 4) AS FinYr2Date
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the `inyeartodate()` function.

Results table

InvDate	FinYr2Date
28/03/2012	0 (False)
10/12/2012	-1 (True)
5/2/2013	0 (False)
31/3/2013	0 (False)
19/5/2013	0 (False)

## 5 Functions in scripts and chart expressions

InvDate	FinYr2Date
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)

### 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})
```

Return data type: dual

#### Arguments:

##### Arguments

Argument	Description
<b>start_date</b>	The start date to evaluate.
<b>no_of_workdays</b>	The number of working days to achieve.
<b>holiday</b>	<p>Holiday periods to exclude from working days. A holiday period is stated as a start date and an end date, separated by commas.</p> <p><b>Example:</b> '25/12/2013', '26/12/2013'</p> <p>You can specify more than one holiday period, separated by commas.</p> <p><b>Example:</b> '25/12/2013', '26/12/2013', '31/12/2013', '01/01/2014'</p>

#### 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.

## 5 Functions in scripts and chart expressions

Scripting examples

Example	Result
<code>lastworkdate ('19/12/2014', 9)</code>	Returns '31/12/2014'
<code>lastworkdate ('19/12/2014', 9, '2014-12-25', '2014-12-26')</code>	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. Then add the fields listed in the results column to a sheet in your app to see the result.

```
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 from the system clock for a specified time zone.

### Syntax:

```
LocalTime([timezone [, ignoreDST ]])
```

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
timezone	The <b>timezone</b> is specified as a string containing any of the geographical places listed under <b>Time Zone</b> in the <b>Windows Control Panel</b> for <b>Date and Time</b> or as a string in the form 'GMT+hh:mm'.  If no time zone is specified the local time will be returned.
ignoreDST	If <b>ignoreDST</b> 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]])
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>date</b>	The date to evaluate.
<b>period_ no</b>	<b>period_no</b> is an integer or expression resolving to an integer, where the value 0 indicates the lunar week which contains <b>date</b> . Negative values in <b>period_no</b> indicate preceding lunar weeks and positive values indicate succeeding lunar weeks.
<b>first_ week_ day</b>	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
<code>lunarweekend('12/01/2013')</code>	Returns 14/01/2013 23:59:59.
<code>lunarweekend('12/01/2013', -1)</code>	Returns 7/01/2013 23:59:59.
<code>lunarweekend('12/01/2013', 0, 1)</code>	Returns 15/01/2013 23:59:59.

**Example:**

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

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
```



## 5 Functions in scripts and chart expressions

```
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]])
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>date</b>	The date to evaluate.
<b>period_ no</b>	<b>period_no</b> is an integer or expression resolving to an integer, where the value 0 indicates the lunar week which contains <b>date</b> . Negative values in <b>period_no</b> indicate preceding lunar weeks and positive values indicate succeeding lunar weeks.
<b>first_ week_ day</b>	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
<code>lunarweekname('12/01/2013')</code>	Returns 2006/02.
<code>lunarweekname('12/01/2013', -1)</code>	Returns 2006/01.
<code>lunarweekname('12/01/2013', 0, 1)</code>	Returns 2006/02.

**Example:**

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

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
];
```

## 5 Functions in scripts and chart expressions

---

```
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
<b>date</b>	The date to evaluate.

## 5 Functions in scripts and chart expressions

Argument	Description
<b>period_no</b>	<b>period_no</b> is an integer or expression resolving to an integer, where the value 0 indicates the lunar week which contains <b>date</b> . Negative values in <b>period_no</b> indicate preceding lunar weeks and positive values indicate succeeding lunar weeks.
<b>first_week_day</b>	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
<code>lunarweekstart ('12/01/2013')</code>	Returns 08/01/2013.
<code>lunarweekstart ('12/01/2013', -1)</code>	Returns 01/01/2013.
<code>lunarweekstart ('12/01/2013', 0, 1 )</code>	Returns 09/01/2013. Because the offset specified by setting <b>first_week_day</b> 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. Then add the fields listed in the results column to a sheet in your app to see the result.

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:

## 5 Functions in scripts and chart expressions

```
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.

## 5 Functions in scripts and chart expressions

Example: Chart expression

Chart expressions examples

Example	Result
<code>makedate(2012)</code>	returns 2012-01-01
<code>makedate(12)</code>	returns 0012-01-01
<code>makedate(2012,12)</code>	returns 2012-12-01
<code>makedate(2012,2,14)</code>	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, 0.3333333333333333, 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
<code>maketime( 22 )</code>	returns 22:00:00

## 5 Functions in scripts and chart expressions

Example	Result
<code>maketime( 22, 17 )</code>	returns 22:17:00
<code>maketime( 22, 17, 52 )</code>	returns 22:17:52

### makeweekdate

This function returns a date calculated from the year **YYYY**, the week **WW** and the day-of-week **D**.

#### Syntax:

```
MakeWeekDate (YYYY [ , WW [ , D ] ])
```

**Return data type:** dual

#### Arguments:

##### Arguments

Argument	Description
YYYY	The year as an integer.
WW	The week as an integer.
D	The day-of-week as an integer.  If no day-of-week is stated, 0 (Monday) is assumed.

Examples and results:

##### Scripting examples

Example	Result
<code>makeweekdate(2014,6,6)</code>	returns 2014-02-09
<code>makeweekdate(2014,6,1)</code>	returns 2014-02-04
<code>makeweekdate(2014,6)</code>	returns 2014-02-03 (weekday 0 is assumed)

### 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
<code>minute ( '09:14:36' )</code>	returns 14
<code>minute ( '0.5555' )</code>	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.

**Syntax:**

```
month (expression)
```

**Return data type:** dual

Examples and results:

Scripting examples

Example	Result
<code>month( '2012-10-12' )</code>	returns Oct
<code>month( '35648' )</code>	returns Aug, because 35648 = 1997-08-06

### 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])
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>date</b>	The date to evaluate.
<b>period_no</b>	<b>period_no</b> is an integer, which, if 0 or omitted, indicates the month that contains <b>date</b> . Negative values in <b>period_no</b> 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
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. Then add the fields listed in the results column to a sheet in your app to see the result.

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 *,
```

## 5 Functions in scripts and chart expressions

```
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

#### Arguments:

Arguments

Argument	Description
<b>date</b>	The date to evaluate.
<b>period_no</b>	<b>period_no</b> is an integer, which, if 0 or omitted, indicates the month that contains <b>date</b> . Negative values in <b>period_no</b> indicate preceding months and positive values indicate succeeding months.

## 5 Functions in scripts and chart expressions

---

### Example: Chart expression

This example uses the date format **DD/MM/YYYY**, specified in the **SET DateFormat** statement at the top of your data load script. Change the format in the examples to suit your requirements. The **SET Monthnames** statement is set to Jan;Feb;Mar, and so on.

Chart expression examples

Example	Result
monthname('19/10/2013')	Returns Oct 2013
monthname('19/10/2013', -1)	Returns Sep 2013

### Example: Load script

In this example, for each invoice date in the table, the month name is created from the month name shifted four months from base\_date, and from the year.

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

```
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 *,
MonthName(InvDate, 4) AS MthName
Resident TempTable;
Drop table TempTable;
```

### Results

*The resulting table  
contains the original dates  
and a column with the  
return value of the  
monthname() function..*

InvDate	MthName
28/03/2012	Jul 2012
10/12/2012	Apr 2013
5/2/2013	Jun 2013
31/3/2013	Jul 2013
19/5/2013	Sep 2013
15/9/2013	Jan 2014
11/12/2013	Apr 2014
2/3/2014	Jul 2014
14/5/2014	Sep 2014
13/6/2014	Oct 2014
7/7/2014	Nov 2014
4/8/2014	Dec 2014

### Example: Load script

In this example, for each transaction\_date in the table, a Returnable\_Until value is created. The Returnable\_Until value is calculated by shifting the month of the transaction\_date to one month later.

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='YYYYMMDD';
SET TimestampFormat='YYYYMMDD h:mm:ss[.fff] TT';
SET FirstMonthOfYear=1;
SET MonthNames=' Jan;Feb;Mar;Apr;May;Jun;Jul;Aug;Sep;Oct;Nov;Dec';
SET
LongMonthNames=' January;February;March;April;May;June;July;August;September;October;November;December';
```

Transactions:

Load

\*,

MonthName(Date#(transaction\_date,'YYYYMMDD'), 1) as Returnable\_Until,

;

```
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, 0.3333333333333333, 2038593, M, Blue  
3757, 20180923, 3177.4, 21, .14, 203521, XL, Black  
];
```

### Results

*Qlik Sense table showing results of the monthname function  
being used in the load script.*

transaction_id	transaction_date	Returnable_Until
3750	20180830	Sep 2018
3751	20180907	Oct 2018
3752	20180916	Oct 2018
3753	20180922	Oct 2018
3754	20180922	Oct 2018
3756	20180922	Oct 2018
3757	20180923	Oct 2018

### monthsend

This function returns a value corresponding to a timestamp of the last millisecond of the month, bi-month, quarter, tertial, 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]])
```

## 5 Functions in scripts and chart expressions

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>n_ months</b>	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 (tertia), or 6 (half year).
<b>date</b>	The date to evaluate.
<b>period_ no</b>	The period can be offset by <b>period_no</b> , an integer, or expression resolving to an integer, where the value 0 indicates the period that contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding periods and positive values indicate succeeding periods.
<b>first_ month_ of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

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. Then add the fields listed in the results column to a sheet in your app to see the result.

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
```

## 5 Functions in scripts and chart expressions

---

```
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, tertial, or half-year containing a base date.

#### Syntax:

```
MonthsName (n_months, date[, period_no[, first_month_of_year]])
```



## 5 Functions in scripts and chart expressions

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>n_months</b>	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 <code>inmonth()</code> function), 2 (bi-month), 3 (equivalent to the <code>inquarter()</code> function), 4 (terial), or 6 (half year).
<b>date</b>	The date to evaluate.
<b>period_no</b>	The period can be offset by <b>period_no</b> , an integer, or expression resolving to an integer, where the value 0 indicates the period that contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding periods and positive values indicate succeeding periods.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

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
<code>monthsname(4, '19/10/2013')</code>	Returns 'Sep-Dec 2013'. Because in this and the other examples, the <b>SET Monthnames</b> statement is set to Jan;Feb;Mar, and so on.
<code>monthsname(4, '19/10/2013', -1)</code>	Returns 'May-Aug 2013'.
<code>monthsname(4, '19/10/2013', 0, 2)</code>	Returns Oct-Jan 2014. Because the year is specified to begin in month 2, therefore the four-month period ends on the first month of the following year.

**Example:**

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

In this example, for each invoice date in the table, the months name is created from the range of months in the bi-month period, and from the year. The range is offset by 4x2 months by specifying `period_no` as 4.

```
TempTable:
LOAD RecNo() as InVID, * Inline [
```

## 5 Functions in scripts and chart expressions

---

```
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 *,
MonthsName(2, InvDate, 4) AS MthsName
Resident TempTable;
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the monthsname() function.

Results table

InvDate	MthsName
28/03/2012	Nov-Dec 2012
10/12/2012	Jul-Aug 2013
5/2/2013	Sep-Oct 2013
31/3/2013	Nov-Dec2013
19/5/2013	Jan-Feb 2014
15/9/2013	May-Jun 2014
11/12/2013	Jul-Aug 2014
2/3/2014	Nov-Dec 2014
14/5/2014	Jan-Feb 2015
13/6/2014	Jan-Feb 2015
7/7/2014	Mar-Apr 2015
4/8/2014	Mar-Apr 2015

### monthsstart

This function returns a value corresponding to the timestamp of the first millisecond of the month, bi-month, quarter, tertial, or half-year containing a base date. It is also possible to find the timestamp for a previous or following time period.

## 5 Functions in scripts and chart expressions

### Syntax:

```
MonthsStart(n_months, date[, period_no [, first_month_of_year]])
```

**Return data type:** dual

### Arguments:

#### Arguments

Argument	Description
<b>n_months</b>	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 <code>inmonth()</code> function), 2 (bi-month), 3 (equivalent to the <code>inquarter()</code> function), 4 (tertil), or 6 (half year).
<b>date</b>	The date to evaluate.
<b>period_no</b>	The period can be offset by <b>period_no</b> , an integer, or expression resolving to an integer, where the value 0 indicates the period that contains <b>base_date</b> . Negative values in <b>period_no</b> indicate preceding periods and positive values indicate succeeding periods.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

### 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
<code>monthsstart(4, '19/10/2013')</code>	Returns 1/09/2013.
<code>monthsstart(4, '19/10/2013', -1)</code>	Returns 01/05/2013.
<code>monthsstart(4, '19/10/2013', 0, 2 )</code>	Returns 01/10/2013. Because the start of the year becomes month 2.

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

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
```

## 5 Functions in scripts and chart expressions

---

```
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])
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>date</b>	The date to evaluate.
<b>period_no</b>	<b>period_no</b> is an integer, which, if 0 or omitted, indicates the month that contains <b>date</b> . Negative values in <b>period_no</b> 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. Then add the fields listed in the results column to a sheet in your app to see the result.

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 *,
```

## 5 Functions in scripts and chart expressions

```
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
<b>start_date</b>	The start date to evaluate.
<b>end_date</b>	The end date to evaluate.

## 5 Functions in scripts and chart expressions

Argument	Description
<b>holiday</b>	Holiday periods to exclude from working days. A holiday period is stated as a start date and an end date, separated by commas.  <b>Example:</b> '25/12/2013', '26/12/2013'  You can specify more than one holiday period, separated by commas.  <b>Example:</b> '25/12/2013', '26/12/2013', '31/12/2013', '01/01/2014'

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. Then add the fields listed in the results column to a sheet in your app to see the result.

```
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;
```

---

## 5 Functions in scripts and chart expressions

---

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])
```




## 5 Functions in scripts and chart expressions

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
timer_ mode	<p>Can have the following values:</p> <ul style="list-style-type: none"><li>0 (time at last finished data load)</li><li>1 (time at function call)</li><li>2 (time when the app was opened)</li></ul> <div> <i>If you use the function in a data load script, <b>timer_mode=0</b> will result in the time of the last finished data load, while <b>timer_mode=1</b> will give the time of the function call in the current data load.</i></div>

**Examples and results:**

Scripting examples

Example	Result
now( 0)	Returns the time when the last data load completed.
now( 1)	<p>When used in a visualization expression, this returns the time of the function call.</p> <p>When used in a data load script, this returns the time of the function call in the current data load.</p>
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]])
```

## 5 Functions in scripts and chart expressions

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>date</b>	The date to evaluate.
<b>period_no</b>	<b>period_no</b> is an integer, where the value 0 indicates the quarter which contains <b>date</b> . Negative values in <b>period_no</b> indicate preceding quarters and positive values indicate succeeding quarters.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

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. Then add the fields listed in the results column to a sheet in your app to see the result.

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
```

## 5 Functions in scripts and chart expressions

---

```
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]])
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>date</b>	The date to evaluate.
<b>period_ no</b>	<b>period_no</b> is an integer, where the value 0 indicates the quarter which contains <b>date</b> . Negative values in <b>period_no</b> indicate preceding quarters and positive values indicate succeeding quarters.
<b>first_ month_ of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

**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. Then add the fields listed in the results column to a sheet in your app to see the result.

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
];
```

## 5 Functions in scripts and chart expressions

---

```
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
<b>date</b>	The date to evaluate.

## 5 Functions in scripts and chart expressions

Argument	Description
<b>period_no</b>	<b>period_no</b> is an integer, where the value 0 indicates the quarter which contains <b>date</b> . Negative values in <b>period_no</b> indicate preceding quarters and positive values indicate succeeding quarters.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

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
<code>quarterstart('29/10/2005')</code>	Returns 01/10/2005.
<code>quarterstart('29/10/2005', -1 )</code>	Returns 01/07/2005.
<code>quarterstart('29/10/2005', 0, 3)</code>	Returns 01/09/2005.

### Example:

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

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;
```

## 5 Functions in scripts and chart expressions

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
<code>second( '09:14:36' )</code>	returns 36
<code>second( '0.5555' )</code>	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.

## 5 Functions in scripts and chart expressions

---

### Syntax:

```
setdateyear (timestamp, year)
```

**Return data type:** dual

### Arguments:

#### Arguments

Argument	Description
<b>timestamp</b>	A standard Qlik Sense timestamp (often just a date).
<b>year</b>	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
<code>setdateyear ( '29/10/2005', 2013)</code>	Returns '29/10/2013'
<code>setdateyear ( '29/10/2005 04:26:14', 2013)</code>	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. Then add the fields listed in the results column to a sheet in your app to see the result.

```
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  
];
```



## 5 Functions in scripts and chart expressions

---

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
<b>timestamp</b>	A standard Qlik Sense timestamp (often just a date).
<b>year</b>	A four-digit year.
<b>month</b>	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**

## 5 Functions in scripts and chart expressions

statement at the top of your data load script. Change the format in the examples to suit your requirements.

Scripting examples

Example	Result
<code>setdateyearmonth ( '29/10/2005', 2013, 3)</code>	Returns '29/03/2013'
<code>setdateyearmonth ( '29/10/2005 04:26:14', 2013, 3)</code>	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. Then add the fields listed in the results column to a sheet in your app to see the result.

```
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

## 5 Functions in scripts and chart expressions

testdates	NewYearMonth
13/6/2014	13/3/2013
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	<p>Can have the following values:</p> <ul style="list-style-type: none"><li>0 (day of last finished data load)</li><li>1 (day of function call)</li><li>2 (day when the app was opened)</li></ul> <div> <i>If you use the function in a data load script, <b>timer_mode=0</b> will result in the day of the last finished data load, while <b>timer_mode=1</b> will give the day of the current data load.</i></div>

## 5 Functions in scripts and chart expressions

---

### 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'.

## 5 Functions in scripts and chart expressions

---

Argument	Description
<b>first_ week_day</b>	<p>If you don't specify <b>first_week_day</b>, the value of variable <b>FirstWeekDay</b> will be used as the first day of the week.</p> <p>If you want to use another day as the first day of the week, set <b>first_week_day</b> to:</p> <ul style="list-style-type: none"><li>• 0 for Monday</li><li>• 1 for Tuesday</li><li>• 2 for Wednesday</li><li>• 3 for Thursday</li><li>• 4 for Friday</li><li>• 5 for Saturday</li><li>• 6 for Sunday</li></ul> <p>The integer returned by the function will now use the first day of the week that you set with <b>first_week_day</b>.</p>
<b>broken_ weeks</b>	<p>If you don't specify <b>broken_weeks</b>, the value of variable <b>BrokenWeeks</b> will be used to define if weeks are broken or not.</p> <p>By default Qlik Sense functions use unbroken weeks. This means that:</p> <ul style="list-style-type: none"><li>• In some years, week 1 starts in December, and in other years, week 52 or 53 continues into January.</li><li>• Week 1 always has at least 4 days in January.</li></ul> <p>The alternative is to use broken weeks.</p> <ul style="list-style-type: none"><li>• Week 52 or 53 do not continue into January.</li><li>• Week 1 starts on January 1 and is, in most cases, not a full week.</li></ul> <p>The following values can be used:</p> <ul style="list-style-type: none"><li>• 0 (=use unbroken weeks)</li><li>• 1 (= use broken weeks)</li></ul>

## 5 Functions in scripts and chart expressions

Argument	Description
<b>reference_day</b>	<p>If you don't specify <b>reference_day</b>, the value of variable <b>ReferenceDay</b> 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.</p> <p>The following values can be used to set a different reference day:</p> <ul style="list-style-type: none"><li>• 1 (= January 1)</li><li>• 2 (= January 2)</li><li>• 3 (= January 3)</li><li>• 4 (= January 4)</li><li>• 5 (= January 5)</li><li>• 6 (= January 6)</li><li>• 7 (= January 7)</li></ul>

Examples and results:

Scripting examples

Example	Result
<code>week( '2012-10-12' )</code>	returns 41.
<code>week( '35648' )</code>	returns 32, because 35648 = 1997-08-06
<code>week('2012-10-12', 0, 1)</code>	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
<b>date</b>	The date to evaluate.

## 5 Functions in scripts and chart expressions

Argument	Description
<b>first_week_day</b>	<p>If you do not specify <b>first_week_day</b>, the value of variable <b>FirstWeekDay</b> will be used as the first day of the week.</p> <p>If you want to use another day as the first day of the week, set <b>first_week_day</b> to:</p> <ul style="list-style-type: none"><li>• 0 for Monday</li><li>• 1 for Tuesday</li><li>• 2 for Wednesday</li><li>• 3 for Thursday</li><li>• 4 for Friday</li><li>• 5 for Saturday</li><li>• 6 for Sunday</li></ul> <p>The integer returned by the function will now use the first day of the week that you set with <b>first_week_day</b> as base (0).</p> <p><i>FirstWeekDay (page 144)</i></p>

Example: Chart expression

Unless stated otherwise **FirstWeekDay** is set to 0 in these examples.

### Scripting examples

Example	Result
<code>weekday( '1971-10-12' )</code>	returns 'Tue' and 1
<code>weekday( '1971-10-12' , 6)</code>	returns 'Tue' and 2.  In this example we use Sunday (6) as the first day of the week.
<code>SET FirstWeekDay = 6;</code>  <code>... weekday( '1971-10-12')</code>	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.

## 5 Functions in scripts and chart expressions

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, 0.3333333333333333, 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
<b>date</b>	The date to evaluate.
<b>period_no</b>	<b>shift</b> is an integer, where the value 0 indicates the week which contains <b>date</b> . Negative values in shift indicate preceding weeks and positive values indicate succeeding weeks.
<b>first_week_day</b>	<p>Specifies the day on which the week starts. If omitted, the value of variable <b>FirstWeekDay</b> is used.</p> <p>The possible values <b>first_week_day</b> are:</p> <ul style="list-style-type: none"><li>• 0 for Monday</li><li>• 1 for Tuesday</li><li>• 2 for Wednesday</li><li>• 3 for Thursday</li><li>• 4 for Friday</li><li>• 5 for Saturday</li><li>• 6 for Sunday</li></ul> <p><i>FirstWeekDay (page 144)</i></p>

#### 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
<code>weekend('10/01/2013')</code>	Returns 12/01/2013 23:59:59.
<code>weekend('10/01/2013', -1)</code>	Returns 06/01/2013 23:59:59.
<code>weekend('10/01/2013', 0, 1)</code>	Returns 14/01/2013 23:59:59.

## 5 Functions in scripts and chart expressions

---

### Example:

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

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]])
```

**Return data type:** dual

#### Arguments:

##### Arguments

Argument	Description
<b>date</b>	The date to evaluate.
<b>period_no</b>	<b>shift</b> is an integer, where the value 0 indicates the week which contains <b>date</b> . Negative values in shift indicate preceding weeks and positive values indicate succeeding weeks.
<b>first_week_day</b>	<p>Specifies the day on which the week starts. If omitted, the value of variable <b>FirstWeekDay</b> is used.</p> <p>The possible values <b>first_week_day</b> are:</p> <ul style="list-style-type: none"><li>• 0 for Monday</li><li>• 1 for Tuesday</li><li>• 2 for Wednesday</li><li>• 3 for Thursday</li><li>• 4 for Friday</li><li>• 5 for Saturday</li><li>• 6 for Sunday</li></ul> <p><i>FirstWeekDay (page 144)</i></p>

## 5 Functions in scripts and chart expressions

Examples and results:

Example	Result
<code>weekname('12/01/2013')</code>	Returns 2013/02.
<code>weekname('12/01/2013', -1)</code>	Returns 2013/01.
<code>weekname('12/01/2013', 0, 1)</code>	Returns '2013/02.

### Example:

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

In this example, for each invoice date in the table, the week name is created from the year in which the week lies and its associated 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 *,  
  weekName(InvDate, 1) AS wkName  
Resident TempTable;  
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the `weekname()` function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	WkName
28/03/2012	2012/14
10/12/2012	2012/51
5/2/2013	2013/07
31/3/2013	2013/14

InvDate	WkName
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	2014/29
4/8/2014	2014/33

### 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
<b>date</b>	The date to evaluate.
<b>period_no</b>	<b>shift</b> is an integer, where the value 0 indicates the week which contains <b>date</b> . Negative values in shift indicate preceding weeks and positive values indicate succeeding weeks.

## 5 Functions in scripts and chart expressions

Argument	Description
<b>first_ week_ day</b>	<p>Specifies the day on which the week starts. If omitted, the value of variable <b>FirstWeekDay</b> is used.</p> <p>The possible values <b>first_week_day</b> are:</p> <ul style="list-style-type: none"><li>• 0 for Monday</li><li>• 1 for Tuesday</li><li>• 2 for Wednesday</li><li>• 3 for Thursday</li><li>• 4 for Friday</li><li>• 5 for Saturday</li><li>• 6 for Sunday</li></ul> <p><i>FirstWeekDay (page 144)</i></p>

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
<code>weekstart('12/01/2013')</code>	Returns 07/01/2013.
<code>weekstart('12/01/2013', -1 )</code>	Returns 31/11/2012.
<code>weekstart('12/01/2013', 0, 1)</code>	Returns 08/01/2013.

### Example:

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

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
```

## 5 Functions in scripts and chart expressions

---

```
];
```

```
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)
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** integer

Examples and results:

Scripting examples

Example	Result
<code>weekyear( '1996-12-30' )</code>	returns 1997, because week 1 of 1997 starts on 1996-12-30
<code>weekyear( '1997-01-02' )</code>	returns 1997
<code>weekyear( '1997-12-28' )</code>	returns 1997
<code>weekyear( '1997-12-30' )</code>	returns 1998, because week 1 of 1998 starts on 1997-12-29
<code>weekyear( '1999-01-02' )</code>	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
<code>year( '2012-10-12' )</code>	returns 2012
<code>year( '35648' )</code>	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]])
```



## 5 Functions in scripts and chart expressions

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>date</b>	The date to evaluate.
<b>period_no</b>	<b>period_no</b> is an integer, where the value 0 indicates the year which contains <b>date</b> . Negative values in <b>period_no</b> indicate preceding years and positive values indicate succeeding years.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

**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
yearend ( '19/10/2001' )	Returns 31/12/2001 23:59:59.
yearend ( '19/10/2001', -1 )	Returns 31/12/2000 23:59:59.
yearend ( '19/10/2001', 0, 4)	Returns 31/03/2002 23:59:59.

**Example:**

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

This example finds the final day in the year of each invoice date in the table, where the first month in 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
```

## 5 Functions in scripts and chart expressions

---

```
];
```

```
InvoiceData:  
LOAD *,  
YearEnd(InvDate, 0, 4) AS YrEnd  
Resident TempTable;  
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the `yearend()` function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	YrEnd
28/03/2012	31/03/2011
10/12/2012	31/03/2012
5/2/2013	31/03/2013
31/3/2013	31/03/2013
19/5/2013	31/03/2014
15/9/2013	31/03/2014
11/12/2013	31/03/2014
2/3/2014	31/03/2014
14/5/2014	31/03/2015
13/6/2014	31/03/2015
7/7/2014	31/03/2015
4/8/2014	31/03/2015

### 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**.

#### Syntax:

```
YearName (date[, period_no[, first_month_of_year]] )
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** dual

**Arguments:**

Argument	Description
<b>date</b>	The date to evaluate.
<b>period_no</b>	<b>period_no</b> is an integer, where the value 0 indicates the year which contains <b>date</b> . Negative values in <b>period_no</b> indicate preceding years and positive values indicate succeeding years.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> . The display value will then be a string showing two years.

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.

Script examples

Example	Result
yearname ( '19/10/2001' )	Returns 2001.
yearname ( '19/10/2001', -1 )	Returns '2000.
yearname ( '19/10/2001', 0, 4 )	Returns '2001-2002.

**Example:**

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

This example finds the first day in the year of each invoice date in the table, where the first month in the year is specified as month 4.

This example creates a four-plus-four digit name for the years in which each invoice date in the table is found. This is because the first month in 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
```

## 5 Functions in scripts and chart expressions

---

```
13/6/2014  
7/7/2014  
4/8/2014  
];
```

```
InvoiceData:  
LOAD *,  
YearName(InvDate, 0, 4) AS YrName  
Resident TempTable;  
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the `yearname()` function.

Results table

InvDate	YrName
28/03/2012	2011-2012
10/12/2012	2012-2013
5/2/2013	2012-2013
31/3/2013	2012-2013
19/5/2013	2013-2014
15/9/2013	2013-2014
11/12/2013	2013-2014
2/3/2014	2013-2014
14/5/2014	2014-2015
13/6/2014	2014-2015
7/7/2014	2014-2015
4/8/2014	2014-2015

### 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]])
```

## 5 Functions in scripts and chart expressions

---

**Return data type:** dual

**Arguments:**

Arguments

Argument	Description
<b>date</b>	The date to evaluate.
<b>period_no</b>	<b>period_no</b> is an integer, where the value 0 indicates the year which contains <b>date</b> . Negative values in <b>period_no</b> indicate preceding years and positive values indicate succeeding years.
<b>first_month_of_year</b>	If you want to work with (fiscal) years not starting in January, indicate a value between 2 and 12 in <b>first_month_of_year</b> .

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
yearstart ('19/10/2001')	Returns 01/01/2001.
yearstart ('19/10/2001', -1)	Returns 01/01/2000.
yearstart ('19/10/2001', 0, 4)	Returns 01/04/2001.

**Example:**

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

This example finds the first day in the year of each invoice date in the table, where the first month in 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
```

## 5 Functions in scripts and chart expressions

```
4/8/2014  
];
```

```
InvoiceData:  
LOAD *,  
YearStart(InvDate, 0, 4) AS YrStart  
Resident TempTable;  
Drop table TempTable;
```

The resulting table contains the original dates and a column with the return value of the yearstart() function. You can display the full timestamp by specifying the formatting in the properties panel.

Results table

InvDate	YrStart
28/03/2012	01/04/2011
10/12/2012	01/04/2012
5/2/2013	01/04/2012
31/3/2013	01/04/2012
19/5/2013	01/04/2013
15/9/2013	01/04/2013
11/12/2013	01/04/2013
2/3/2014	01/04/2013
14/5/2014	01/04/2014
13/6/2014	01/04/2014
7/7/2014	01/04/2014
4/8/2014	01/04/2014

### 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 <b>yearoffset</b> , <b>yeartodate</b> returns True for the same period in another year. A negative <b>yearoffset</b> 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 <b>firstmonth</b> 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 <b>firstmonth</b> = 5.
todaydate	By specifying a <b>todaydate</b> (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 525) (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.

## 5 Functions in scripts and chart expressions

```
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 <b>John</b> is selected in <b>First name</b> .  <code>GetAlternativeCount ([First name])</code>	4 as there are 4 unique and excluded (gray) values in <b>First name</b> .

## 5 Functions in scripts and chart expressions

Examples	Results
Given that <b>John</b> and <b>Peter</b> are selected. <code>GetAlternativeCount ([First name])</code>	3 as there are 3 unique and excluded (gray) values in <b>First name</b> .
Given that no values are selected in <b>First name</b> . <code>GetAlternativeCount ([First name])</code>	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
<code>record_sep</code>	Separator to be put between field records. The default is <CR><LF> meaning a new line.
<code>tag_sep</code>	Separator to be put between the field name tag and the field values. The default is ': '.
<code>value_sep</code>	The separator to be put between field values. The default is ', '.

## 5 Functions in scripts and chart expressions

Arguments	Description
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 <b>state_name</b> 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 <b>John</b> is selected in <b>First name</b> . <code>GetCurrentSelections ()</code>	'First name: John'
Given that <b>John</b> and <b>Peter</b> are selected in <b>First name</b> . <code>GetCurrentSelections ()</code>	'First name: John, Peter'
Given that <b>John</b> and <b>Peter</b> are selected in <b>First name</b> and <b>JA</b> is selected in <b>Initials</b> . <code>GetCurrentSelections ()</code>	'First name: John, Peter Initials: JA'
Given that <b>John</b> is selected in <b>First name</b> and <b>JA</b> is selected in <b>Initials</b> . <code>GetCurrentSelections ( chr(13)&amp;chr(10) , ' = ' )</code>	'First name = John Initials = JA'
Given that you have selected all names except Sue in <b>First name</b> and no selections in <b>Initials</b> . <code>GetCurrentSelections (chr(13)&amp;chr(10), '=', ' ', 3)</code>	'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 <b>First name</b> .	GetExcludedCount (Initials) = 0 There are no selections.
If <b>John</b> is selected in <b>First name</b> .	GetExcludedCount (Initials) = 5 There are 5 excluded values in <b>Initials</b> with dark gray color. The sixth cell (JA) will be white as it is associated with the selection John in <b>First name</b> .
If <b>John</b> and <b>Peter</b> are selected.	GetExcludedCount (Initials) = 3 John is associated with 1 value and Peter is associated with 2 values, in <b>Initials</b> .
If <b>John</b> and <b>Peter</b> are selected in <b>First name</b> , and then <b>Franc</b> is selected in <b>Last name</b> .	GetExcludedCount ([First name]) = 4 There are 4 excluded values in <b>First name</b> with dark gray color. <b>GetExcludedCount()</b> evaluates for fields with excluded values, including alternative and selected excluded fields.
If <b>John</b> and <b>Peter</b> are selected in <b>First name</b> , and then <b>Franc</b> and <b>Anderson</b> are selected in <b>Last name</b> .	GetExcludedCount (Initials) = 4 There are 4 excluded values in <b>Initials</b> with dark gray color. The other two cells (JA and PF) will be white as they associated with the selections John and Peter in <b>First name</b> .

## 5 Functions in scripts and chart expressions

Examples	Results
If <b>John</b> and <b>Peter</b> are selected in <b>First name</b> , and then <b>Franc</b> and <b>Anderson</b> are selected in <b>Last name</b> .	<code>GetExcludedCount ([Last name]) = 4</code> There are 4 excluded values in <b>Initials</b> . 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

#### Arguments:

##### Arguments

Arguments	Description
<code>field_name</code>	The field containing the range of data to be measured.
<code>value_sep</code>	The separator to be put between field values. The default is ','.
<code>max_values</code>	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.
<code>state_name</code>	The name of an alternate state that has been chosen for the specific visualization. If the <b>state_name</b> 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 <b>John</b> is selected in <b>First name</b> .  <code>GetFieldSelections ([First name])</code>	'John'
Given that <b>John</b> and <b>Peter</b> are selected.  <code>GetFieldSelections ([First name])</code>	'John,Peter'
Given that <b>John</b> and <b>Peter</b> are selected.  <code>GetFieldSelections ([First name],'; ')</code>	'John; Peter'
Given that <b>John</b> , <b>Sue</b> , <b>Mark</b> are selected in <b>First name</b> .  <code>GetFieldSelections ([First name],';',2)</code>	'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 <b>includeexcluded</b> 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*

transacti on_date	custom er_id	transacti on_ quantity	=GetObjectDime nsion ()	=GetObjectDime nsion (0)	=GetObjectDime nsion (1)
2018/08/3 0	049681	13	transaction_date	transaction_date	customer_id
2018/08/3	203521	6	transaction_date	transaction_date	customer_id



## 5 Functions in scripts and chart expressions

transacti on_date	custom er_id	transacti on_ quantity	=GetObjectDime nsion ()	=GetObjectDime nsion (0)	=GetObjectDime nsion (1)
0					
2018/08/30	203521	21	transaction_date	transaction_date	customer_id
0					

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.

transactio n_date	custome r_id	transactio n_quantity	=GetObjectFiel d ()	=GetObjectFiel d (0)	=GetObjectFiel d (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*

custom er_id	sum (transacti on_ quantity)	Avg (transacti on_ quantity)	=GetObjectMea sure ()	=GetObjectMea sure(0)	=GetObjectMeasu re(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)
```

## 5 Functions in scripts and chart expressions

### 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 <b>John</b> is selected in <b>First name</b> .  GetPossibleCount ([Initials])	1 as there is 1 value in Initials associated with the selection, <b>John</b> , in <b>First name</b> .
Given that <b>John</b> is selected in <b>First name</b> .  GetPossibleCount ([First name])	1 as there is 1 selection, <b>John</b> , in <b>First name</b> .
Given that <b>Peter</b> is selected in <b>First name</b> .  GetPossibleCount ([Initials])	2 as Peter is associated with 2 values in <b>Initials</b> .
Given that no values are selected in <b>First name</b> .  GetPossibleCount ([First name])	5 as there are no selections and there are 5 unique values in <b>First name</b> .
Given that no values are selected in <b>First name</b> .  GetPossibleCount ([Initials])	6 as there are no selections and there are 6 unique values in <b>Initials</b> .

### 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 <b>True()</b> , 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 <b>state_name</b> 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 <b>John</b> is selected in <b>First name</b> .  <code>GetSelectedCount ([First name])</code>	1 as one value is selected in <b>First name</b> .
Given that <b>John</b> is selected in <b>First name</b> .  <code>GetSelectedCount ([Initials])</code>	0 as no values are selected in <b>Initials</b> .
With no selections in <b>First name</b> , select all values in <b>Initials</b> and after that select the value <b>Yes</b> in <b>Has cellphone</b> .  <code>GetSelectedCount ([Initials], True())</code>	6. Although selections with <b>Initials</b> MC and PD have <b>Has cellphone</b> set to <b>No</b> , the result is still 6, because the argument <code>include_excluded</code> is set to <code>True()</code> .

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 ()
```

##### FileName

The **FileName** function returns a string containing the name of the table file currently being read, without path or extension.

```
FileName ()
```

##### 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

---

## 5 Functions in scripts and chart expressions

---

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 for the date and time of the last modification of the file filename. If no filename is specified, the function will refer to 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.

```
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 ID3v3 for MP3 files or EXIF for JPG files, not meta information saved in the **Windows File Explorer**.*

#### Arguments:

##### Arguments

Argument	Description
filename	<p>The name of a media file including path, if needed, as a folder data connection.</p> <p><b>Example: 'lib://Table Files/'</b></p> <p>In legacy scripting mode, the following path formats are also supported:</p> <ul style="list-style-type: none"><li>absolute</li></ul> <p><b>Example: c:\data\</b></p> <ul style="list-style-type: none"><li>relative to the Qlik Sense app working directory.</li></ul> <p><b>Example: data\</b></p>
attributename	<p>The name of a meta tag.</p>

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.

*File system access restriction (page 762)*

### 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
```



---

## 5 Functions in scripts and chart expressions

```
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, // Unsynchronized 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,
```

---

## 5 Functions in scripts and chart expressions

---

```
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,
```

---

## 5 Functions in scripts and chart expressions

---

```
// 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,
```

## 5 Functions in scripts and chart expressions

---

```
// 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,
```

---

## 5 Functions in scripts and chart expressions

---

```
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.

### FileName

The **FileName** function returns a string containing the name of the table file currently being read, without path or extension.

#### Syntax:

```
FileName()
```

## 5 Functions in scripts and chart expressions

Examples and results:

Scripting examples

Example	Result
LOAD *, filename( ) 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 ( )**

## 5 Functions in scripts and chart expressions

---

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	<p>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.</p> <p><b>Example: 'lib://Table Files/'</b></p> <p>In legacy scripting mode, the following path formats are also supported:</p> <ul style="list-style-type: none"><li>absolute</li></ul> <p><b>Example: c:\data\</b></p> <ul style="list-style-type: none"><li>relative to the Qlik Sense app working directory.</li></ul> <p><b>Example: data\</b></p> <ul style="list-style-type: none"><li>URL address (HTTP or FTP), pointing to a location on the Internet or an intranet.</li></ul> <p><b>Example: http://www.qlik.com</b></p>

### 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 for the date and time of the last modification of the file filename. If no filename is specified, the function will refer to the currently read table file.

### Syntax:

```
FileTime( [ filename ] )
```

### Arguments:

Arguments

Argument	Description
filename	<p>The name of a file, if necessary including path, as a folder or web file data connection.</p> <p><b>Example: 'lib://Table Files'</b></p> <p>In legacy scripting mode, the following path formats are also supported:</p> <ul style="list-style-type: none"><li>absolute</li></ul> <p><b>Example: c:\data\</b></p> <ul style="list-style-type: none"><li>relative to the Qlik Sense app working directory.</li></ul> <p><b>Example: data\</b></p> <ul style="list-style-type: none"><li>URL address (HTTP or FTP), pointing to a location on the Internet or an intranet.</li></ul> <p><b>Example: http://www.qlik.com</b></p>

### Examples and results:

Script examples

Example	Result
LOAD *, FileTime( ) as X from abc.txt;	Will return the date and time of the last modification of the file (abc.txt) as a timestamp 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
<b>foldername</b>	<p>Name of the Microsoft Windows folder.</p> <p>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.</p> <p>Examples:</p> <p><i>MyMusic</i></p> <p><i>MyDocuments</i></p>

### 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\smul\Music* for the folder *MyMusic*
- *C:\Users\smul\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.

### Syntax:

```
QvdCreateTime (filename)
```

### Arguments:

Arguments	
Argument	Description
filename	<p>The name of a QVD file, if necessary including path, as a folder or web data connection.</p> <p><b>Example: 'lib://Table Files'</b></p> <p>In legacy scripting mode, the following path formats are also supported:</p> <ul style="list-style-type: none"><li>absolute</li></ul> <p><b>Example: c:\data\</b></p> <ul style="list-style-type: none"><li>relative to the Qlik Sense app working directory.</li></ul> <p><b>Example: data\</b></p> <ul style="list-style-type: none"><li>URL address (HTTP or FTP), pointing to a location on the Internet or an intranet.</li></ul> <p><b>Example: http://www.qlik.com</b></p>

### Example:

```
QvdCreateTime('MyFile.qvd')
QvdCreateTime('C:\MyDir\MyFile.qvd')
QvdCreateTime('lib://data\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	<p>The name of a QVD file, if necessary including path, as a folder or web data connection.</p> <p><b>Example: 'lib://Table Files'</b></p> <p>In legacy scripting mode, the following path formats are also supported:</p> <ul style="list-style-type: none"><li>absolute</li></ul> <p><b>Example: c:\data\</b></p> <ul style="list-style-type: none"><li>relative to the Qlik Sense app working directory.</li></ul> <p><b>Example: data\</b></p> <ul style="list-style-type: none"><li>URL address (HTTP or FTP), pointing to a location on the Internet or an intranet.</li></ul> <p><b>Example: http://www.qlik.com</b></p>
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://data\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	<p>The name of a QVD file, if necessary including path, as a folder or web data connection.</p> <p><b>Example: 'lib://Table Files'</b></p> <p>In legacy scripting mode, the following path formats are also supported:</p> <ul style="list-style-type: none"><li>absolute</li></ul> <p><b>Example: c:\data\</b></p> <ul style="list-style-type: none"><li>relative to the Qlik Sense app working directory.</li></ul> <p><b>Example: data\</b></p> <ul style="list-style-type: none"><li>URL address (HTTP or FTP), pointing to a location on the Internet or an intranet.</li></ul> <p><b>Example: http://www.qlik.com</b></p>

### Examples:

```
QvdNoOfFields ('MyFile.qvd')  
QvdNoOfFields ('C:\MyDir\MyFile.qvd')  
QvdNoOfFields ('lib://data\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	<p>The name of a QVD file, if necessary including path, as a folder or web data connection.</p> <p><b>Example: 'lib://Table Files'</b></p> <p>In legacy scripting mode, the following path formats are also supported:</p> <ul style="list-style-type: none"><li>absolute</li></ul> <p><b>Example: c:\data\</b></p> <ul style="list-style-type: none"><li>relative to the Qlik Sense app working directory.</li></ul> <p><b>Example: data\</b></p> <ul style="list-style-type: none"><li>URL address (HTTP or FTP), pointing to a location on the Internet or an intranet.</li></ul> <p><b>Example: http://www.qlik.com</b></p>

### Examples:

```
QvdNoOfRecords ('MyFile.qvd')  
QvdNoOfRecords ('C:\MyDir\MyFile.qvd')  
QvdNoOfRecords ('lib://data\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	<p>The name of a QVD file, if necessary including path, as a folder or web data connection.</p> <p><b>Example: 'lib://Table Files/'</b></p> <p>In legacy scripting mode, the following path formats are also supported:</p> <ul style="list-style-type: none"><li>absolute</li></ul> <p><b>Example: c:\data\</b></p> <ul style="list-style-type: none"><li>relative to the Qlik Sense app working directory.</li></ul> <p><b>Example: data\</b></p> <ul style="list-style-type: none"><li>URL address (HTTP or FTP), pointing to a location on the Internet or an intranet.</li></ul> <p><b>Example: http://www.qlik.com</b></p>

### 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)
```

## 5 Functions in scripts and chart expressions

**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.
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
<code>BlackAndSchole(130, 4, 68.5, 0.4, 0.04, 'call')</code>  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 ] ])
```

## 5 Functions in scripts and chart expressions

**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.
pv	The present value, or lump-sum amount, that a series of future payments is worth right now. If <b>pv</b> 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 <b>type</b> 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.

## 5 Functions in scripts and chart expressions

Argument	Description
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 <b>p</b> v 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 <b>f</b> v 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 <b>t</b> ype 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?  <code>nPer(0.005, -20, 0, 800)</code>	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 <b>p</b> v is omitted, it is assumed to be 0 (zero).

## 5 Functions in scripts and chart expressions

Argument	Description
fv	The future value, or cash balance, you want to attain after the last payment is made. If <b>fv</b> 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 <b>type</b> 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:  <code>Pmt(0.1/12,8,20000)</code>	Returns - \$2,594.66
For the same loan, if payment is due at the beginning of the period, the payment is:  <code>Pmt(0.1/12,8,20000,0,1)</code>	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 <b>fv</b> 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 <b>type</b> is omitted, it is assumed to be 0.

## 5 Functions in scripts and chart expressions

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%?  <code>PV(0.07/12, 12*5, -100, 0, 0)</code>	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 <b>pv</b> 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 <b>fv</b> 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 <b>type</b> 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?  <code>Rate(60, -300, 10000)</code>	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.



*For reasons of clarity, 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 an expression numerically in the number format set in the system variables in the data load script, or in the operating system, unless a format string is supplied, and optional decimal and thousands separators.

```
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 \(page 593\)](#)

## 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 <b>codepage</b> .
codepage	Number representing the code page to be applied to the field or expression given by <b>text</b> .

**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;	<p>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.</p> <p>The <b>codepage</b> value 1253 represents Windows Greek character set, the value 1255 represents Hebrew, and the value 65001 represents standard Latin UTF-8 characters.</p>

---

**See also:** *Character set (page 101)*

## 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.

---

## 5 Functions in scripts and chart expressions

---

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

## 5 Functions in scripts and chart expressions

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.*

## 5 Functions in scripts and chart expressions

Examples and results:

### Scripting examples

Example	Description
<p>Add the following examples to your script and run it.</p> <pre>Load dual (   NameDay, NumDay ) as   DayOfWeek inline [ NameDay, NumDay Monday, 0 Tuesday, 1 Wednesday, 2 Thursday, 3 Friday, 4 Saturday, 5 Sunday, 6 ];</pre>	<p>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.</p>
<pre>Load Dual('Q' &amp; Ceil(   Month(Now())/3), Ceil(   Month(Now())/3)) as Quarter AutoGenerate 1;</pre>	<p>This example finds the current quarter. It is displayed as Q1 when the <b>Now()</b> 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.</p>
<pre>Dual('Q' &amp; Ceil(Month(   Date)/3), Ceil(Month(   Date)/3)) as Quarter</pre>	<p>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.</p>
<pre>Dual(WeekYear(Date) &amp; '-W' &amp; Week(Date), WeekStart(Date)) as YearWeek</pre>	<p>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.</p>

## 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
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]])
```

## 5 Functions in scripts and chart expressions

**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 an expression numerically in the number format set in the system variables in the data load script, or in the operating system, unless a format string is supplied, and optional decimal and thousands separators.

#### 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 number 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 describing how the resulting string is to be formatted. If omitted, the number 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.

Example: Chart expression

The examples below assume the following default settings:

- Number format setting 1: # ##0
- Number format setting 2: #,##0

#### Example:

```
Num( 35648.375 , '0.0' )
```

Results table

Results	Setting 1	Setting 2
String	35 648 375	35648.375
Number	35648375	35648.375

## 5 Functions in scripts and chart expressions

### Example:

```
Num( 35648, '#,##0.##', '.', ',' )
```

Results table

Results	Setting 1	Setting 2
String	35,648.00	35,648.00
Number	35648	35648

### Example:

```
Num( pi(), '0,00' )
```

Results table

Results	Setting 1	Setting 2
String	3,14	003
Number	3.141592653	3.141592653

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
```



## 5 Functions in scripts and chart expressions

```
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, 0.3333333333333333, 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, 0.3333333333333333, 2038593, M, Blue
3757, 20180923, 3177.4, 21, .14, 203521, XL, Black
];
```

## 5 Functions in scripts and chart expressions

Qlik Sense table showing the results of the *Num* function being used in the load script to format percentages.

Discount	Discount #,##0%
0.3333333333333333	33%
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

## 5 Functions in scripts and chart expressions

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.

## 5 Functions in scripts and chart expressions

Argument	Description
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

---

## 5 Functions in scripts and chart expressions

---

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:  $\text{Combin}(p, q) = p! / q! (p-q)!$  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:  $\text{Permut}(p, q) = (p)! / (p - q)!$  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

---

## 5 Functions in scripts and chart expressions

---

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
<b>x</b>	Input number.
<b>step</b>	Interval increment. The default value is 1.
<b>offset</b>	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 ... $0 < x \leq 1$ , $1 < x \leq 2$ , <b><math>2 &lt; x \leq 3</math></b> , $3 < x \leq 4$ ...

## 5 Functions in scripts and chart expressions

Examples	Results
<code>ceil(4.2 )</code>	Returns 5
<code>ceil(3.88 ,0.1)</code>	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, <b>3.8 &lt; x &lt;= 3.9</b> , 3.9 < x <= 4.0...
<code>ceil(3.88 ,5)</code>	Returns 5
<code>ceil(1.1 ,1)</code>	Returns 2
<code>ceil(1.1 ,1,0.5)</code>	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 ... <b>0.5 &lt; x &lt;=1.5</b> , 1.5 < x <= 2.5, 2.5 < x <=3.5, 3.5 < x <=4.5...
<code>ceil(1.1 ,1,-0.01)</code>	Returns 1.99  The intervals are ...-0.01 < x <= 0.99, <b>0.99 &lt; x &lt;= 1.99</b> , 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:  $\text{combin}(p,q) = p! / q!(p-q)!$  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?  <code>combin( 35,7 )</code>	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
<code>fabs( 2.4 )</code>	Returns 2.4
<code>fabs( -3.8 )</code>	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
<code>Fact( 1 )</code>	Returns 1
<code>Fact( 5 )</code>	Returns 120 ( $1 * 2 * 3 * 4 * 5 = 120$ )
<code>Fact( -5 )</code>	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]])
```

## 5 Functions in scripts and chart expressions

**Return data type:** numeric

**Arguments:**

Arguments

Argument	Description
<b>x</b>	Input number.
<b>step</b>	Interval increment. The default value is 1.
<b>offset</b>	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, <b>2 &lt;= x &lt; 3</b> , 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, <b>3.8 &lt;= x &lt; 3.9</b> , 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 ... <b>0.5 &lt;= x &lt; 1.5</b> , 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
<b>a</b>	Dividend
<b>b</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  $\text{Frac}(x) + \text{Floor}(x) = x$ . 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 it.

For example: The fractional part of 11.43 =  $11.43 - 11 = 0.43$

For a negative number, say -1.4,  $\text{Floor}(-1.4) = -2$ , 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

## 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
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:  $\text{Permut}(p,q) = (p)! / (p - q)!$  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
<b>x</b>	Input number.
<b>step</b>	Interval increment. The default value is 1.
<b>offset</b>	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, <b>3&lt;= x &lt;4</b> ...
Round(3.8,4 )	Returns 4
Round(2.5 )	Returns 3. Rounded up because 2.5 is exactly half of the default step interval.
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 ... <b>0 &lt;= x &lt;4</b> , 4 <= x <8, 8<= x <12...

## 5 Functions in scripts and chart expressions

Examples	Results
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 ... <b>0 &lt;= x &lt;6</b> , 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, <b>3.8 &lt;= x &lt;3.9</b> , 3.9 <= x < 4.0...
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 ... <b>0.5 &lt;= x &lt;1.5</b> , 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](https://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 functions take a single 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)
```

---

## 5 Functions in scripts and chart expressions

---

### 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.

---

## 5 Functions in scripts and chart expressions

---

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 <b>AreaPolygon</b> is transformed using the inverse transformation of the Mercator projection and stored as <b>InvProjectGeometry</b> 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)
```

## 5 Functions in scripts and chart expressions

**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.*

---

## 5 Functions in scripts and chart expressions

---

Example:

Script examples

Example	Result
In a Load statement: GeoProject('mercator',Area) as GetProject	The Mercator projection is applied to the geometry loaded as <b>Area</b> , and the result is stored as <b>GetProject</b> .

### 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 <b>AreaPolygon</b> is transformed using the Mercator projection and stored as <b>ProjectGeometry</b> 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.
value	<div>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. <div> <i>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.</i></div></div>

**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:

## 5 Functions in scripts and chart expressions

---

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.



*For reasons of clarity, 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 595) (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 596) (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 597) (text[, format[, dec_sep[, thou_sep ] ] ])
```

#### Num#

**Num#()** converts a text string to a numerical value, in the number format set in the data load script or the operating system. Custom decimal and thousand separator symbols are optional parameters.

```
Num# (page 598) (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 599) (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 600) (text[, format])
```

### See also:

 [Formatting functions \(page 559\)](#)

### 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 *,
```

## 5 Functions in scripts and chart expressions

```
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: .

## 5 Functions in scripts and chart expressions

### Results

Example	Result
<code>Interval#( A, 'D hh:mm' )</code> 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
```

## 5 Functions in scripts and chart expressions

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#()** converts a text string to a numerical value, in the number format set in the data load script or the operating system. Custom decimal and thousand separator symbols are optional parameters.

#### Syntax:

```
Num# (text[, format[, dec_sep [, thou_sep ] ] ])
```

**Return data type:** dual

#### Arguments:

Arguments

Argument	Description
text	The text string to be evaluated.
format	String describing how the resulting date string is to be formatted. If omitted, the number format set in the operating system is used.
dec_sep	String specifying the decimal number separator. If omitted, the DecimalSep value set in the data load script is used.
thou_sep	String specifying the thousands number separator. If omitted, the ThousandSep value set in the data load script is used.

#### Examples and results:

The examples below assume the two following operating system settings:

## 5 Functions in scripts and chart expressions

- Number format default setting 1: # ##0
- Number format default setting 2: #,##0

Num#( A, '#.#', '.', ',')  
where A=35,648.375

Results

Results	Setting 1	Setting 2
String	35,648.375	35,648.375
Number	35648.375	35648.375

### 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])

## 5 Functions in scripts and chart expressions

**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 finds the number of distinct values in a field.

```
FieldValueCount(field_name)
```

## Pivot table functions

These functions can only be used in chart expressions.

---

## 5 Functions in scripts and 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

## 5 Functions in scripts and chart expressions

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()** finds the value of a field in a table for a row that has already been loaded or that exists in internal memory. The row number can be specified, as can the table.

```
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 634) (expr)
```

### See also:

📄 [Range functions \(page 653\)](#)

## 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	<p>Specifying an offsetn, greater than 0, moves the evaluation of the expression n rows further up from the current row.</p> <p>Specifying an offset of 0 will evaluate the expression on the current row.</p> <p>Specifying a negative offset number makes the Above function work like the Below function with the corresponding positive offset number.</p>

## 5 Functions in scripts and chart expressions

Argument	Description
count	By specifying a third argument <b>count</b> greater than 1, the function will return a range of <b>count</b> values, one for each of <b>count</b> 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. <i>Range functions (page 653)</i>
TOTAL	If the table is one-dimensional or if the qualifier <b>TOTAL</b> 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))`.

## 5 Functions in scripts and chart expressions

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*

## 5 Functions in scripts and chart expressions

Customer	Product	Month	Sum([Sales])	Above(Sum(Sales))
			<b>2566</b>	-
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))
			<b>2566</b>	-
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



## 5 Functions in scripts and chart expressions

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)).	<p>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 expression Sum(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.</p> <p>A table with Customer as dimension gives the following results for the RangeAvg() expression.</p> <table><tr><td>Astrida</td><td>-</td></tr><tr><td>Betacab</td><td>587</td></tr><tr><td>Canutility</td><td>563</td></tr><tr><td>Divadip:</td><td>603</td></tr></table>	Astrida	-	Betacab	587	Canutility	563	Divadip:	603
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 \(page 610\)](#)
-  [Bottom - chart function \(page 613\)](#)
-  [Top - chart function \(page 635\)](#)
-  [RangeAvg \(page 656\)](#)

### 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	<p>Specifying an <b>offset</b><i>n</i>, greater than 1 moves the evaluation of the expression <i>n</i> rows further down from the current row.</p> <p>Specifying an offset of 0 will evaluate the expression on the current row.</p> <p>Specifying a negative offset number makes the <b>Below</b> function work like the <b>Above</b> function with the corresponding positive offset number.</p>
count	By specifying a third parameter <b>count</b> greater than 1, the function will return a range of <b>count</b> values, one for each of <b>count</b> 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. <i>Range functions (page 653)</i>
TOTAL	If the table is one-dimensional or if the qualifier <b>TOTAL</b> 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.

## 5 Functions in scripts and chart expressions



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).

## 5 Functions in scripts and chart expressions

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 <b>Below</b> function can be used as input to the range functions. For example: <code>RangeAvg (Below(sum(Sales),1,3))</code> .	<p>In the arguments for the <b>Below()</b> function, offset is set to 1 and count is set to 3. The function finds the results of the expression <b>Sum(Sales)</b> 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 <code>RangeAvg()</code> function, which finds the average of the values in the supplied range of numbers.</p> <p>A table with <b>Customer</b> as dimension gives the following results for the <code>RangeAvg()</code> expression.</p>								
	<table><tr><td>Astrida</td><td>659.67</td></tr><tr><td>Betacab</td><td>720</td></tr><tr><td>Canutility</td><td>757</td></tr><tr><td>Divadip:</td><td>-</td></tr></table>	Astrida	659.67	Betacab	720	Canutility	757	Divadip:	-
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 \(page 605\)](#)
-  [Bottom - chart function \(page 613\)](#)
-  [Top - chart function \(page 635\)](#)
-  [RangeAvg \(page 656\)](#)

## 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.

## 5 Functions in scripts and chart expressions

Argument	Description
offset	Specifying an <b>offset</b> <i>n</i> greater than 1 moves the evaluation of the expression up <i>n</i> rows above the bottom row.  Specifying a negative offset number makes the <b>Bottom</b> function work like the <b>Top</b> function with the corresponding positive offset number.
count	By specifying a third parameter <b>count</b> greater than 1, the function will return not one but a range of <b>count</b> values, one for each of the last <b>count</b> rows of the current column segment. In this form, the function can be used as an argument to any of the special range functions. <i>Range functions (page 653)</i>
TOTAL	If the table is one-dimensional or if the qualifier <b>TOTAL</b> 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))**.

## 5 Functions in scripts and chart expressions

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
			<b>2566</b>	-
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).*

## 5 Functions in scripts and chart expressions

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 <b>Bottom</b> function can be used as input to the range functions. For example: RangeAvg (Bottom (Sum(Sales),1,3)).	In the arguments for the <b>Bottom()</b> function, offset is set to 1 and count is set to 3. The function finds the results of the expression <b>Sum(Sales)</b> 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 <b>Customer</b> 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 \(page 635\)](#)

## 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. <div> <i>The Column() function disregards dimension columns.</i></div>

### Limitations:

If **ColumnNo** references a column for which there is no measure, a NULL value is returned.

Recursive calls will return NULL.

## 5 Functions in scripts and chart expressions

### 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: <code>sum (UnitPrice*UnitSales)</code> .  Total Sales Value is added as a measure with the expression: <code>Sum(TOTAL UnitPrice*UnitSales)</code>  % Sales is added as a measure with the expression <code>100*Column(1)/Column(2)</code>	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 <i>Percentage total sales (page 618)</i> .
Make the selection Customer A.	The selection changes the Total Sales Value, and therefore the %Sales. See the example <i>Percentage of sales for selected customer (page 618)</i> .

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. The number of dimensions in all rows, except the total which will be 0, will be returned. 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:

A typical use for dimensionality is when you want to make a calculation only if there is a value present for a dimension.

#### Examples and results

Example	Result
For a table containing the dimension UnitSales, you might only want to indicate an invoice is sent:  IF(Dimensionality()=3, "Invoiced").	

### 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	<p>The name of the field where you want to search for a value. You can use an explicit field name without quotes.</p> <p>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.</p>
expr	<p>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.</p> <div> You cannot refer to fields that are not included in the current load statement.</div> <p>This argument is optional. If you omit it, the function will check if the value of <b>field_name</b> in the current record already exists.</p>

Data used in example:

```
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 '|');
```

```
Drop Tables Employees;
```

### Example:

```
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:

```
Exists(Employee, 'Bill')
```

Returns -1 (True) if the field value **'Bill'** is found in the current content of the field **Employee**.

### Example:

```
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.

## 5 Functions in scripts and chart expressions

---

Results

Employee	Address
Bill	New York
John	Miami
Steve	Chicago

### Example:

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|Miami2222222223  
+0  
(delimiter is '|') where not Exists (Employee);
```

Drop Tables Employees;

The where clause includes not: where not Exists (Employee, Name).

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 the value is included in the Employee symbol table. Hence, when the second line is checked, the value now exists.

The next example shows how to load all values.

Results

Employee	Address
Mary	London
Lucy	Madrid

### Example:

```
Employees:
Load Employee As Name, ID, Salary;
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 you need to change two things:

- Add a preceding load to Employees where you rename Employee to Name.  
Load Employee As Name, ID, Salary;
- Change the Where condition in Citizens to:  
not Exists (Name, Employee).

This will create different symbol tables for Name and Employee. When the second row for 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)
```

## 5 Functions in scripts and chart expressions

**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 <b>field_name</b> .

**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 <b>Names</b> is loaded, as in the sample data.
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 <b>First name</b> field. Note that in a filter pane <b>John</b> 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 <b>FieldIndex()</b> returns only one value, that is the first occurrence in the load order.
Script function: Given the table <b>Names</b> 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 <b>First name</b> field. Note that in a filter pane <b>John</b> would appear as number 2 from the top as it's sorted alphabetically and not as in the load order.



## 5 Functions in scripts and chart expressions

Examples	Results
Peter1: Load FieldIndex('First name','Peter') as MyPeterPos Resident Names;	MyPeterPos=4, because <b>FieldIndex()</b> 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.

## 5 Functions in scripts and chart expressions

### 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 <b>Names</b> is loaded, as in the sample data.
Chart function: In a table containing the dimension First name, add as a measure:	
Fieldvalue('First name','1')	John, because John appears first in the load order of the <b>First name</b> field. Note that in a filter pane <b>John</b> would appear as number 2 from the top, after <b>Jane</b> , as it's sorted alphabetically and not as in the load order.
Fieldvalue('First name','7')	NULL, because there are only 6 values in the <b>First name</b> field.
Script function: Given the table <b>Names</b> is loaded, as in the example data:	
John1: Load Fieldvalue('First name',1) as MyPos1 Resident Names;	MyPos1=John, because 'John' appears first in the load order of the <b>First name</b> field.
Peter1: Load Fieldvalue('First name',7) as MyPos2 Resident Names;	MyPos2= - (Null), because there are only 6 values in the <b>First name</b> field.

### 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 FieldValue('First name',1) as MyPos1
Resident Names;

Peter1:
Load FieldValue('First name',7) as MyPos2
Resident Names;
```

### FieldValueCount

**FieldValueCount()** is an **integer** function that finds the number of distinct values in a field.

#### 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 <b>Names</b> 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 <b>Peter</b> appears twice.
FieldValueCount('Initials')	6 as <b>Initials</b> only has distinct values.
Script function, Given the table <b>Names</b> is loaded, as in the example data:	
John1: Load FieldValueCount('First name') as MyFieldCount1 Resident Names;	MyFieldCount1=5, because 'John' appears twice.
John1: Load FieldValueCount('Initials') as MyInitialsCount1 Resident Names;	MyFieldCount1=6, because 'Initials' only has distinct values.

Data used in example:

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
```

## 5 Functions in scripts and chart expressions

```
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 <b>match_field_value</b> in. Input value must be given as a string (for example, quoted literals).
match_field_value	Value to look up in <b>match_field_name</b> 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 <b>table_name</b> 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.

The sample data uses the **Lookup()** function in the following form:

```
Lookup('Category', 'ProductID', ProductID, 'ProductList')
```

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

```
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
```

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 **OrderData** table with the following result:

## 5 Functions in scripts and chart expressions

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



*The Lookup() function is flexible and can access any previously loaded table. However, it is slow compared with the Applymap() function.*

### See also:

[ApplyMap \(page 646\)](#)

## 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 <b>TOTAL</b> is used as argument, the current column segment is always equal to the entire column.

### Example:

```
if( RowNo( )= NoOfRows( ), 0, Above( sum( Sales )))
```

### See also:

 [RowNo - chart function \(page 403\)](#)

## Peek

**Peek()** finds the value of a field in a table for a row that has already been loaded or that exists in internal memory. The row number can be specified, as can the table.

### 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 <b>row</b> is stated, -1 is assumed.
table_name	A table label without the ending colon. If no <b>table_name</b> is stated, the current table is assumed. If used outside the <b>LOAD</b> statement or referring to another table, the <b>table_name</b> must be included.

### Limitations:

In the first record of an internal table, the function returns NULL.

### Example:

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

```
EmployeeDates:  
Load * Inline [  
EmployeeCode|StartDate|EndDate  
101|02/11/2010|23/06/2012  
102|01/11/2011|30/11/2013  
103|02/01/2012|
```

## 5 Functions in scripts and chart expressions

```
104|02/01/2012|31/03/2012
105|01/04/2012|31/01/2013
106|02/11/2013|
] (delimiter is '|');
```

```
FirstEmployee:
Load EmployeeCode, Peek('EmployeeCode',0) As EmpCode
Resident EmployeeDates;
```

EmpCode = 101, because `Peek('EmployeeCode',0)` returns the first 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)` returns the third value in the table: 103.

However, note that without specifying the table as the third argument **table\_no**, the function references the current (in this case, internal) table. The result of `Peek('EmployeeCode',-2)` is multiple values:

Resulting table

Employee code	EmpCode
101	-
102	-
103	101
104	102
105	103
106	104

### Example:

```
FirstEmployee:
Load EmployeeCode, Peek('EmployeeCode',-2,'EmployeeDates') As EmpCode
Resident EmployeeDates;
```

By specifying the argument **table\_no** as 'EmployeeDates', the function returns the second-to-last value of EmployeeCode in the table EmployeeDates: 105.

### Example:

The **Peek()** function can be used to reference data that is not yet loaded.

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

```
T1:
LOAD * inline [
ID|value
1|3
1|4
1|6
```



## 5 Functions in scripts and chart expressions

```
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	6	6
1	6,3	3
1	6,3,4	4
2	11	11
2	11,10	10
2	11,10,1	1
3	8	8
3	8,7	7
5	13	13
5	13,2	2
5	13,2,78	78

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:

```
LOAD A, B, numsum( B, Peek( 'Bsum' ) ) as Bsum...;  
Creates an accumulation of B in Bsum.
```

## 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 <b>previous()</b> 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
```

## 5 Functions in scripts and chart expressions

---

```
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
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 ]])
```

## 5 Functions in scripts and chart expressions

**Return data type:** dual

**Arguments:**

### Arguments

Argument	Description
expr	The expression or field containing the data to be measured.
offset	<p>Specifying an <b>offset</b> of n, greater than 1, moves the evaluation of the expression down n rows below the top row.</p> <p>Specifying a negative offset number makes the <b>Top</b> function work like the <b>Bottom</b> function with the corresponding positive offset number.</p>
count	By specifying a third parameter <b>count</b> greater than 1, the function will return a range of <b>count</b> values, one for each of the last <b>count</b> rows of the current column segment. In this form, the function can be used as an argument to any of the special range functions. <i>Range functions (page 653)</i>
TOTAL	If the table is one-dimensional or if the qualifier <b>TOTAL</b> 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**.

## 5 Functions in scripts and chart expressions

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	Q	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).*

## 5 Functions in scripts and chart expressions

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 <b>Top</b> function can be used as input to the range functions. For example: RangeAvg (Top(Sum (Sales),1,3)).	<p>In the arguments for the <b>Top()</b> function, offset is set to 1 and count is set to 3. The function finds the results of the expression <b>Sum(Sales)</b> 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.</p> <p>A table with <b>Customer</b> as dimension gives the following results for the RangeAvg() expression.</p>								
	<table><tr><td>Astrida</td><td>603</td></tr><tr><td>Betacab</td><td>603</td></tr><tr><td>Canutility</td><td>603</td></tr><tr><td>Divadip:</td><td>603</td></tr></table>	Astrida	603	Betacab	603	Canutility	603	Divadip:	603
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 \(page 613\)](#)
-  [Above - chart function \(page 605\)](#)
-  [Sum - chart function \(page 195\)](#)
-  [RangeAvg \(page 656\)](#)
-  [Range functions \(page 653\)](#)

## 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	<p>Specifying an <b>offset</b> n, greater than 1 moves the evaluation of the expression n rows further to the right from the current row.</p> <p>Specifying an offset of 0 will evaluate the expression on the current row.</p> <p>Specifying a negative offset number makes the <b>After</b> function work like the <b>Before</b> function with the corresponding positive offset number.</p>
count	By specifying a third parameter <b>count</b> greater than 1, the function will return a range of values, one for each of the table rows up to the value of <b>count</b> , counting to the right from the original cell.
TOTAL	If the table is one-dimensional or if the qualifier <b>TOTAL</b> 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	<p>Specifying an <b>offset</b> n, greater than 1 moves the evaluation of the expression n rows further to the left from the current row.</p> <p>Specifying an offset of 0 will evaluate the expression on the current row.</p> <p>Specifying a negative offset number makes the <b>Before</b> function work like the <b>After</b> function with the corresponding positive offset number.</p>
count	By specifying a third parameter <b>count</b> greater than 1, the function will return a range of values, one for each of the table rows up to the value of <b>count</b> , counting to the left from the original cell.
TOTAL	If the table is one-dimensional or if the qualifier <b>TOTAL</b> 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	<p>Specifying an <b>offset</b> n, greater than 1 moves the evaluation of the expression n rows further to the right from the current row.</p> <p>Specifying an offset of 0 will evaluate the expression on the current row.</p> <p>Specifying a negative offset number makes the <b>First</b> function work like the <b>Last</b> function with the corresponding positive offset number.</p>
count	By specifying a third parameter <b>count</b> greater than 1, the function will return a range of values, one for each of the table rows up to the value of <b>count</b> , counting to the right from the original cell.
TOTAL	If the table is one-dimensional or if the qualifier <b>TOTAL</b> 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	<p>Specifying an <b>offset</b> n, greater than 1 moves the evaluation of the expression n rows further to the left from the current row.</p> <p>Specifying an offset of 0 will evaluate the expression on the current row.</p> <p>Specifying a negative offset number makes the <b>First</b> function work like the <b>Last</b> function with the corresponding positive offset number.</p>
count	By specifying a third parameter <b>count</b> greater than 1, the function will return a range of values, one for each of the table rows up to the value of <b>count</b> , counting to the left from the original cell.
TOTAL	If the table is one-dimensional or if the qualifier <b>TOTAL</b> 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 <b>TOTAL</b> is used as argument, the current column segment is always equal to the entire column.

## 5 Functions in scripts and chart expressions

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 <b>TOTAL</b> 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 ] )
```

## 5 Functions in scripts and chart expressions

### 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 <b>mapping load</b> or the <b>mapping select</b> statement. Its name must be enclosed by single, straight quotation marks. <div> <i>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.</i></div>
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
```

## 5 Functions in scripts and chart expressions

---

```
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	<p>The name of a mapping table previously read by a <b>mapping load</b> or a <b>mapping select</b> statement. The name must be enclosed by single straight quotation marks.</p> <div><i>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.</i></div>
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  $\pi$  (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
<code>EmptyIsNull(AdditionalComments)</code>	This expression will return as null any empty string values of the <i>AdditionalComments</i> field, instead of empty strings. Non-empty strings and numbers are returned.
<code>EmptyIsNull(PurgeChar(PhoneNumber, ' -()'))</code>	This expression will strip any dashes, spaces and parentheses from the <i>PhoneNumber</i> field. If there are no characters left, the <code>EmptyIsNull</code> 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];
```

## 5 Functions in scripts and chart expressions

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

ID	Value	ValueNullConv	IsNull
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.

---

## 5 Functions in scripts and chart expressions

---

```
RangeOnly(first_expr[, Expression])
```

RangeSum

**RangeSum()** returns the sum of a range of values. All non-numeric values are treated as 0, unlike the + operator.

```
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])
```

---

## 5 Functions in scripts and chart expressions

---

### 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 \(page 601\)](#)

## 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
RangeAvg (1, 'xyz')	Returns 1
RangeAvg (null( ), 'abc')	Returns NULL

### Example:

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

RangeTab3:

```
LOAD recno() as RangeID, RangeAvg(Field1,Field2,Field3) as MyRangeAvg INLINE [  
Field1, Field2, Field3  
10,5,6  
2,3,7
```



## 5 Functions in scripts and chart expressions

```
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 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	6.666666667	The equivalent to <code>RangeAvg(10,2,8)</code>
18	9.333333333	-
5	10.333333333	-
9	10.666666667	-

```
RangeTab:
LOAD * INLINE [
MyField
10
```

```
2
8
18
5
9
1 ;
```

### See also:

-  [Avg - chart function \(page 234\)](#)
-  [Count - chart function \(page 200\)](#)

## 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 <b>x-value</b> or a range of <b>y-values</b> .
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
<code>RangeCorrel (2,3,6,8,9,4,8,5)</code>	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. Then add the fields listed in the results column to a sheet in your app to see the result.

```
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.


## 5 Functions in scripts and chart expressions

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 \(page 236\)](#)

## 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
<code>RangeCount (1,2,4)</code>	Returns 3

## 5 Functions in scripts and chart expressions

Examples	Results
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. Then add the fields listed in the results column to a sheet in your app to see the result.

```
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


## 5 Functions in scripts and chart expressions

MyField	RangeCount(Above(MyField,1,3))
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 \(page 200\)](#)

### 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.

## 5 Functions in scripts and chart expressions

Argument	Description
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. Then add the fields listed in the results column to a sheet in your app to see the result.

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))
```

## 5 Functions in scripts and chart expressions

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 \(page 605\)](#)
-  [Fractile - chart function \(page 240\)](#)

### 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])
```



## 5 Functions in scripts and chart expressions

**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														
<code>RangeIRR(-70000,12000,15000,18000,21000,26000)</code>	Returns 0.0866														
<p>Add the example script to your app and run it. Then add the fields listed in the results column to a sheet in your app to see the result.</p> <pre>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 ' ');</pre>	<p>The resulting table shows the returned values of RangeIRR for each of the records in the table.</p> <table><tr><th>RangeID</th><th>RangeIRR</th></tr><tr><td>1</td><td>0.0639</td></tr><tr><td>2</td><td>0.8708</td></tr><tr><td>3</td><td>-</td></tr><tr><td>4</td><td>5.8419</td></tr><tr><td>5</td><td>0.9318</td></tr><tr><td>6</td><td>-0.2566</td></tr></table>	RangeID	RangeIRR	1	0.0639	2	0.8708	3	-	4	5.8419	5	0.9318	6	-0.2566
RangeID	RangeIRR														
1	0.0639														
2	0.8708														
3	-														
4	5.8419														
5	0.9318														
6	-0.2566														

**See also:**

 [Inter-record functions \(page 601\)](#)

### RangeKurtosis

**RangeKurtosis()** returns the value that corresponds to the kurtosis of a range of numbers.

**Syntax:**

```
RangeKurtosis(first_expr[, Expression])
```

## 5 Functions in scripts and chart expressions

---

**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 \(page 248\)](#)

## 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.

## 5 Functions in scripts and chart expressions

---

### 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. Then add the fields listed in the results column to a sheet in your app to see the result.

```
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))
```

## 5 Functions in scripts and chart expressions

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 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.

## 5 Functions in scripts and chart expressions

### 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 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 \(page 359\)](#)

## 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. Then add the fields listed in the results column to a sheet in your app to see the result.

```
RangeTab3:  
LOAD recno() as RangeID, RangeMin(Field1,Field2,Field3) as MyRangeMin INLINE [  
Field1, Field2, Field3  
10,5,6
```

## 5 Functions in scripts and chart expressions

```
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 \(page 186\)](#)

## 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 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 \(page 362\)](#)

### 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.

## 5 Functions in scripts and chart expressions

### 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 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 \(page 204\)](#)

## 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

## 5 Functions in scripts and chart expressions

---

### Example:

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

```
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	MyRangeMode
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 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 \(page 190\)](#)

## 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 689)*.

### 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														
<p>Add the example script to your app and run it. Then add the fields listed in the results column to a sheet in your app to see the result.</p> <pre>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 ' ');</pre>	<p>The resulting table shows the returned values of RangeNPV for each of the records in the table.</p> <table><tr><th>RangeID</th><th>RangeNPV</th></tr><tr><td>1</td><td>\$-49.13</td></tr><tr><td>2</td><td>\$777.78</td></tr><tr><td>3</td><td>\$98.77</td></tr><tr><td>4</td><td>\$25.51</td></tr><tr><td>5</td><td>\$250.83</td></tr><tr><td>6</td><td>\$20.40</td></tr></table>	RangeID	RangeNPV	1	\$-49.13	2	\$777.78	3	\$98.77	4	\$25.51	5	\$250.83	6	\$20.40
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 \(page 601\)](#)

## 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.

## 5 Functions in scripts and chart expressions

### Examples and results:

Function examples

Examples	Results
<code>RangeNullCount (1,2,4)</code>	Returns 0
<code>RangeNullCount (5, 'abc')</code>	Returns 0
<code>RangeNullCount (null( ), null( ))</code>	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	<code>RangeNullCount(Above(Sum(MyField),0,3))</code>
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 \(page 206\)](#)

## RangeNumericCount

**RangeNumericCount()** finds the number of numeric values in an expression or field.

### Syntax:

```
RangeNumericCount(first_expr[, Expression])
```

## 5 Functions in scripts and chart expressions

**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 example works as expected.*

Sample data

MyField	RangeNumericCount(Above(MaxString(MyField),0,3))
10	1
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 \(page 209\)](#)

## 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
<code>RangeOnly (1,2,4)</code>	Returns NULL
<code>RangeOnly (5,'abc')</code>	Returns NULL
<code>RangeOnly (null( ), 'abc')</code>	Returns 'abc'
<code>RangeOnly(10,10,10)</code>	Returns 10

---

### See also:

 [Only - chart function \(page 193\)](#)

## RangeSkew

**RangeSkew()** returns the value corresponding to the skewness of a range of numbers.

## 5 Functions in scripts and chart expressions

### 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 \(page 274\)](#)

### 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

## 5 Functions in scripts and chart expressions

Examples	Results
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 \(page 277\)](#)

### RangeSum

**RangeSum()** returns the sum of a range of values. All non-numeric values are treated as 0, unlike the + operator.

**Syntax:**

```
RangeSum (first_expr[, Expression])
```

## 5 Functions in scripts and chart expressions

**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, unlike the **+** operator.

### 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. Then add the fields listed in the results column to a sheet in your app to see the result.

```
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

## 5 Functions in scripts and chart expressions

RangeID	MyRangeSum
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 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 \(page 195\)](#)
-  [Above - chart function \(page 605\)](#)

## 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 example works as expected.*

## 5 Functions in scripts and chart expressions

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 \(page 212\)](#)

### 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
<code>RangeXIRR(-2500, '2008-01-01', 2750, '2008-09-01')</code>	Returns 0.1532

### See also:

 [RangeIRR \(page 664\)](#)

## 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.

## 5 Functions in scripts and chart expressions

Example table

Examples	Results														
RangeXNPV(0.1, -2500, '2008-01-01', 2750, '2008-09-01')	Returns 80.25														
<p>Add the example script to your app and run it. Then add the fields listed in the results column to a sheet in your app to see the result.</p> <pre>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 ' ');</pre>	<p>The resulting table shows the returned values of RangeXNPV for each of the records in the table.</p> <table><thead><tr><th>RangeID</th><th>RangeXNPV</th></tr></thead><tbody><tr><td>1</td><td>\$-49.13</td></tr><tr><td>2</td><td>\$777.78</td></tr><tr><td>3</td><td>\$98.77</td></tr><tr><td>4</td><td>\$25.51</td></tr><tr><td>5</td><td>\$250.83</td></tr><tr><td>6</td><td>\$20.40</td></tr></tbody></table>	RangeID	RangeXNPV	1	\$-49.13	2	\$777.78	3	\$98.77	4	\$25.51	5	\$250.83	6	\$20.40
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

## 5 Functions in scripts and chart expressions

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.

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 <b>TOTAL</b> qualifier, the function is evaluated across 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:

## 5 Functions in scripts and chart expressions

---

**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:

**fmt**examples

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.

## 5 Functions in scripts and chart expressions

### Rank examples

Examples	Results
Example 1. Create a table with the dimensions Customer and Sales and the measure Rank(Sales)	<p>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.</p> <p>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.</p>
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 <b>mode</b> and <b>fmt</b> 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 \(page 195\)](#)

### 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.

## 5 Functions in scripts and chart expressions

### 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 <b>TOTAL</b> qualifier, the function is evaluated across 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:



## 5 Functions in scripts and chart expressions

### 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 )
```

## 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:

## 5 Functions in scripts and chart expressions

```
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	<p>The optional normalization method applied to datasets before KMeans clustering.</p> <p>Possible values:</p> <p>0 or 'none' for no normalization</p> <p>1 or 'zscore' for z-score normalization</p> <p>2 or 'minmax' for min-max normalization</p> <p>If no parameter is supplied or if the supplied parameter is incorrect, no normalization is applied.</p> <p>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.</p> <p>Min-max normalization ensures that the features have the same scale by taking the minimum and maximum values of each and recalculating each datapoint.</p>

**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:

## 5 Functions in scripts and chart expressions

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*

**Data**

**Dimensions**

Bubble

Id > [grid icon]

Alternative dimensions

Add alternative

**Measures**

X-axis

Sum [petal.length] > [grid icon]

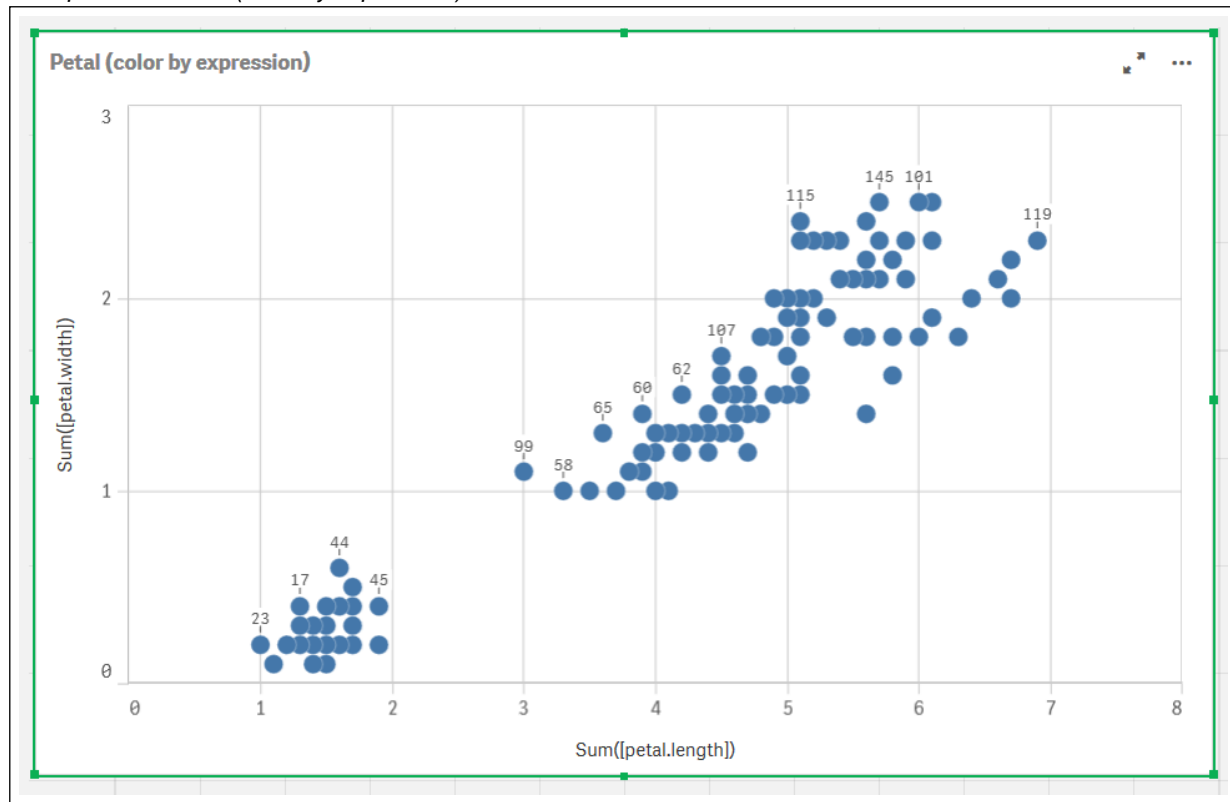
Y-axis

Sum [petal.width] > [grid icon]

The data points are plotted on the chart.

## 5 Functions in scripts and chart expressions

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`

## 5 Functions in scripts and chart expressions

---

*Appearance settings for Petal (color by expression) chart*

Appearance


▼ Colors and legend

Colors ☐

Custom

By expression ▼

Expression


kmeans2d(\$(KmeansPetalC 


☐ The expression is a color code

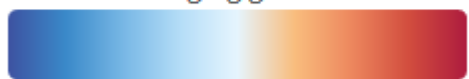
Label

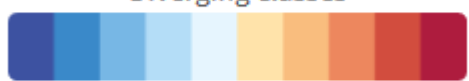
Cluster Id

Color scheme

Sequential gradient 

Sequential classes 

Diverging gradient 

Diverging classes 

☐ Reverse colors

Range ☒

Auto

Show legend ☒

Auto

Legend position

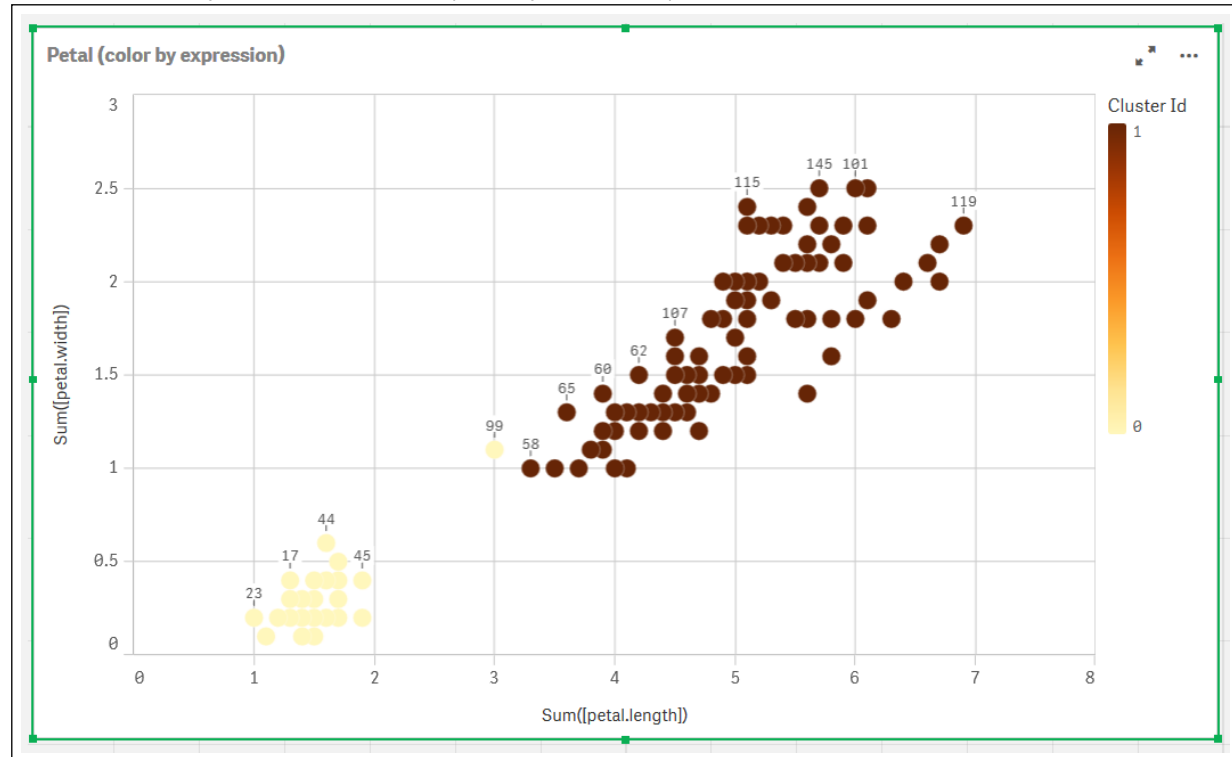
Horizontal

☒ Show legend title

## 5 Functions in scripts and chart expressions

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,


## 5 Functions in scripts and chart expressions

---

*Appearance for Clusters variable input box*



▼ General

Show titles 

On

Title

Clusters

*fx*

Subtitle

*fx*

Footnote

*fx*

☐ Disable hover menu

▼ Variable

Name

KmeansPetalClusters

▼

Show as

Slider

▼

☐ Update on drag

▼ Values

Min

2

*fx*

Max

10

*fx*

Step

1

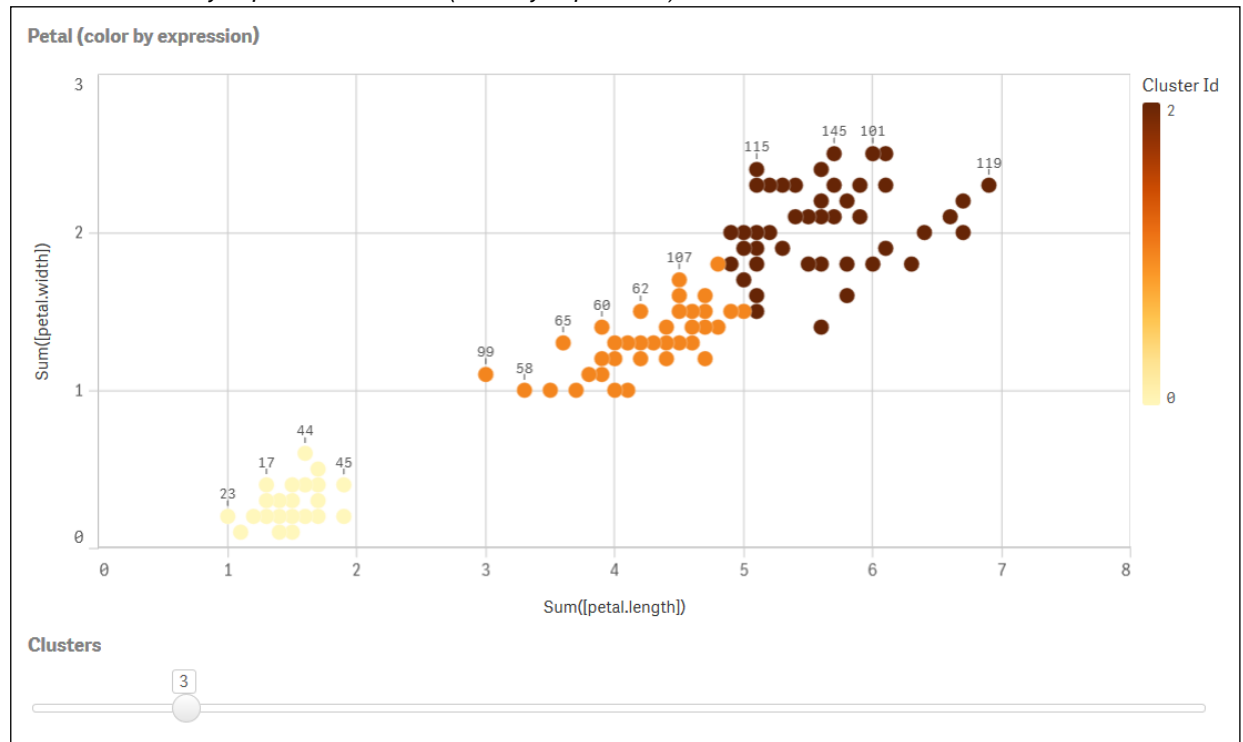
*fx*

☒ Slider label

## 5 Functions in scripts and chart expressions

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*

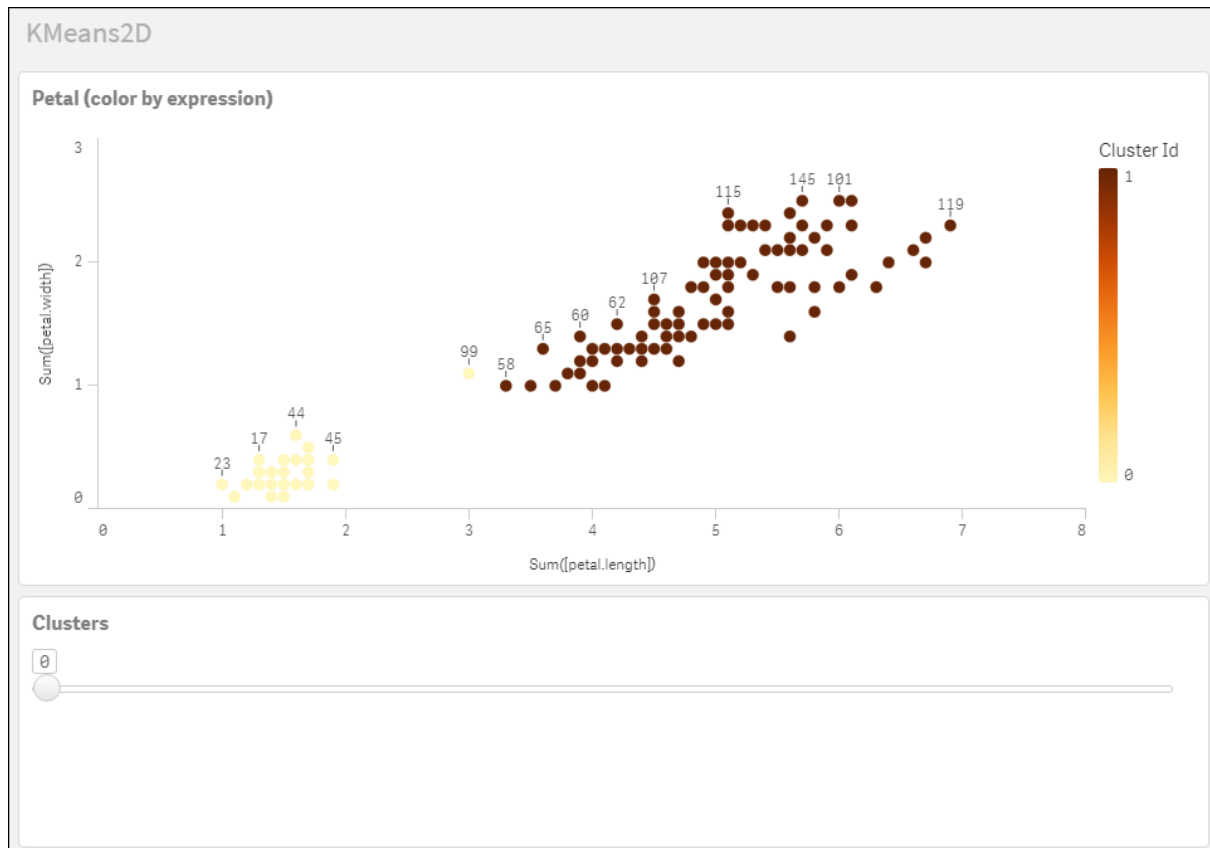


### Auto-clustering

**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*

## 5 Functions in scripts and chart expressions



### 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 Functions in scripts and chart expressions

---

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

## 5 Functions in scripts and chart expressions

---

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
<code>num_clusters</code>	Integer that specifies the number of clusters.
<code>num_iter</code>	The number of iterations of clustering with reinitialized cluster centers.
<code>coordinate_1</code>	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

---

## 5 Functions in scripts and chart expressions

---

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*

Data

Dimensions

Bubble

Id

>

Alternative dimensions

Add alternative

Measures

X-axis

Sum

[petal.length]

>

Y-axis

Sum

[petal.width]

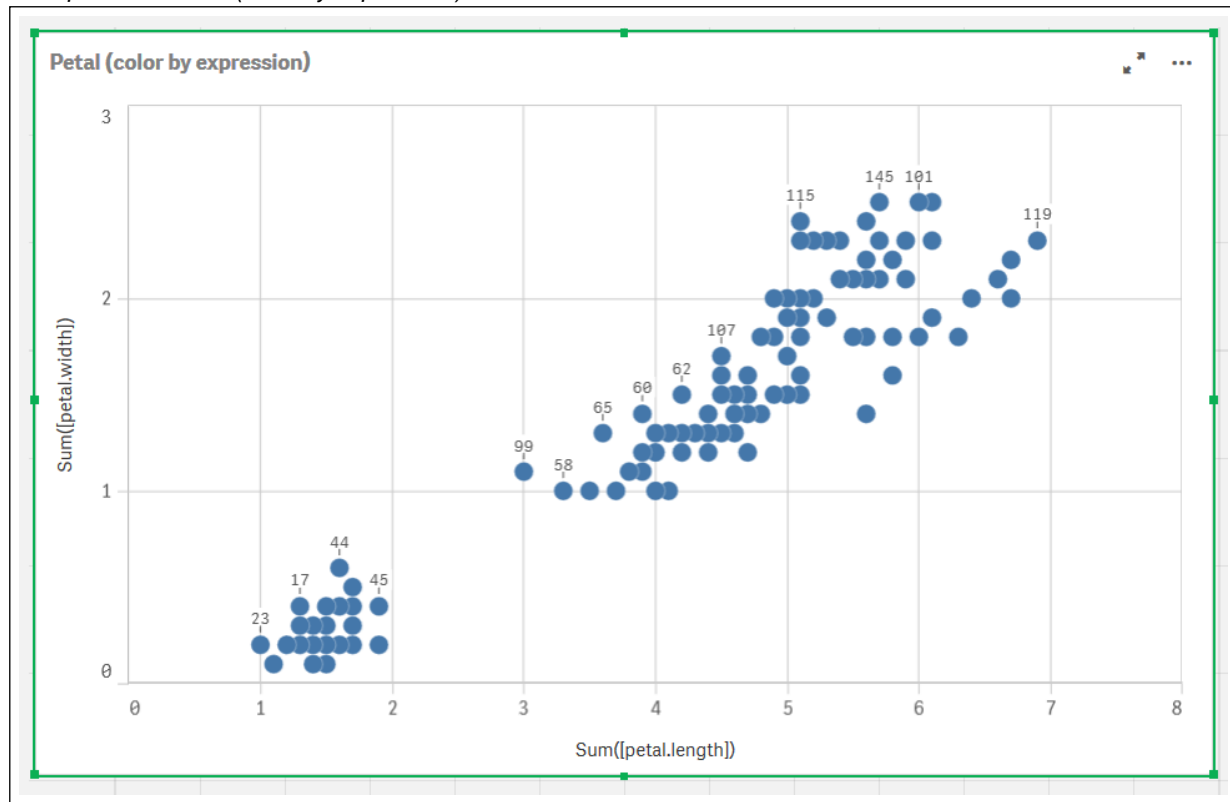
>

The data points are plotted on the chart.



## 5 Functions in scripts and chart expressions

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`

## 5 Functions in scripts and chart expressions

---

*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 ☒

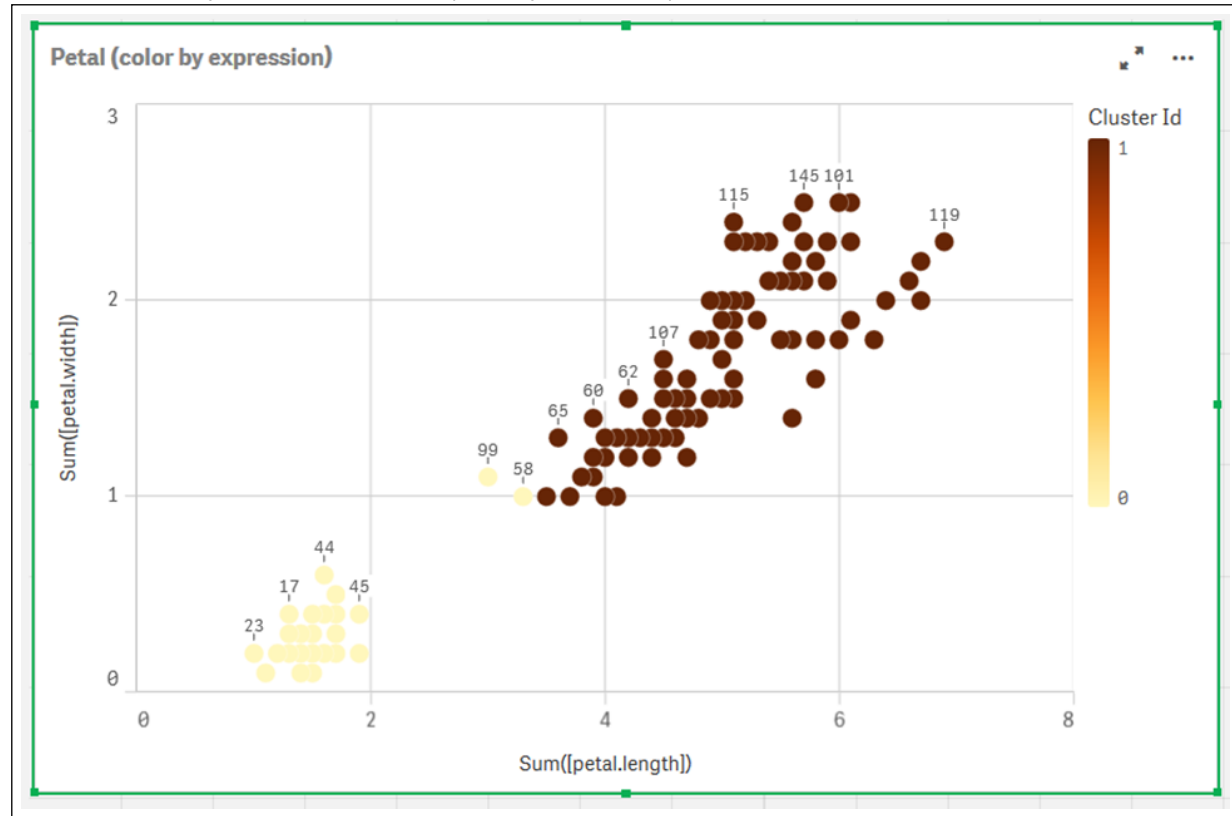
Auto

Legend position

## 5 Functions in scripts and chart expressions

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,

## 5 Functions in scripts and chart expressions

---

*Appearance for Clusters variable input box*

▼ General

Show titles   
On

Title

Clusters

*fx*

Subtitle

*fx*

Footnote

*fx*

☐ Disable hover menu

▼ Variable

Name

KmeansPetalClusters

▼

Show as

Slider

▼

☐ Update on drag

▼ Values

Min

2

*fx*

Max

10

*fx*

Step

1

*fx*

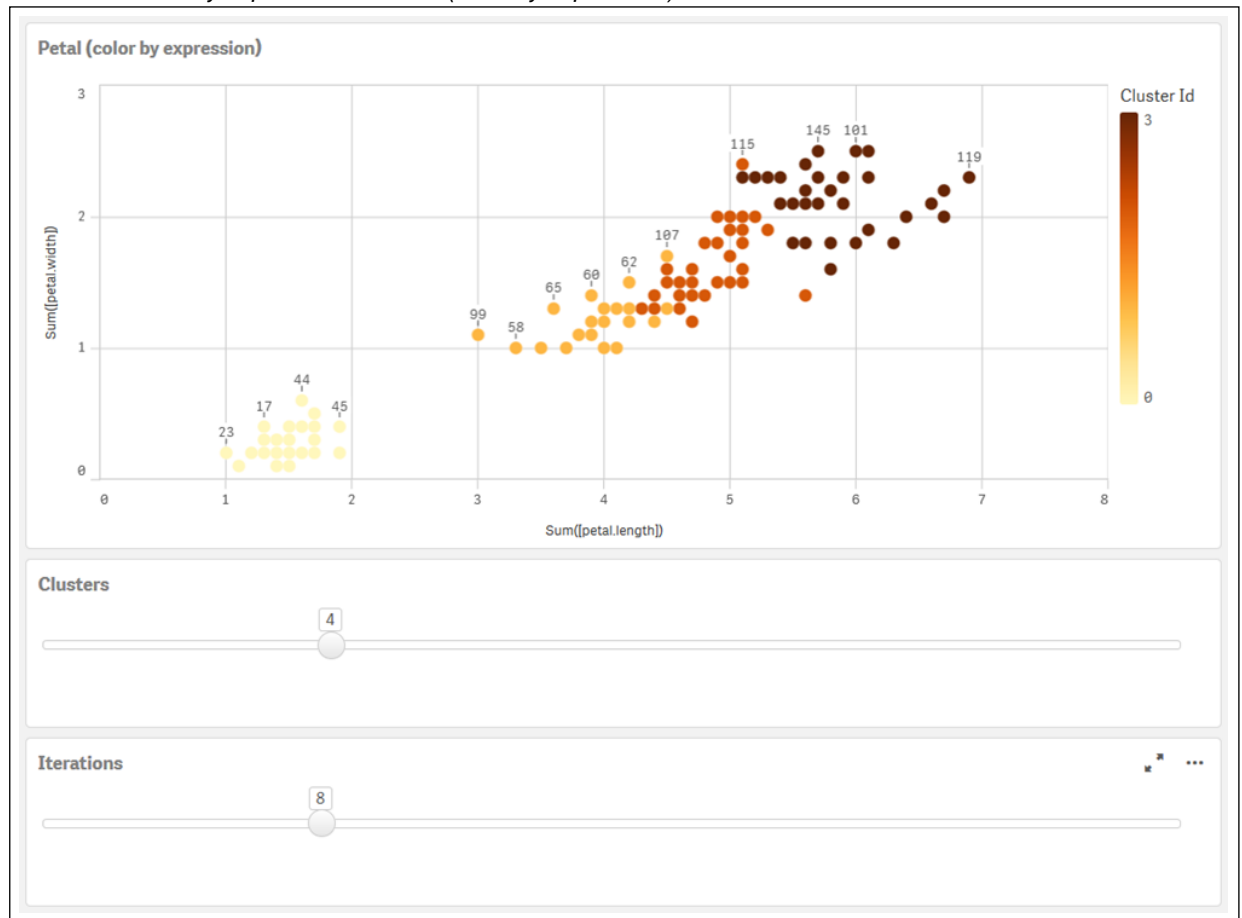
☒ Slider label

## 5 Functions in scripts and chart expressions

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*

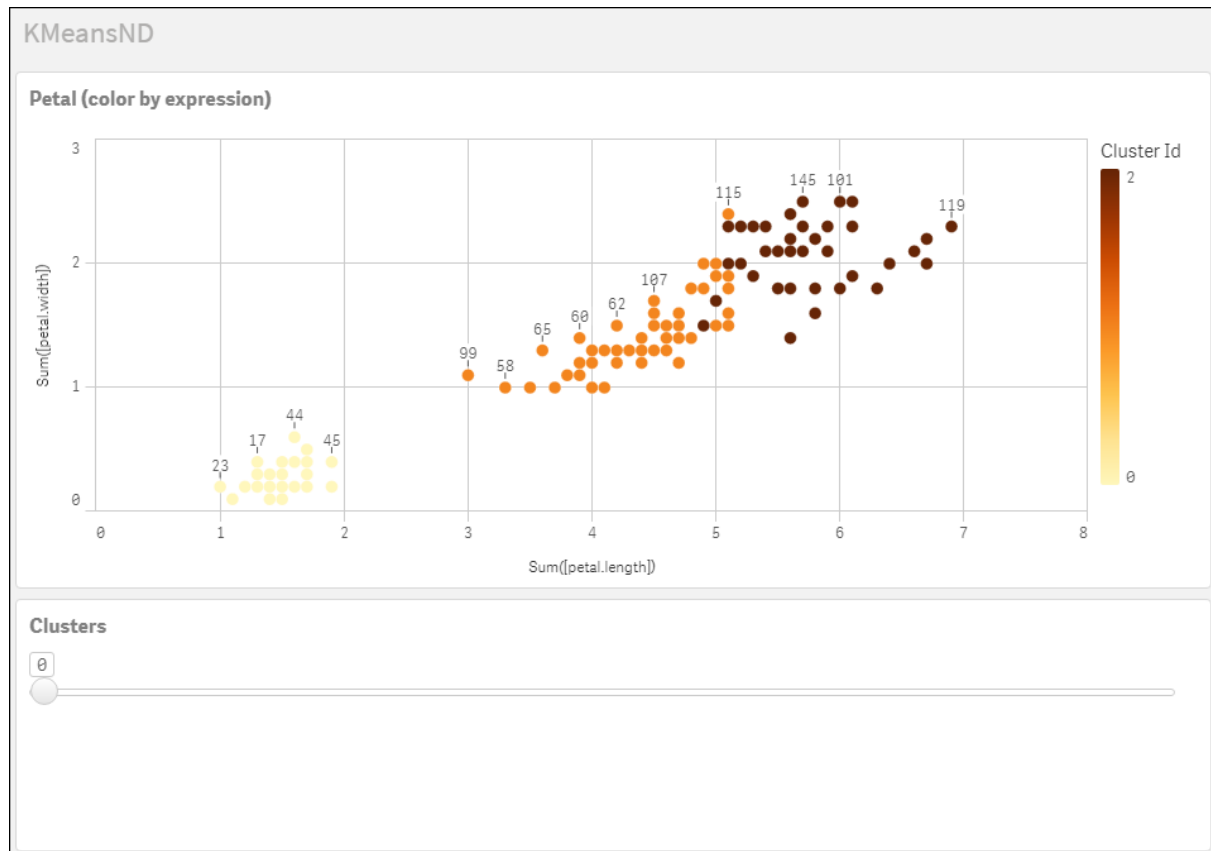


### Auto-clustering

**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.

## 5 Functions in scripts and chart expressions

*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 Functions in scripts and chart expressions

---

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

## 5 Functions in scripts and chart expressions

---

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
<code>num_clusters</code>	Integer that specifies the number of clusters.
<code>coordinate_no</code>	The desired coordinate number of the centroids (corresponding, for example, to the x, y, or z axis).
<code>coordinate_1</code>	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, <code>coordinate_2</code> , calculates the second coordinate.

Argument	Description
norm	<p>The optional normalization method applied to datasets before KMeans clustering.</p> <p>Possible values:</p> <p>0 or 'none' for no normalization</p> <p>1 or 'zscore' for z-score normalization</p> <p>2 or 'minmax' for min-max normalization</p> <p>If no parameter is supplied or if the supplied parameter is incorrect, no normalization is applied.</p> <p>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.</p> <p>Min-max normalization ensures that the features have the same scale by taking the minimum and maximum values of each and recalculating each datapoint.</p>

### 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  $\chi^2$  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 \(page 226\)](#)

## CHIDIST

**CHIDIST()** returns the one-tailed probability of the  $\chi^2$  distribution. The  $\chi^2$  distribution is associated with a  $\chi^2$  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
<code>CHIINV(0.9237827, 15)</code>	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. <b>Value</b> 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.



## 5 Functions in scripts and chart expressions

---

This function is related to the **FINV** function in the following way:

If  $\text{prob} = \text{FDIST}(\text{value}, \text{df1}, \text{df2})$ , then  $\text{FINV}(\text{prob}, \text{df1}, \text{df2}) = \text{value}$

### Limitations:

All arguments must be numeric, else NULL will be returned.

Examples and results:

Example	Result
<code>FDIST(15, 8, 6)</code>	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  $\text{prob} = \text{FDIST}(\text{value}, \text{df1}, \text{df2})$ , then  $\text{FINV}(\text{prob}, \text{df1}, \text{df2}) = \text{value}$

### Limitations:

All arguments must be numeric, else NULL will be returned.

Examples and results:

Example	Result
<code>FINV(0.0019369, 8, 6)</code>	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  $\text{prob} = \text{NORMDIST}(\text{value}, m, sd)$ , then  $\text{NORMINV}(\text{prob}, m, sd) = \text{value}$

**Limitations:**

All arguments must be numeric, else NULL will be returned.

Examples and results:

Example	Result
<code>NORMINV( 0.6914625, 0, 1 )</code>	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.

## 5 Functions in scripts and chart expressions

Argument	Description
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  $\text{prob} = \text{TDIST}(\text{value}, \text{df}, 2)$ , then  $\text{TINV}(\text{prob}, \text{df}) = \text{value}$

### Limitations:

All arguments must be numeric, else NULL will be returned.

Examples and results:

Example	Result
$\text{TDIST}(1, 30, 2)$	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  $\text{prob} = \text{TDIST}(\text{value}, \text{df}, 2)$ , then  $\text{TINV}(\text{prob}, \text{df}) = \text{value}$ .

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. In the functions below, the parameters are expressions where **s** should be interpreted as a string.

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})
```

---

## 5 Functions in scripts and chart expressions

---

### 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])
```

### 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 (left-most) 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)
```

---

## 5 Functions in scripts and chart expressions

---

### 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 (right-most) 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.

---

## 5 Functions in scripts and chart expressions

---

```
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

Examples and results:

Example	Result
Capitalize ( 'star trek' )	Returns 'Star Trek'
Capitalize ( 'AA bb cC Dd' )	Returns 'Aa Bb Cc Dd'

## 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 can not be used in chart expressions.*

Examples and results:

Example	Result
Evaluate ( 5 * 8 )	Returns '40'

### 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.

Examples and results:

Example	Result
FindOneOf( 'my example text string', 'et%s')	Returns '4'.
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, and is in position 12.
FindOneOf( 'my example text string', 'æ%&')	Returns '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:**

```
Hash128 ( 'abc', 'xyz', '123' )  
Hash128 ( Region, Year, Month )
```

### 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:**

```
Hash160 ( 'abc', 'xyz', '123' )  
Hash160 ( Region, Year, Month )
```

### 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:**

```
Hash256 ( 'abc', 'xyz', '123' )  
Hash256 ( Region, Year, Month )
```

### 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 <b>substring</b> to search for. For example, a value of 2 searches for the second occurrence.

## 5 Functions in scripts and chart expressions

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 <b>Date</b> = 1997-07-14	Returns 1997
Mid( Date, Index( Date, '-', 2 ) -2, 2 ) where <b>Date</b> = 1997-07-14	Returns 07

### 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.

Examples and results:

Example	Result
KeepChar ( 'a1b2c3', '123' )	Returns '123'.
KeepChar ( 'a1b2c3', '1234' )	Returns '123'.
KeepChar ( 'a1b22c3', '1234' )	Returns '1223'.
KeepChar ( 'a1b2c3', '312' )	Returns '123'.

**See also:**

 [PurgeChar \(page 744\)](#)

### Left

**Left()** returns a string consisting of the first (left-most) 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 <b>text</b> .

Examples and results:

Example	Result
Left('abcdef', 3)	Returns 'abc'

 See also *Index* (page 739), which allows more complex string analysis.

### Len

**Len()** returns the length of the input string.

**Syntax:**

```
Len(text)
```

**Return data type:** integer

Examples and results:

Example	Result
Len('Peter')	Returns '5'

### 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

Examples and results:

Example	Result
LevenshteinDist('Kitten', 'Sitting')	Returns '3'

### Lower

**Lower()** converts all the characters in the input string to lower case.

**Syntax:**

```
Lower(text)
```

**Return data type:** string

Examples and results:

Example	Result
Lower('abcD')	Returns 'abcd'

### LTrim

**LTrim()** returns the input string trimmed of any leading spaces.

**Syntax:**

```
LTrim(text)
```

**Return data type:** string

Examples and results:

Example	Result
LTrim( ' abc' )	Returns 'abc'
LTrim( 'abc ' )	Returns 'abc '

**See also:**

📄 [RTrim \(page 746\)](#)

### 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 <b>start</b> are included.

**Examples and results:**

Example	Result
Mid('abcdef', 3 )	Returns 'cdef'
Mid('abcdef', 3, 2 )	Returns 'cd'

**See also:**

 [Index \(page 739\)](#)

### 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	Result
<code>ord('A')</code>	Returns the integer 65.
<code>ord('Ab')</code>	Returns the integer 65.

### 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

Examples and results:

Example	Result
<code>PurgeChar ( 'a1b2c3', '123' )</code>	Returns 'abc'
<code>PurgeChar ( 'a1b2c3', '312' )</code>	Returns 'abc'

**See also:**

 [KeepChar \(page 740\)](#)

### 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 <b>text</b> are to be repeated in the output string.

Examples and results:

Example	Result
Repeat( ' * ', rating ) when <b>rating</b> = 4	Returns '*****'

## 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 which may occur one or more times within the input string <b>text</b> .
to_str	The string that will replace all occurrences of <b>from_str</b> within the string <b>text</b> .

Examples and results:

Example	Result
Replace('abccde', 'cc', 'xyz')	Returns 'abxyzde'

**See also:**

### Right

**Right()** returns a string consisting of the last (right-most) 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 right-hand part of the string <b>text</b> .

Examples and results:

Example	Result
Right('abcdef', 3)	Returns 'def'

### RTrim

**RTrim()** returns the input string trimmed of any trailing spaces.

**Syntax:**


```
RTrim(text)
```

**Return data type:** string

Examples and results:

Example	Result
RTrim( ' abc' )	Returns 'abc'
RTrim( 'abc ' )	Returns 'abc'

**See also:**

 [LTrim \(page 742\)](#)

## 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 <b>text</b> that divides the string into component parts.
field_no	<p>The optional third argument is an integer that specifies which of the substrings of the parent string <b>text</b> is to be returned. Use the value 1 to return the first substring, 2 to return the second substring, and so on.</p> <ul style="list-style-type: none"><li>• If <b>field_no</b> is a positive value, substrings are extracted from left to right.</li><li>• If <b>field_no</b> is a negative value, substrings are extracted from right to left.</li></ul>



*SubField() can be used instead of using complex combinations of functions such as Len(), Right(), Left(), Mid(), and other string functions.*

### Examples and results:

Example	Result
<code>SubField(S, ';' ,2)</code>	Returns 'cde' if <b>S</b> is 'abc;cde;efg'.

## 5 Functions in scripts and chart expressions

Example	Result
<code>SubField(S, ';' ,1)</code>	Returns an empty string if <b>S</b> is an empty string.
<code>SubField(S, ';' ,1)</code>	Returns an empty string if <b>S</b> is ';'. Returns the substring starting from the first ';' and ending at the first space.
Suppose you have a variable that holds a path name vMyPath,  Set vMyPath=\Users\ext_jrb\Documents\Qlik\Sense\Apps;  <code>SubField(vMyPath, '\', -3)</code>	In a text & image chart, you can add a measure such as: <code>SubField(vMyPath, '\', -3)</code> , which results in 'Qlik', because it is the substring third from the right-hand end of the variable vMyPath.

### Example:

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

```
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;
```

Resulting table

Name	FirstName	Surname
Dave Owen	Dave	Owen
Joe Tem	Joe	Tem

### Example:

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 record.

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

```
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 '|');
```

Resulting table

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

### 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 <b>text</b> .

## 5 Functions in scripts and chart expressions

Examples and results:

Example	Result
<code>substringCount ( 'abcdefgcdxyz', 'cd' )</code>	Returns '2'
<code>substringCount ( 'abcdefgcdxyz', 'dc' )</code>	Returns '0'

### 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 <b>text</b> .
delimiter2	Specifies the second delimiting character (or string) to search for in <b>text</b> .
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.

Examples and results:

Example	Result
<code>TextBetween('&lt;abc&gt;', '&lt;', '&gt;')</code>	Returns 'abc'
<code>TextBetween('&lt;abc&gt;&lt;de&gt;', '&lt;', '&gt;', 2)</code>	Returns 'de'
<code>TextBetween('abc', '&lt;', '&gt;')</code> <code>TextBetween('&lt;a&lt;b', '&lt;', '&gt;')</code>	Both examples return NULL.  If any of the delimiter is not found in the string, NULL is returned.
<code>TextBetween('&lt;&gt;', '&lt;', '&gt;')</code>	Returns a zero-length string.
<code>TextBetween('&lt;abc&gt;', '&lt;', '&gt;', 2)</code>	Returns NULL, as n is larger than the occurrence of the delimiters.

### Trim

**Trim()** returns the input string trimmed of any leading and trailing spaces.

### Syntax:

```
Trim(text)
```

**Return data type:** string

Examples and results:

Example	Result
Trim( ' abc' )	Returns 'abc'
Trim( 'abc ' )	Returns 'abc'
Trim( ' abc ' )	Returns 'abc'

## 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

Examples and results:

Example	Result
Upper(' abcd')	Returns 'ABCD'

## 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.*

---

## 5 Functions in scripts and chart expressions

---

### 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.



## 5 Functions in scripts and chart expressions

**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

---

## 5 Functions in scripts and chart expressions

---

**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()
```

**See also:**

 [GetFolderPath \(page 546\)](#)

### 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.

---

## 5 Functions in scripts and chart expressions

---

```
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;
```

## 5 Functions in scripts and chart expressions

```
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
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  $-1 \leq x \leq 1$ . The result is a number between 0 and  $\pi$ .

```
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  $-1 \leq x \leq 1$ . The result is a number between  $-\pi/2$  and  $\pi/2$ .

```
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  $-\pi/2$  and  $\pi/2$ .

---

## 5 Functions in scripts and chart expressions

---

```
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  $-\pi$  and  $+\pi$ .

```
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 *,
```



## 5 Functions in scripts and chart expressions

---

```
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 absolute or relative paths in the data load script or functions and variables that expose the file system.

However, since absolute and relative 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.*

### 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 <i>a:</i> .

## 6 File system access restriction

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 <i>c:</i> 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: Library connection	Supported input: Library connection or absolute/relative path	The <b>Include/Must_Include</b> variable specifies a file that contains text that should be included in the script and evaluated as script code. 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: Library connection	Supported input: Library connection or absolute/relative path	The <b>binary</b> statement is used for loading data from another app.
Connect	Supported input: Library connection	Supported input: Library connection or absolute/relative path	The <b>CONNECT</b> 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: Library connection	Supported input: Library connection or absolute/relative path	The <b>Directory</b> statement defines which directory to look in for data files in subsequent <b>LOAD</b> statements, until a new <b>Directory</b> statement is made.
Execute	Not supported	Supported input: Library connection or absolute/relative path	The <b>Execute</b> 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: Library connection	Supported input: Library connection or absolute/relative path	Returns the browse string to the Qlik Sense executable.
Store into ...	Supported input: Library connection	Supported input: Library connection or absolute/relative path	Returns the root directory of the Qlik Sense executable.

### Script control statements

Script control statements

Statement	Standard mode	Legacy mode	Definition
For each...  filelist mask/dirlist mask	Supported input: Library connection  Returned output: Library connection	Supported input: Library connection or absolute/relative path  Returned output: Library connection or absolute path, depending on input	The filelist mask syntax produces a comma separated list of all files in the current directory matching the <b>filelist mask</b> . The <b>dirlist mask</b> 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: Library connection	Supported input: Library connection or absolute/relative path	Returns the value of the meta tags of different media files as text.
ConnectionString()	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 absolute path, depending on input	The <b>FileDir</b> function returns a string containing the path to the directory of the table file currently being read.
FilePath()	Returned output: Library connection	Returned output: Library connection or absolute path, depending on input	The <b>FilePath</b> function returns a string containing the full path to the table file currently being read.

## 6 File system access restriction

Function	Standard mode	Legacy mode	Definition
FileSize()	Supported input: Library connection	Supported input: Library connection or absolute/relative path	The <b>FileSize</b> 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: Library connection	Supported input: Library connection or absolute/relative path	The <b>FileTime</b> function returns a timestamp for the date and time of the last modification of the file filename. If no filename is specified, the function will refer to the currently read table file.
GetFolderPath()	Not supported	Returned output: Absolute path	The <b>GetFolderPath</b> function returns the value of the Microsoft Windows <i>SHGetFolderPath</i> function. This function takes as input the name of a Microsoft Windows folder and returns the full path of the folder.
QvdCreateTime()	Supported input: Library connection	Supported input: Library connection or absolute/relative path	This script function returns the XML-header time stamp from a QVD file, if any is present, otherwise it returns NULL.
QvdFieldName()	Supported input: Library connection	Supported input: Library connection or absolute/relative path	This script function returns the name of field number <b>fieldno</b> in a QVD file. If the field does not exist NULL is returned.

## 6 File system access restriction

Function	Standard mode	Legacy mode	Definition
QvdNoOfFields()	Supported input: Library connection	Supported input: Library connection or absolute/relative path	This script function returns the number of fields in a QVD file.
QvdNoOfRecords()	Supported input: Library connection	Supported input: Library connection or absolute/relative path	This script function returns the number of records currently in a QVD file.
QvdTableName()	Supported input: Library connection	Supported input: Library connection or absolute/relative path	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)



# 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 <b>SQL</b> 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 <b>SQL</b> instead.
CustomConnect	Use <b>Custom Connect</b> 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 <b>Incremental</b> instead of: <ul style="list-style-type: none"><li>• <b>Inc</b> (not recommended)</li><li>• <b>Incr</b> (not recommended)</li></ul>

## 8 Functions and statements not recommended in Qlik Sense

---

Statement	Parameters
<b>LOAD</b>	<p>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:</p> <ul style="list-style-type: none"><li>• <b>Bottom</b></li><li>• <b>Cellvalue</b></li><li>• <b>Col</b></li><li>• <b>Colmatch</b></li><li>• <b>Colsplit</b></li><li>• <b>Colxtr</b></li><li>• <b>Compound</b></li><li>• <b>Contain</b></li><li>• <b>Equal</b></li><li>• <b>Every</b></li><li>• <b>Expand</b></li><li>• <b>Filters</b></li><li>• <b>Intarray</b></li><li>• <b>Interpret</b></li><li>• <b>Length</b></li><li>• <b>Longer</b></li><li>• <b>Numerical</b></li><li>• <b>Pos</b></li><li>• <b>Remove</b></li><li>• <b>Rotate</b></li><li>• <b>Row</b></li><li>• <b>Rowcnd</b></li><li>• <b>Shorter</b></li><li>• <b>Start</b></li><li>• <b>Strcnd</b></li><li>• <b>Top</b></li><li>• <b>Transpose</b></li><li>• <b>Unwrap</b></li><li>• <b>XML: XMLSAX and Pattern is Path</b></li></ul>

### 8.3 Functions not recommended in Qlik Sense

This table describes script and chart functions that are not recommended for use in Qlik Sense.

## 8 Functions and statements not recommended in Qlik Sense

---

Functions that are not recommended

Function	Recommendation
<b>NumAvg</b>	Use Range functions instead.
<b>NumCount</b>	<i>Range functions (page 653)</i>
<b>NumMax</b>	
<b>NumMin</b>	
<b>NumSum</b>	
<b>QliktechBlue</b> <b>QliktechGray</b>	Use other color functions instead. <b>QliktechBlue()</b> can be replaced by <b>RGB(8, 18, 90)</b> and <b>QliktechGray</b> can be replaced by <b>RGB(158, 148, 137)</b> to get the same colors.  <i>Color functions (page 372)</i>
<b>QlikViewVersion</b>	Use <b>EngineVersion</b> instead.  <i>EngineVersion (page 754)</i>
<b>ProductVersion</b>	Use <b>EngineVersion</b> instead.  <i>EngineVersion (page 754)</i>
<b>QVUser</b>	
<b>Year2Date</b>	Use <b>YearToDate</b> instead.
<b>Vrank</b>	Use <b>Rank</b> instead.
<b>WildMatch5</b>	Use <b>WildMatch</b> 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.