Retail passion. Innovative solutions.

Retek® Predictive Application Server™ 11.1

Rules Functions Reference Guide

Corporate Headquarters:

Retek Inc.
Retek on the Mall
950 Nicollet Mall
Minneapolis, MN 55403
USA

888.61.RETEK (toll free US)
Switchboard:
+1 612 587 5000

Fax:
+1 612 587 5100

European Headquarters:

Retek
110 Wigmore Street
London
W1U 3RW
United Kingdom

Switchboard:
+44 (0)20 7563 4600

Sales Enquiries:
+44 (0)20 7563 46 46

Fax:
+44 (0)20 7563 46 10

Customer Support

Customer Support hours

Customer Support is available 7x24x365 via email, phone, and Web access.

Depending on the Support option chosen by a particular client (Standard, Plus, or Premium), the times that certain services are delivered may be restricted. Severity 1 (Critical) issues are addressed on a 7x24 basis and receive continuous attention until resolved, for all clients on active maintenance. Retek customers on active maintenance agreements may contact a global Customer Support representative in accordance with contract terms in one of the following ways.

Contact Method	Contact Information
E-mail	support@retek.com
Internet (ROCS)	rocs.retek.com Retek's secure client Web site to update and view issues
Phone	+1 612 587 5800

Toll free alternatives are also available in various regions of the world:

Australia	+1 800 555 923 (AU-Telstra) or +1 800 000 562 (AU-Optus)
France	0800 90 91 66
Hong Kong	800-96-4262
Korea	00-308-13-1341
United Kingdom	0800 917 2863
United States	+1 800 61 RETEK or 800 617 3835
Mail	Retek Customer Support Retek on the Mall 950 Nicollet Mall Minneapolis, MN 55403

When contacting Customer Support, please provide:

- Product version and program/module name.
- Functional and technical description of the problem (include business impact).
- Detailed step-by-step instructions to recreate.
- Exact error message received.
- Screen shots of each step you take.

Contents

Chapter 1 – Overview

Introduction

The purpose of this document is to provide the syntax and design of functions, procedures, and modifiers that are used in expressions in the RPAS 11 calculation engine.

The following terminology will be used throughout this document – note that there are important distinctions between each of these definitions:

- **Functions (single result)** – mechanisms for performing operations within an expression that are controlled and executed by the calculation engine.
  - Functions are most commonly used in RPAS
  - Functions can have one or more results depending on the type of function that was written; most functions in base RPAS return only a single measure
  - Calculation engine controls and executes the evaluation of a function
  - Functions may be used in expressions with other functions and keywords
- **Multiple-result functions** – similar to the features and behavior of single result functions but with semantic and syntactic differences
  - There can be more than one left-hand side (result) measure that can be specified implicitly by position in the expression or explicitly by label
  - Left-hand side measures have to be at same intersection; however, the calendar hierarchy can be dropped or added
  - The result(s) from a multi-result function cannot be used as arguments to another function, nor can the result(s) be chained with other operations to form long expression.
  - Expressions can be used as arguments to multi-result functions.
  - Multi-result functions cannot be part of a cycle group.
- **Procedures** – mechanisms for performing operations in an expression where the calculation engine controls the execution, which is performed by the procedure itself
  - Procedures can only use measures or scalars
  - Procedure executes the evaluation (instead of RPAS/calculation engine), but the calculation engine still controls protection processing, sequence of calculation, when the procedure is called, etc.
  - Can have multiple arguments on the left and right hand sides
  - Cannot be used with functions, other procedures, keywords, and certain modifiers
  - Because of their flexibility and the control that the developer has, procedures can be used for a wide variety of special calculations and activities
    - Require a different syntax: “<-“ instead of “=”

- **Modifiers** – used to directly modify the source or destination of measures, to override the level, aggregation type, position, etc. Syntax is <*measure*>.<*modifier*>
- **Key words** – used stand alone in expressions, or as arguments inside functions to return specific data values

Syntactical Approach

The agreed approach to the syntax is to provide a syntax that is as quick and straightforward to implement as possible. This implies simplicity of syntax, through a proliferation of similar functions, rather than a reduced number of functions with many parameters that are more complex to parse. Function names, keywords, etc., are currently in lowercase only.

Keywords will be allowed, but kept to a minimum. Function parameters will be comma separated, and may be optional, but will be positional, so that the absence of a parameter will need to be specified by commas if a subsequent parameter is supplied.

The following is the functional syntax used in this document:

- Large square brackets [ ] are used to indicate an optional parameter
- Small square brackets [ ] are part of the expression syntax, and are used to specify a hierarchy, dimension, and/or position
- Large braces { } indicate a choice where one of the items (which will be separated by a pipe sign |) must be selected
- Small braces {} are part of the expression syntax, and are used to specify a measure set for functions that accept a variable number of arguments (i.e. {<*measureset*>})
- Parameters of a specific type (such as expressions or dimension names) are shown in angle brackets <>
- A plus “+” sign is used to specify an intersection, which is done by connecting 2 or more dimension specifications
- Keywords, modifiers, function names, and procedure names are shown in **bold**

Note that a future extension to the syntax is anticipated for procedures to allow parameters to be passed by name.

Specification of Hierarchy, Dimension, or Position

Many functions in RPAS require the specification of a hierarchy, dimension, or a combination thereof, to define the level at which an expression is evaluated. When defining the hierarchy and dimension names in expressions square brackets [ ] must be used.

In the document the following syntax will be used to designate a hierarchy and dimension (note that position is noted as optional as it can only be specified in a limited number of functions):

[<*hierarchy*>].[<*dimension*>].[[<*position*>]]

For simplicity of parsing and clarity of rule writing, the <*hierarchy*> must be supplied in all cases, even when, as in calendar index functions, it might be implied from the context. Functions that require a hierarchy and dimension specification will have standard validation rules whereby [<*hierarchy*>] must be a valid hierarchy name and [<*dimension*>] must be a valid dimension in [<*hierarchy*>]; in some functions or procedures one of the hierarchical keywords **top**, **bottom**, or **current** (used conditionally based on context) can be used to specify the dimension. Should this validation fail, an ***error*** will be generated.

Function Inverses

Some functions (such as **cover**) have what are referred to as 'inverse' functions. This is required, as all expressions in a rule group must be algorithmic inverses of each other. Each function states whether it has an inverse, and, if so, what the syntax of the inverse is.

An inverse function is only relevant when the function encompasses the whole of the expression. Functions embedded in longer expressions do not have inverses, though the expression itself may have an inverse as long as the measure being 'solved' for is not an input into the function. Functions that have inverses usually have enough scope in their syntax to cover the eventualities that would typically cause them to be embedded in longer expressions (such as code to prevent an error result).

Functions with Multiple Results

The following special syntax should be used for functions with multiple results.

The left-hand side measures in a multi-result expression are comma-separated and can be identified by a labeling mechanism. The syntax for labels is:

<measure>:<label>

Valid label names are specified by the multi-result function syntax. If a multi-result function specifies valid labels, then the function can be used in an expression without specifying all possible results. The multi-result function itself is aware of which results are being stored and may be able to run faster by skipping the computation of unneeded results.

Chapter 2 – Special Handling for Functions

There are several keywords and functions that have special control flow over the evaluation of the expression.

Error Handling

RPAS has no facility for holding an 'error' value for a cell. Should the evaluation of any expression, or clause in an expression, result in an error, the value for the cell or clause will be the '***naval***'. Note, however, that it is good programming practice to check for any clauses that may return an error, and the **prefer** function provides a way to specify the behavior under these circumstances. Some functions have their own implicit error handling.

if

Used for handling conditional logic and masking updates within expressions.

Syntax: **if**(<*condition*>, <*use-expression*>, <*else-expression*>)

where <*condition*> is any valid Boolean expression. <*use-expression*> and <*else-expression*> are any valid expressions that are evaluated based on the result of <*condition*>; one (and only one) of these expressions can contain the keyword **ignore**. <*use-expression*> is evaluated when the result of <*condition*> is true; <*else-expression*> is evaluated when the result of <*condition*> is not true.

<*expression*> is any valid expression. **ignore** is a keyword used to indicate that the entire expression is not to be evaluated, i.e. masking the update to the entire expression. Note that **ignore** can ONLY be used in either the <*use-expression*> or <*else-expression*>, but not both.

The use of **ignore** always flags the expression as a masked update – this will always prevent the expression from being evaluated or involved with aggregations when the condition is not met. To reiterate, note that the entire expression is not evaluated, not just the sub-expression that uses the **if** clause. When **ignore** is used in expression where the LHS measure is modified with the **master** keyword (typically in a commit rule group), then the <*condition*> must be a Boolean measure (i.e. not an expression). This syntactical restriction is validated when the expression is parsed.

if clauses can be nested without restrictions but must be enclosed with parentheses when used more than once within an expression.

Examples:

- Conditional logic:
  - BOP = **if**(**current** == **first,** SeasOP, **lag**(EOP))
- OTB = **if**(ProjEOP > PlanEOP, 0, PlanRecpt – OnOrder)

- Masked update with a single expression:
  - SalesOP = **if**(Approved, SalesWP, **ignore**) Updates Sales for the Original Plan version to the value in the Working Plan version when the Boolean measure Approved is set to true. **ignore** designates that no update is made to SalesOP if the Approved measure is false. This is functionally equivalent to the next example.
  - SalesOP = **if**(NotApproved, **ignore**, SalesWP) Does not update the measure SalesOP with the values from the measure SalesWP when the Boolean measure NotApproved is true.
- Note the distinctly different behavior between the following similar expressions:
  - a = b + (**if**(<*condition*>, c, **ignore**)) Is an example of a masked update where no update is made to measure a if the condition is not met, i.e. the entire expression is not evaluated.
  - a = b + (**if**(<*condition*>,c, 0)) Is an example of conditional logic where an else clause is provided and the expression is *always* evaluated, thus "a" is always updated to either "b" or "b+c".

prefer

Returns the first non-error value from a series of expressions.

The primary use is to enable the capture and appropriate calculation of error conditions.

Syntax: **prefer**(<expression1>, <expression2> [, <expression3> … <expressionn>])

Where < *expression1-n*> are expressions which return values of the appropriate data type. The function returns the value of the first of the expressions that does not generate an ***error*** when it is evaluated. It is good coding practice to use a **prefer** function around any clause of an expression which could potentially generate an ***error***.

Inverse: The **prefer** function does not have an inverse.

Examples:

- **prefer**(A/B, 100) Returns the value of A divided by B, unless that generates an ***error*** (as it would if B is zero), when it returns 100.

prefer(lag(A), B) Returns the value the **lag** of A, unless that generates an ***error*** (as it would when evaluating the first period of the plan horizon), when it returns the value of B. The **prefer** function in this example is thus the functional equivalent of the expression: **if(current == first,** B, **lag**(A)).

Chapter 3 – Non-Conforming Measures

Definition

One of the strengths of the RPAS engine is that a workbook may contain measures with different ‘scopes’: the size and shape of the ‘multidimensional cube’ of data may vary by measure. Any two given measures in a workbook may have scopes that align exactly (e.g. both measures have a base intersection of SKU/Store/Week), or where one is a subset of the other (e.g. one has a base intersection of SKU/Store/Week and the other is at Class/Week). There can also be circumstances where each measure includes a hierarchy in its base intersection that the other dimension does not use (e.g. one has a base intersection of Class/Week and the other is Store/Week). In extreme circumstances, the scopes of two measures may have no point of overlap at all (e.g. one has a base intersection of Class and the other Store).

It is the scope of the measure on the left hand side of an expression (referred to as the LHS measure) that determines the cells that must be calculated by the expression, though that scope may be modified by the use of a modifier such as **level**. Where one or more measures on the right hand side of an expression (referred to as RHS measures) have a scope that is different (in any way) to the LHS measure, the expression is deemed to be ‘non-conforming’. There is special logic to handle the calculation of non-conforming expressions, which depends on the type of nonconformity.

Although not explicitly declared, there is a single logical ‘All’ position at the top of every hierarchy. When considering non-conformity, any measure that is not dimensioned on a hierarchy, is implicitly assumed to be dimensioned on the ‘All’ level of that hierarchy, and thus all data values are assumed to be for the ‘All’ position. This concept is key to the understanding of the handling of non-conforming expressions.

When the concept of the ‘All’ position is understood, all expressions can be considered to contain measures that use exactly the same hierarchies, the only potential differences between them are the ‘bottom levels’ (dimensions in the base intersection). Thus for handling non-conformity, only three cases need to be considered, for each hierarchy:

1 RHS same
In this case the RHS measure has the same bottom level as the LHS measure. The RHS measure is ‘conforming’ for that hierarchy, and values for the RHS measure are taken from the same position as the position being calculated for the LHS measure.

2 RHS higher
In this case the RHS measure has a higher bottom level than the LHS measure. The RHS measure is ‘non-conforming’ for that hierarchy. The values for the RHS measure for the position being calculated are assumed to be the same as the value of the RHS measure for the position in its bottom dimension that is the parent (ancestor) of the position being calculated. Effectively, it can be considered that the value of the measure has been ‘replicated’ down the hierarchy to the required level.

3 RHS lower
In this case the RHS measure has a lower bottom level than the LHS measure. The RHS measure is ‘non-conforming’ for that hierarchy, but because the scope of the RHS measure includes the bottom level for the LHS measure, values for the RHS measure are taken from the same position as the position being calculated for the LHS measure (RHS measure is aggregated using the default aggregation method).

The conceptual case where the measures have scopes that do not overlap, because they have base intersections in a hierarchy that are for dimensions that are up different ‘branches’ of the hierarchy, fails rule validation.

Examples

 **Note:** these examples all use the simple expression a = b + c.

Example 1

- a has a base intersection of SKU/Store/Week
- b has a base intersection of SKU/Week
- c has a base intersection of SKU/Region/Week

For each SKU/Store/Week, a is calculated from the value of b at SKU/Week (i.e. it is assumed that the value of b is the same for all positions in the location hierarchy) and the value of c at SKU/Region/Week, for the Region the Store belongs in. If ‘replication’ from the Region level is not appropriate, the rule writer can simulate other ‘spreading’ techniques by the use of functions and modifiers such as **count** and **level**.

For example, the **count** function may be used to determine the number of Stores in the Region, and so dividing the measure c by that count will simulate ‘even’ spreading. Additionally **level** could be used to force the calculation of a at Region instead of a’s base intersection, Store (a.level([loc].[reg])=b+c); in this scenario edits to b or c would calculate a at Region and would then spread those values down to Store for measure a using the default spread method.

Example 2

- a has a base intersection of SKU/Store/Week
- b has a base intersection of SKU/Week
- c has a base intersection of SKU

For each SKU/Store/Week, a is calculated from the value of b at SKU/Week (i.e. it is assumed that the value of b is the same for all positions in the location hierarchy) and the value of c at SKU (i.e. it is assumed that the value of b is the same for all positions in the location hierarchy and time hierarchy). Note that an alternative approach, if required, would be to use a **level** modifier on the measure a, so that it is calculated at, say, SKU/Week, and then spread down to SKU/Store/Week, using the existing store participations to the measure a.

Example 3

- a has a base intersection of SKU/Week
- b has a base intersection of SKU/Store/Week
- c has a base intersection of SKU/Region/Week

For each SKU/Week combination, a is calculated from the value of b and c at SKU/’All’/Week. Otherwise stated b and c are aggregated up the location hierarchy, then added to a for each position in SKU/Week.

Chapter 4 – Functional Keywords

Functional Keywords are keywords that may be used in expressions that return specific data values. There are a group of keywords that provide information (in the form of index numbers) about the calendar hierarchy, and a further group of keywords that provide information of the current session.

Calendar Index Functional Keywords

Certain calendar index functional keywords are supported in the syntax, as described below. In this context, a calendar index number is an ordinal position counter of the position in a dimension within the scope of the calendar horizon, where the dimension is as for the cell being evaluated. For example, in a plan whose scope is a year, the first week will have an index of 0, week 26 will have an index of 25, and week 52 will have an index of 51. Similarly, if an expression is being evaluated at the quarter level, the first quarter will have an index of 0, and the last one an index of 3. Calendar index functional keywords may be included in any numeric expression.

first

Returns the index number of the first calendar position.

This keyword is provided for completeness and clarity of rule function writing, since the value will always be zero!

last

Returns the index number of the last calendar position.

last + 1 will therefore always be the number of positions in the calendar horizon in the current dimension.

current

Returns the index number of the period being evaluated.

current can be used as a standalone keyword only under the context of time. Note that it can also be used in the syntax of a function as a hierarchical keyword (for specifying the current level in a hierarchy) and is allowed for any hierarchy (but must follow the syntax <hierarchy>.**current)**.

today

Returns the index number of the period that contains the current time as given by the system clock.

Note that the effect of this keyword may be overridden by providing the environment variable RPAS_TODAY. If this is present, the time in the RPAS_TODAY environment variable is used instead of the system clock time. Note the difference between the keywords **today** and **now** in that **today** returns an index number; **now** returns the value of the current date and time. An **error** is generated when the current period is not included in the workbook.

elapsed

Returns the index number of the period that is the last elapsed period

elapsed is interpreted as the last period for which actuals have been posted. If there is no elapsed period, this keyword returns –1.

elapsed must be assigned (a measure on the left hand side of an expression evaluated in a load rule group) before it can be used in calculations (on the right hand side of other rule groups). Use the following syntax for assigning the index number in the base calendar dimension as the elapsed value.

Syntax: **elapsed** = <*expression*>

Where <*expression*> is any valid expression that returns a numeric value, of which only the integer portion is used. **elapsed** can only be used on the left hand side of an expression in a load rule group; the **elapsed** key word can be used on the right hand side of other rule groups only after the elapsed value has been assigned.

Session Keywords

now

Returns the current date and time from the system clock.

now is stored with date and time information. Note the difference between the keywords **today** and **now** in that **today** returns an index number; **now** returns the value of the current date and time.

The displayed format of **now** is based on the measure type; note, however, that the RPAS 11.0 client cannot display time. This keyword is used to hold information about when data was changed (e.g. the beginning date and time of a batch run). The value returned by **now** can be overridden by RPAS_TODAY environment variable.

userid

Returns a string that contains the id of the current user.

This keyword is used to hold information about the user who made a specific change.

Calendar Hierarchical Date Keywords

begin

Returns a date type value for the first index in the root calendar dimension.

Because it returns a date type value, this keyword is not context sensitive (meaning it does not depend on where it is being used) and can be compared with the **now** keyword.

Note that the root calendar dimension is defined as the unique dimension that is at the root of the calendar hierarchy.

end

Returns a date type value for the last index in the root calendar dimension.

Because it returns a date type value, this keyword is not context sensitive (meaning it does not depend on where it is being used) and can be compared with the **now** keyword.

Chapter 5 – Modifiers

Modifiers are used to directly modify the source or destination of measures. Modifiers must be used in conjunction with a measure in the following manner:

Syntax: <*measure*>[.<*modifier*>[.<*modifier*>]…]

The following modifiers can be used with measures in a variety of ways – note the acceptable uses for each modifier as there are restrictions regarding use on the left hand side and if they can be used in conjunction with other modifiers.

master

References the domain-version of a measure.

master is used as a modifier to a measure to reference the version of the measure that resides in the domain. It can only be used in load and commit rule groups; it cannot be used in calculation rule groups.

Syntax: <*measure*>.**master**

Where <*measure*> is any valid measure. **master** can be used on both the left hand side and right hand side of expressions and can be used with functions. When used with other modifiers **master** must be the first modifier.

On the right hand side of an expression **master** can be used with both **level** and **aggtype**; on the left hand side **master** must be used by itself.

Examples:

- Sales=Sales.**master**
  Used in load rule group to retrieve Sales from the domain into a workbook
- Sales**.master**=Sales
  Used in commit rule group to commit the updated Sales measure to the domain from the version in the workbook

aggtype

References to alternative aggregation types.

When a measure is referenced just by name in an expression, or as a parameter in a rule function, the value used is for the default aggregation type for the measure. Values from alternative aggregation types are also available by using the syntax:

Syntax: <*measure*>.<***aggtype***>

Where <***aggtype***> is a supported aggregation type as listed in an appendix of this document. Every function parameter that requires a measure will also accept this extended form. Note that if alternate aggregation types are required for a measure in rules, this approach is more efficient that defining another measure with the alternate aggregation type, as data values at the base intersection are not duplicated.

The ***aggtype*** modifier can only be used on the right hand side of an expression but can be used with functions and other modifiers. When used with **level** and/or **master** modifiers, **aggtype** must be the last specified.

level

Returns the value of an expression for a specific intersection of 'parent' positions, or forces the calculation at a specific intersection.

The parents specified may be in one or more hierarchies.

Syntax: <*measure*>.***level***(<*dimspec1*>[+<*dimspec2*>… +<*dimspecn*>])

Where <*dimspec1-n*> is [<*hierarchy*>].{[<*dimension*>] | **top** | **current**} and each dimension specification is separated by a plus "+" sign.

<*measure*> is the measure to be specified. <*hierarchy*> is the name of a valid hierarchy. **top** and **current** are keywords referring to the highest, and current (i.e. being evaluated if on the RHS, or base intersection in the hierarchy if on the LHS) dimensions in the hierarchy. If a hierarchy is not specified, the <*dimension*> for that hierarchy is assumed to be **current**. If the <*dimension*> for a hierarchy is lower than the base intersection for the measure (when used on the LHS), or the <*dimension*> is not a valid dimension in the specified hierarchy, an ***error*** is generated.

This modifier can be used on both the LHS and RHS of a rule expression. It can only be used by itself on the LHS, but can be combined with other functions and modifiers on the RHS.

When this modifier is on the LHS of a rule expression the rule is evaluated at the specified intersection. The newly calculated value at an aggregated intersection is then spread down the hierarchies to the base intersection for the measure, using the default spread-type for the measure. A typical usage of this modifier on the LHS of a rule expression is to calculate a 'non-conforming' measure, where the scope of the measure includes hierarchies not present in the measures on the RHS of the expression. The calculation would usually be at the base intersection of the common hierarchies, but at the 'top' of the additional hierarchies, and spread to their base intersections.

When this modifier is on the RHS of a rule expression the measure being modified is evaluated at the specified intersection (note that just the measure, not the rule, is evaluated at the designated level). Under normal circumstances a measure is always calculated at the base intersection, or the intersection at which the LHS is being evaluated if it is higher. Use of the modifier will evaluate the measure at the designated (higher) level using the measure's default aggregation type, which can be overridden by the aggtype modifier. An example of its use on the RHS could be calculating a ratio of sales for each SKU with respect to its parent department.

Examples:

- sales**.level**([loc].**top**)
  Returns the value for the measure sales for the position at the top of the location hierarchy and for the current position in all other hierarchies.
- sales.**level**([loc].[area])
  Returns the value for the measure sales for the position in the area dimension that is the parent of the position being evaluated and the current position in all other hierarchies, i.e. 'the total sales in my area'.

- sales.**level**([loc].[area]+[prod].[div])
  Returns the value for the measure sales for the position in the area and division dimensions that is the parent of the position being evaluated and current position in all other hierarchies, i.e. 'the total sales in my area for my division'.
- recpts.**level**([rec].**top**) = <expression>
  The measure "recpts" is calculated at the base intersection of all hierarchies except the "rec" hierarchy, where it is calculated at the top. This value is then spread down to the base intersection for the measure.

old

References the value of a measure as of the previous calculate.

Syntax: <*measure*>.***old***

Any measure modified with **old** will use the value that was available at the start of the calculation process, which means that these modified measures can be ignored for such things as protection processing. Most importantly, this means that a measure can effectively be calculated from itself, as the .**old** modifier breaks the cycle.

Assumptions/restrictions:

- Can only be used in a rule group of type "calculation"
- Can only be used on the right-hand side of an expression
- Cannot be used in combination with .**master**, **level**, or .**aggtype** modifiers
- Cannot be used with (i.e. cannot modify) non-materialized measures

Use of the **old** modifier has no effect on calculation sequence or protection processing, as the values of measures modified with **old** are known before the calculation starts.

Example:

The **old** modifier can be used in conjunction with the **propspread** function to implement a hierarchical relationship among measures. In the following example, Total sales (TotalSls) is the "parent" measure and regular sales (RegSls), promotional sales (PromSls), and markdown sales (MkdSales) are the "child" measures. Using **old** and **propspread** to configure this relationship allows the manipulation of any combination of these measures before calculating, except all of them.

In the following example and in other such hierarchical measure relationships the order of the expressions within a rule is critical for the measures to be correctly calculated.

TotalSls = RegSls + PromoSls + MkdSls

RegSls, PromoSls, MkdSls = **propspread**(TotalSls, RegSls**.old**, PromoSls**.old**, MkdSls**.old**)

PromoSls, MkdSls = **propspread**(TotalSls - RegSls, PromoSls**.old**, MkdSls**.old**)

RegSls, MkdSls = **propspread**(TotalSls - PromoSls, RegSls**.old**, MkdSls**.old**)

RegSls, PromoSls = **propspread**(TotalSls - MkdSls, RegSls**.old**, PromoSls**.old**)

RegSls = TotalSls - PromoSls - MkdSls

PromoSls = TotalSls - RegSls - MkdSls

MkdSls = TotalSls - RegSls - PromoSls

Chapter 6 – Description of Functions

Calendar Index Functions

These are functions that return the calendar index numbers of positions that are specified relative to the current position through hierarchical relationships, or by date. Support is in place for functions to find the first and last children of a parent at a given dimension, e.g. the first week of the current quarter, the last week of the current month. These are to support relative time series functions, such as month to date totals. These may be constrained by setting a condition under which the expression is evaluated.

indexfirst

Returns the calendar index number of the first position in the current dimension that is descended from the parent of the current position at the specified dimension.

See the **tssum** function for an example of typical usage. The function may be constrained by setting a condition for the evaluation.

Syntax: **indexfirst**([<clndhierarchy>].{[<dimension>] | **top**}[, <boolexpr>])

Where <*clndhierarchy*> is the name of the calendar (time) hierarchy. <*dimension*> is the name of a dimension in the calendar hierarchy. **top** is a keyword that implies the top dimension in the calendar hierarchy. If <*dimension*> is not a valid dimension in the calendar hierarchy, or is not a dimension that is equal to or higher than the current (i.e. being evaluated) dimension (in any alternate hierarchy), an ***error*** is generated.

<*boolexpr*> is optional and is any valid Boolean expression used to set a condition for the evaluation of the function. If <*boolexpr*> is not specified the function returns the index number of the first position of the dimension descended from the parent of the current position of the specified dimension. When <*boolexpr*> is specified the function returns the index number of the first position of the dimension descended from the parent of the current position at the specified dimension where the <*boolexpr*> evaluates to true.

Inverse: The **indexfirst** function does not have an inverse.

Examples:

- **indexfirst**([clnd].[qtr])
  If, for example, the cell being evaluated is a week, this returns the calendar index number of the first week in the quarter that the week of the cell being evaluated belongs to, i.e. ‘the first week in the current quarter’.
- **indexfirst**([clnd].[week], Receipts != 0)
  If, for example, the cell being evaluated is a day, this returns the calendar index number of the first day of the current week when that has a value for Receipts that is not equal zero; i.e. ‘the first day in the current week with recorded Receipts’.
- **indexfirst**([clnd].**top**)
  If, for example, the cell being evaluated is a week, this returns the calendar index number of the first week in the calendar horizon. This keyword is included for consistency with other functions, as it will always return the value **first** (i.e. zero).

indexlast

Returns the calendar index number of the last position in the current dimension that is descended from the parent of the current position at the specified dimension.

The function may be constrained by setting a condition for the evaluation.

Syntax: **indexlast**([<clndhierarchy>].{[<dimension>] | **top**}[, <boolexpr>])

Where <*clndhierarchy*> is the name of the calendar (time) hierarchy. <*dimension*> is the name of a dimension in the calendar hierarchy. **top** is a keyword that implies the top dimension in the calendar hierarchy. If <*dimension*> is not a valid dimension in the calendar hierarchy, or is not a dimension that is equal to or higher than the current (i.e. being evaluated) dimension (in any alternate hierarchy), an ***error*** is generated.

<*boolexpr*> is optional and is any valid Boolean expression used to set a condition for the evaluation of the function. If <*boolexpr*> is not specified the function returns the index number of the last position of the dimension descended from the parent of the current position at the specified dimension. When <*boolexpr*> is specified the function returns the index number of the last position of the dimension descended from the parent of the current position at the specified dimension where the <*boolexpr*> evaluates to true.

Inverse: The **indexlast** function does not have an inverse.

Examples:

- **indexlast**([clnd].[qtr])
  If, for example, the cell being evaluated is a week, this returns the calendar index number of the last week in the quarter that the week for the cell being evaluated belongs to, i.e. ‘the last week in the current quarter’.
- **indexlast**([clnd].[week], Receipts != 0)
  If, for example, the cell being evaluated is a day, this returns the calendar index number of the last day of the current week that has a value for Receipts that is not equal to zero; i.e. ‘the last day of the current week with recorded Receipts’.
- **indexlast**([clnd].**top**)
  If, for example, the cell being evaluated is a week, this returns the calendar index number of the last week in the calendar horizon. This keyword is included for consistency with other functions, as it will always return the value **last**.

indextostartdate

Returns the start date of the period whose index number is supplied.

Syntax: **indextostartdate**(<index>[,[<clndhierarchy>].{[<dimension>] | **current**}])

Where <*clndhierarchy*> is the name of the calendar (time) hierarchy, and <*dimension*> is the name of a dimension in the calendar hierarchy. **current** is a keyword that implies the current dimension in the calendar hierarchy. If <*dimension*> is not a valid dimension in the calendar hierarchy, an ***error*** is generated. If the calendar hierarchy and dimension are not supplied, the default is the current calendar dimension.

Note this function requires that the day dimension of the calendar hierarchy be included in the workbook. If the lowest dimension of the calendar hierarchy is above the day dimension the function will not be able to return a valid date.

<*index*> is an expression that returns an index number in the indicated calendar dimension. If <*index*> is non-integer, only the integer portion is used. If <*index*> is not a valid index number for the specified dimension, an ***error*** is generated. If the measure being evaluated does not have a base intersection in the calendar hierarchy, and the **current** option is used, an ***error*** is generated.

The function returns a date that is the start date of the period indicated by the dimension and index number. If the period being evaluated is at or below the day level, the start date is the date of the whole of the period. If the period being evaluated is above the day level, the start date is the date of the first child position at the day level of the period being evaluated.

Inverse: The ***indextostartdate*** function does not have an inverse.

Examples:

- **indextostartdate**(**current**)
  Returns the start date of the current time period
- **indextostartdate** (**indexfirst**([clnd].[qtr]))
  Returns the start date of the first period in the current time dimension in the current quarter.
- **indextostartdate** (**index**([clnd].[week], openweek), [clnd].[week])
  Returns the start date of the period at the week level whose name is held in the openweek measure.

Index and Position Functions

This is a class of general functions that maybe used for any hierarchy that enables reference to positions in a generic manner. In most cases, the functions do not generate results that are useful in themselves, but that are used as parameters that are embedded into other functions.

An 'index' is an internal reference to a position in a dimension. For dimensions in the calendar hierarchy, the index reflects an ordering of positions, since there is a well-defined sequence (oldest to newest, based on the start and end dates) of periods. There are special calendar index functions (see above) that exploit this property. For other dimensions there is no such ordering, and the index number can be considered to be 'random'. Note that index numbers (including calendar index numbers) should not be saved and reused between planning sessions, as there is no guarantee that the same index numbers will apply in subsequent sessions, as the positions or relationships in a hierarchy may change.

These general index functions may be used for any hierarchy, including the calendar hierarchy.

index

Returns the index number of the specified position in the specified dimension of the specified hierarchy.

Syntax: **index**([<hierarchy>].{[<dimension>] | **current**}[,{ <stringexpr> | <dateexpr>}])

Where <*hierarchy*> is the name of a valid hierarchy, and <*dimension*> is the name of a valid dimension in that hierarchy. **current** is a keyword that returns the current dimension in <*hierarchy*>. If <*hierarchy*> is not a valid hierarchy or <*dimension*> is not a valid dimension in that hierarchy, an ***error*** is generated.

<*stringexpr*> and <*dateexpr*> are optional expressions that can be used to specify a position. If neither <*stringexpr*> nor <*dateexpr*> are specified the function returns the index number of the current position of the dimension being evaluated. <*stringexpr*> is a string expression that results in a position name. If the result of <*stringexpr*> is not a valid position name in the dimension being evaluated an ***error*** is generated. <*dateexpr*> is a numeric expression that results in a date type value and can only be used if <*hierarchy*> is the calendar hierarchy. If the result of <*dateexpr*> is not a date type value, or the result is returned when evaluating a dimension that is not in the calendar hierarchy an ***error*** is generated.

The function returns the index number of the indicated position in the specified dimension of the specified hierarchy. When used with dates the indicated position is the position that contains the date specified.

Inverse: The **index** function does not have an inverse.

Examples:

- **index**([prod].[item], likeitem)
  This returns the index number of the string position in the item dimension referenced in the likeitem measure.
- **index**([prod].[cls], "cls123")
  This returns the index number of the class cls123.
- **index**([clnd].[mnth], opendate)
  This returns the index number of the month that contains the date that results from the opendate measure.

position

Returns the position name of the position in the specified dimension of the specified hierarchy with the supplied index number.

Syntax: **position**([<hierarchy>].{[<dimension>] | **current**}[, <indexexpression>])

<*hierarchy*> must be the name of a valid hierarchy. If specified, <*dimension*> must be the name of a valid dimension in that hierarchy. **current** is a keyword that returns the current dimension in <*hierarchy*>. If <*hierarchy*> is not a valid hierarchy or <*dimension*> is not a valid dimension in that hierarchy, an ***error*** is generated.

<*indexexpression*> is an optional parameter to specify the index of the position to be evaluated. If <*indexexpression*> is not specified the current position is assumed. The expression must be a valid expression that results in a numeric measure. The integers of the resulting values of the expression are used as the index numbers to determine the position to be evaluated. If <*indexexpression*> does not return a valid index number for the specified dimension an ***error*** is generated.

The function returns a string that is the position name of the position with the specified index number for the specified dimension of the specified hierarchy.

Inverse: The **position** function does not have an inverse.

Examples:

- **position**([prod].[item], 3)
  This returns the position name of the item with index number =3.
- **position**([prod].[item], likeindex)
  This returns the position name of the item with the index number in the measure likeindex.
- **position**([prod].current)
  This returns the position name of the current position of the current dimension in the product hierarchy.

attribute

Returns the value of the specified attribute for the current position, or the position with the supplied index number.

Syntax: **attribute**(<attribute>, [<hierarchy>].{[<dimension>] | **current**}[, <indexexpression>])

Where <*attribute*> is a valid attribute for the dimension to be used, otherwise an ***error*** is generated. <*hierarchy*> must be the name of a valid hierarchy. If specified, <*dimension*> must be the name of a valid dimension in that hierarchy. **current** is a keyword that returns the current dimension in <*hierarchy*>. If <*hierarchy*> is not a valid hierarchy or <*dimension*> is not a valid dimension in that hierarchy, an ***error*** is generated.

<*indexexpression*> is an optional parameter to specify the index of the position to be evaluated. If <*indexexpression*> is not specified the current position is assumed. The expression must be a valid expression that results in a numeric measure. The integers of the resulting values of the expression are used as the index numbers to determine the position to be evaluated. If <*expression*> does not return a valid index number for the specified dimension an ***error*** is generated.

Valid values for <*attribute*> for all non-measure dimensions include:

- "label" – the label (description) for the position; this value must be specified using quotes

The function returns the value of the specified attribute for the specified position.

Inverse: The **attribute** function does not have an inverse.

Examples:

- **attribute**(label, [prod].current)
  This returns the value of the label attribute for the current position of the current dimension in the product hierarchy.
- **attribute**(label, [prod].[item], likeindex)
  This returns the value of the label attribute for the item with the index number in the measure likeindex, i.e. 'the label for my like item'.

Time Series Functions

This is a collection of very similar functions to perform typical calculation tasks over a range of cells in one or more time series. A <start> and an <end> position, defined using calendar index numbers, specifies the range of cells to be used. Typically, there may be some arithmetic performed to calculate the start and/or end positions.

Note that by using the **indexdate** or **index** functions to provide calendar index numbers, the <start> and <end> positions to be used in the time series can effectively be specified by position name or by date. Further note that if the level modifier is used, the **current** keyword only has a value when the level used is higher than the level being evaluated (since, for example, the concept of "the current week" is ambiguous when evaluating a month, so an ***error*** is generated).

Single Time Series functions:

tssum

Produces a sum of the cells in the time series for the measure defined by the start and end positions.

Is used for such calculations as 'season to date', 'balance to achieve', '4 week moving sum', and so on. The function produces a sum of the cells in the time series for the positions implied by the <*start*> and <*end*> for the specified dimension.

Syntax: **tssum**(<*expression*>[, <*start*>[, <*end*>]])

Where <*expression*> is an expression or measure whose time series is to be used, and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*expression*> is mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.**

Use the **level** modifier to specify a dimension in the calendar hierarchy when calculating above or below the current calendar dimension.

Inverse: The **tssum** function does not have an inverse.

Examples:

- **tssum**(PlanSales)
  This is a plan-to-date or running total value for sales.
- **tssum**(PlanSales, **current**, **last**)
  This provides a 'balance to achieve' (i.e. a sum from the current period to the end of the horizon).
- **tssum**(PlanSales.level([clnd].[week]), **current** – 3, **current**)
  This provides a 4 week moving total for sales.
- **tssum**(PlanSales, **indexfirst**([clnd].[qtr]))
  This provides a 'quarter to date' running total (see the **indexfirst** function).

tsavg

The average (mean) value of the cells in the range.

The function produces an average of the cells in the time series for the positions implied by the <*start*> and <*end*> for the specified dimension.

Syntax: **tsavg**(<*expression*>[, <*start*>[, <*end*>]])

Where <*expression*> is an expression or measure whose time series is to be used, and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*expression*> is mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.**

Use the **level** modifier to specify a dimension in the calendar hierarchy when calculating above or below the current calendar dimension.

Inverse: The **tsavg** function does not have an inverse.

tsmax

The maximum value of any cell in the range.

The function returns the maximum value of the cells in the time series for the positions implied by the <*start*> and <*end*> for the specified dimension.

Syntax: **tsmax**(<*expression*>[, <*start*>[, <*end*>]])

Where <*expression*> is an expression or measure whose time series is to be used, and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*expression*> is mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.**

Use the **level** modifier to specify a dimension in the calendar hierarchy when calculating above or below the current calendar dimension.

Inverse: The **tsmax** function does not have an inverse.

tsmin

The minimum value of any cell in the range.

The function returns the minimum value of the cells in the time series for the positions implied by the <*start*> and <*end*> for the specified dimension.

Syntax: **tsmin**(<*expression*>[, <*start*>[, <*end*>]])

Where <*expression*> is an expression or measure whose time series is to be used, and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*expression*> is mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.**

Use the **level** modifier to specify a dimension in the calendar hierarchy when calculating above or below the current calendar dimension.

Inverse: The **tsmin** function does not have an inverse.

tsmode

The modal value of the cells in the range.

The function returns the modal value of the cells in the time series for the positions implied by the <*start*> and <*end*> for the specified dimension.

Syntax: **tsmode**(<*expression*>[, <*start*>[, <*end*>]])

Where <*expression*> is an expression or measure whose time series is to be used, and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*expression*> is mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.**

If there is more than one value for the mode the function returns the first value that is calculated.

Use the **level** modifier to specify a dimension in the calendar hierarchy when calculating above or below the current calendar dimension.

Inverse: The **tsmode** function does not have an inverse.

tsmedian

The median value of the cells in the range.

The function returns the median value of the cells in the time series for the positions implied by the <*start*> and <*end*> for the specified dimension.

Syntax: **tsmedian**(<*expression*>[, <*start*>[, <*end*>]])

Where <*expression*> is an expression or measure whose time series is to be used, and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*expression*> is mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.**

If there is no middle number the function returns the average of the middle 2 numbers.

Use the **level** modifier to specify a dimension in the calendar hierarchy when calculating above or below the current calendar dimension.

Inverse: The **tsmedian** function does not have an inverse.

tsstd

The standard deviation of the cells in the range.

The function returns the standard deviation of the cells in the time series for the positions implied by the <*start*> and <*end*> for the specified dimension.

Syntax: **tsstd**(<*expression*>[, <*start*>[, <*end*>]])

Where <*expression*> is an expression or measure whose time series is to be used, and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*expression*> is mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.**

Use the **level** modifier to specify a dimension in the calendar hierarchy when calculating above or below the current calendar dimension.

Inverse: The **tsstd** function does not have an inverse.

tsvar

The variance of the cells in the range.

The function returns the variance of the cells in the time series for the positions implied by the <*start*> and <*end*> for the specified dimension.

Syntax: **tsvar**(<*expression*>[, <*start*>[, <*end*>]])

Where <*expression*> is an expression or measure whose time series is to be used, and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*expression*> is mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.**

Use the **level** modifier to specify a dimension in the calendar hierarchy when calculating above or below the current calendar dimension.

Inverse: The **tsvar** function does not have an inverse.

Double Time Series (statistical error) functions:

tsme

Produces the Mean Error of an 'estimate' time series compared to an 'actuals' time series.

Syntax: **tsme**(<*x*>, <*y*>[, <*start*>[, <*end*>]])

Where <*x*> is an expression or measure that represents the 'estimate' and <*y*> is an expression or measure that represents the 'actuals', and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*x*> and <*y*> are mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.** The Mean error is calculated using the formula:

$$\frac{\sum_{i=0}^{n-1}(x_i - y_i)}{n}$$

Inverse: The **tsme** function does not have an inverse.

Examples:

- tsme(FcstSales, ActSales)
  This calculates the Mean Error of the FcstSales measure from the start of the calendar horizon until the current time period.
- tsme(FcstSales, ActSales, first, elapsed)
  This calculates the Mean Error of the FcstSales measure from the start of the calendar horizon until the last time period with actuals loaded.
- tsme(FcstSales, ActSales, first, min(elapsed, current))
  This calculates the Mean Error of the FcstSales measure from the start of the calendar horizon until the first of the period being evaluated or the last time period with actuals loaded.

tsmae

Mean Absolute Error.

Syntax: **tsmae**(<*x*>, <*y*>[, <*start*>[, <*end*>]])

Where <*x*> is an expression or measure that represents the 'estimate' and <*y*> is an expression or measure that represents the 'actuals', and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*x*> and <*y*> are mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.** The Mean Absolute error is calculated using the formula:

$$\frac{\sum_{i=0}^{n-1} |x_i - y_i|}{n}$$

Inverse: The **tsmae** function does not have an inverse.

tsmape

Mean Absolute Percentage Error.

Syntax: **tsmape**(<*x*>, <*y*>[, <*start*>[, <*end*>]])

Where <*x*> is an expression or measure that represents the 'estimate' and <*y*> is an expression or measure that represents the 'actuals', and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*x*> and <*y*> are mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.** The Mean Absolute Percentage error is calculated using the formula:

$$\frac{\sum_{i\in\{0\le i<n \cap y_i\neq 0\}}\left|\frac{x_i - y_i}{y_i}\right|}{\sum_{i\in\{0\le i<n \cap y_i\neq 0\}}1}$$

Inverse: The **tsmape** function does not have an inverse.

tsrmse

Root Mean Square Error.

Syntax: **tsrmse**(<*x*>, <*y*>[, <*start*>[, <*end*>]])

Where <*x*> is an expression or measure that represents the 'estimate' and <*y*> is an expression or measure that represents the 'actuals', and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*x*> and <*y*> are mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.** The Root Mean Square error is calculated using the formula:

$$\sqrt{\frac{\sum_{i=0}^{n-1}(x_i - y_i)^2}{n}}$$

Inverse: The **tsrmse** function does not have an inverse.

tspae

Percentage Absolute Error.

Syntax: **tspae**(<*x*>, <*y*>[, <*start*>[, <*end*>]])

Where <*x*> is an expression or measure that represents the 'estimate' and <*y*> is an expression or measure that represents the 'actuals', and <*start*> and <*end*> are expressions that calculate numbers. The current calendar dimension is assumed, and if the cell being evaluated does not have a calendar dimension, the bottom calendar dimension is assumed. If the values of <*start*> or <*end*> are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the specified dimension, an ***error*** is generated.

<*x*> and <*y*> are mandatory, the other parameters are optional. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **current.** The Percentage Absolute error is calculated using the formula:

$$\frac{\sum_{i=0}^{n-1}\left|x_i - y_i\right|}{\sum_{i=0}^{n-1}\left|y_i\right|}$$

Inverse: The **tspae** function does not have an inverse.

Hierarchical Functions

This is a collection of functions and procedures that provide some knowledge of hierarchical structures, and the how the current position fits in, or uses knowledge of hierarchical structures.

Future additions to these functions are anticipated to support syntax where the 'parent' or 'child' levels may be specified by relative offset (e.g., 2 above, 1 below). These add a degree of complication as there may be alternative hierarchies, and therefore ambiguity as to which dimension is referred to (issue to be resolved at a later time).

count

Returns the count of children at a specified level that belong to a parent at a higher level.

Syntax: **count***([<hierarchy>][.<childdimspec>[,<parentdimspec>]])*

Where <*childdimspec*> is {[<*childdimension*>] | **bottom** | **current**}

and <*parentdimspec*> is [<*hierarchy*>].{[<*parentdimension*>] | **top** | **current**}

<*hierarchy*> is the name of a valid hierarchy (same hierarchy must be referenced throughout the function). <*childdimension*> and <*parentdimension*> must be valid dimensions in the specified hierarchy. If both are specified, then <*childdimension*> must be lower than <*parentdimension*> in a roll-up, or an ***error*** is generated. **bottom, top** and **current** are keywords referring to the lowest, highest, and current (i.e. being evaluated) dimensions in the hierarchy. If <*childdimspec*> is not specified, the default is **bottom**. If <*parentdimension*> is not specified, the default is **current**.

The function returns the number of children in the dimension <*childdimension*> that are descended from the implied position (i.e. the current position or the ancestor of the current position at the specified level) in the dimension <*parentdimension*>. If the <*childdimension*> and the <*parentdimension*> are the same, the function returns the value of 1.

Inverse: The **count** function does not have an inverse.

Examples:

- count([loc].bottom)
  Returns the number of children in the bottom dimension in the location hierarchy for the current position in the location hierarchy
- count([loc].[str])
  Returns the number of children in the store dimension (str) in the location hierarchy for the current position in the location hierarchy. i.e. 'how many stores do I own'
- count([loc].[str], [loc].[area])
  Returns the number of children in the store dimension in the location hierarchy for the position in the dimension "area" that is the ancestor of the current position in the location hierarchy, i.e., 'how many stores in my area'

lookup

Procedure that returns the value of an expression for a specific intersection.

The positions to be "looked up" may be in one or more hierarchies. This procedure has special uses and restrictions in RPAS 11.0:

- lookup is a procedure and thus cannot be combined with functions and other procedures in any manner
- Used for history mapping and like SKU/sister store functionality
- The base intersection of the output measure must be the same as the input measure and/or one or more of the mapping measures

Syntax: <output> <- **lookup**(<input>, <dimspec1> [, <dimspec2> … , <dimspecn>])

Where <*dimspec1-n*> is [<*hierarchy*>].{[<*dimension*>] | **bottom** | **current** | **top**}, <*map*>

<*output*> is the measure being updated. <*input*> is the measure to be evaluated. Each <*dimspec*> is used to specify the hierarchy and dimension to be used in the mapping process and the measure that contains the mapping values.

For each <*dimspec*> that is specified, the <*hierarchy*> must be the name of a valid hierarchy and the <*dimension*> must be the name of a valid dimension in that hierarchy. **top** is a keyword that refers to the highest dimension in the hierarchy, **bottom** is a keyword that refers to the lowest dimension in the hierarchy, and **current** is a keyword that refers to the current dimension, i.e. the dimension of the cell being evaluated.

<*map*> is either a measure or an explicitly stated position used to designate how positions in <*input*> are mapped to determine the resulting values in <*output*>. The output, input, and mapping measures used in the **lookup** procedure must conform in a certain manner. Specifically the resulting measure, <*output*>, must have the same base intersection as <*input*>, <*map*>, or both measures (so that all conform).

When <*map*> is a measure it must result in either index numbers or position names, either of which must be valid index numbers or position names from the related dimension specification. When <*map*> contains index numbers that do not map to valid positions an ***error*** is generated and the ***na value*** for <*output*> is returned. There is no special 'cycle breaking' logic for the **lookup** procedure. This means that a measure may never be calculated from the **lookup** of the same measure.

Note: **lookup** is a procedure and thus cannot be combined with functions, modifiers, or other procedures in any manner. As a procedure it requires a different syntax: "<-" instead of "=" when being assigned.

Inverse: The **lookup** procedure does not have an inverse.

Examples:

- output <- lookup(input, [presentationstyle].[presentationstyle], map)
  Where output is at sku-week, input is at sku-presentationstyle, and map is at sku-week with position names or index numbers from the presentation style dimension (the output and mapping measures have the same base intersection); this expression calculates the output measure from the mapping of presentation styles that vary for each sku-week combination.
- output <- lookup(input, [prod].[sku], map)
  Where output and input are at sku-week and map is at sku with position names or index numbers from the sku dimension; this expression calculates the output measure from the like sku of the input measure.

tablelookup

Procedure that returns the value (interpolated if necessary) from the entry in a 'table' of information held in measures that matches with supplied 'keys'.

Syntax: **tablelookup**(<expression>, <matching technique>, <keymeasure> [,<resultmeasure>])

Where <*matching technique*> is {exactmatch, <*nomatchvalue*> | average | nearest | high | low | interpolate}

The tablelookup procedure requires that a 'table' be available that may be the target of the lookup. This 'table' will be formed from normal measures with a base intersection of 'normal' dimensions. Nevertheless, the most usual usage will be where the 'table' measures are dimensioned on a dimension built for the purpose, plus other dimensions as required.

An example will make this clearer. Imagine a requirement to look up valid price points that may be applied as prices for an *item*. The collection of valid price points will be different for each *class*. To satisfy this requirement a table is built. A 'table' hierarchy is defined with a 'tableentry' dimension with a number of positions, which are named e01, e02, e03, … e99 to allow for 99 entries in the table, with the order of the positions being the same as their natural sort sequence, and the order of the hierarchy being the highest (innermost) non-time hierarchy. A measure named 'pp' is defined with a base intersection of tableentry/class. The 'pp' measure is populated with valid pricepoints for each class, with the lowest valid pricepoint in position e01, the next lowest in e02, etc. This 'table' can now be used to 'look up' valid pricepoints: the procedure call (indirectly) provides a class and (directly) provides a target price as arguments, and a valid price point is returned based on the selected matching technique. See the examples, below, for an example that uses this 'table'.

<*expression*> is any valid expression that results in a value of the same data type as the <*keymeasure*>. In the description that follows, this value is referred to as the key value. <*keymeasure*> is the name of the measure to be used as a key when matching the key value against the 'table'. <*resultmeasure*> is an optional measure that holds the return value. If <*resultmeasure*> is not specified <*keymeasure*> is used for the values of the result (as in the price point example, below).

The procedure will attempt to match the key value against an entry in the 'table'. The innermost non-time dimension in the base intersection of the <*keymeasure*> is assumed to be the dimension along which entries in the table are indexed. For all other dimensions in the base intersection of the <*keymeasure*>, the procedure will match against the parent at that dimension of the cell being evaluated.

Note that the values in <*keymeasure*> must be in ascending order and must not contain any repeated values. A value that is either out of sequence or repeated designates that the previous value is the last entry in the 'table'. In other words, only the sorted elements in the key measure will be considered in the lookup process.

The <*matching technique*> specifies the matching technique to be used when an exact match of the <*keymeasure*> against the key value is not found. If the matching technique is **exactmatch**, <*nomatchvalue*> is a numeric value that must be specified to indicate the value to use in cells when there is no exact match. Otherwise if the key value is higher than the highest value in the "table", or lower than the lowest value in the table, it is assumed to match against the highest or lowest value accordingly. If the matching technique is **high** and no match against the key value is found, the procedure returns the value of the <*resultmeasure*> for the entry immediately higher than the key value. If the matching technique is **low** and no match against the key value is found, the procedure returns the value of the <*resultmeasure*> for the entry immediately lower than the key value. If the matching technique is **nearest** and no match against the key value is found, the procedure returns the value of the <*resultmeasure*> for the entry immediately lower than the key value or immediately higher than the key value, depending upon which entry is nearest (i.e. this is like rounding to the nearest value). If the matching technique is **average** and no match against the key value is found, the procedure returns the numeric average of the value of the <*resultmeasure*> for the entry immediately lower and immediately higher than the key value, or it generates an error if the <*resultmeasure*> is not of numeric data type.

If the matching technique is **interpolate** and no match against the key value is found, the procedure returns an interpolated value between the value of the <*resultmeasure*> for the entry immediately lower and immediately higher than the key value, or it generates an error if the <*resultmeasure*> is not of numeric data type. The interpolation is calculated as:

$$lowresult + \frac{(highresult - lowresult) * (keyvalue - lowvalue)}{(highvalue - lowvalue)}$$

Note that it is anticipated that there may be future extensions to the syntax of the **tablelookup** procedure to support the specific setting of 'match keys' for the other dimensions that the <*keymeasure*> is dimensioned upon, rather than automatically matching on the parent in the dimension of the base intersection of the <*keymeasure*>.

*Note that **tablelookup** is a procedure in RPAS 11.0 and thus cannot be combined with functions, modifiers, or other procedures in any manner. As a procedure it requires a different syntax: "<-" instead of "=" when being assigned.*

Inverse: The **tablelookup** procedure does not have an inverse.

Examples:

- **tablelookup**(tgtpr, nearest, pp)
  Returns the nearest valid value of the pp measure to the supplied target price (tgtpr).
- **tablelookup**(perc, interpolate, epct, elast)
  Looks up the percentage markdown (perc) of the current position against a percentage change elasticity table (epct). Returns the matching elasticity value (elast). If the percentage markdown is not found in the table, the procedure will interpolate the elasticity value from the nearest values above and below the percentage markdown.

flookup

'Fixed lookup' function which returns the value of a measure for an explicitly named fixed intersection.

Syntax: **flookup**(<measure>, <posspec1> [, <posspec2> … , <posspecn>])

Where <*posspec1-n*> is: [<*hierarchy*>].[<*dimension*>].[<*positionname*>]

<*measure*> is the measure to be looked up. This <*measure*> must conform with the measure being calculated as follows. Some hierarchies may be present in the base intersection of both measures, and these are handled by normal 'non-conforming' logic. For any hierarchies that are only in the base intersection of the measure being calculated (output measure), all positions will lookup the same value. For any hierarchies that are only in the base intersection of the <*measure*> (input measure), the position to be used **must** be **explicitly** named through a position specification (<*posspec*>). Note that if the position to be used can only be specified indirectly (for example, if it is held in a measure), the **flookup** function cannot be used, and the more powerful **lookup** procedure should be used instead.

flookup can be used to return a constant or a slice. In case of a constant, the NA value of the **flookup** function will be the value of the constant. In case of a slice, the NA value of the **flookup** function will be the NA value of <*measure*>.

For each <*posspec*> that is specified, the <*hierarchy*> must be the name of a valid hierarchy, the <*dimension*> must be the name of a valid dimension in that hierarchy, and the <*positionname*> must be the name of a valid position in that dimension; if the position name includes special characters it can be enclosed in quotes (" ") in addition to the standard requirement for square brakets ([ ]). If <*hierarchy*> is not a valid hierarchy or <*dimension*> is not a valid dimension in that hierarchy, or <*positionname*> is not a valid position in that dimension an ***error*** is generated.

Additionally, <*dimension*> must be a dimension in the base intersection of <*measure*>. To use dimensions not in the base intersection, the <*measure*> must have a **level** modifier to explicitly raise it to the desired dimension.

There is no special 'cycle breaking' logic for the ***flookup*** function. This means that a measure may never be calculated from the ***flookup*** of the same measure. The ***flookup*** function returns the value of the expression from the specified fixed intersection.

Inverse: The ***flookup*** function does not have an inverse.

Examples:

- **flookup**(perc, [flvl].[flvl].[flvla])
  Returns the value for the measure perc for the position 'flvla' in the flvl dimension of the flvl hierarchy.
- **flookup**(leadtime, [prod].[cls].[class1], [loc].[whse].[whseA])
  Returns the value for the measure *leadtime* for the class 'class1' for the warehouse 'whseA'.

aggregate

Procedure that returns the value of a measure aggregated from the base intersection to the current level, using the supplied aggregation type.

Syntax: **aggregate** (<cachemeasure>, <hierspec1> [, <hierspec2> … , <hierspecn>])

Where <*hierspec1-n*> is [<*hierarchy*>].<*aggtype*>

The **aggregate** procedure provides a mechanism that is the functional equivalent of an aggregation type that varies by hierarchy. Such variable aggregation types cannot be implemented as aggregation types because of the difficulties of spreading changes. The aggregate procedure avoids that difficulty, because the measure being calculated will be of recalc type, and thus is spread indirectly through a mapping rule.

The rule writer specifies a <*cachemeasure*> which holds the base intersection values to be aggregated, and is also the source of values for 'recalc' aggregation.

The rule writer also specifies the aggregation type for each hierarchy and the priority sequence to be used. The priority sequence is required because at levels that are aggregated in more than one hierarchy (say, Department/Region/Month for a measure with a base intersection of Class/Store/Week), different results would usually be obtained by aggregating up each of the hierarchies. Thus, for example, if the requirement is to aggregate up the product hierarchy by using the total aggregation type, up the location hierarchy using the average aggregation type and the calendar hierarchy by using the first aggregation type, there are three potential ways to calculate a value at Department/Region/Month. We could total from Class/Region/Month, average from Department/Store/Month or first from Department/Region/Week. These would almost certainly generate three completely different values. By providing a priority sequence, the rule writer explicitly determines which of these values are required (See worked example, below).

Note that the effect of a series of aggregations of the same type up a single hierarchy may return different results from those of a measure with the same aggregation type. 'Normal' aggregation for a measure driven by its aggregation type is performed from all base intersection cells descended from the cell being evaluated. Thus, for example, for a measure with a base intersection of Class/Week and an 'average' aggregation type, the value calculated for a cell at Department/Month is the average of all values for all Class/Week cells for the Department/Month. If the measure is a recalc measure, calculated at aggregated levels from a rule with an aggregate function such as **aggregate**(x, [prod].average, [clnd].average), the value for the Department/Month will be the average of all the Class/Months (not Class/Weeks) that belong to the Department/Month. Other than coincidentally, this would generate a different value.

<*cachemeasure*> is the measure to be aggregated, and the value of <*cachemeasure*> is also the value used for cells that are at the base intersection (i.e. bottom levels), and at aggregated levels when the required aggregation type is *recalc*. <*hierarchy*> is the name of a valid hierarchy. Each hierarchy may only be specified once in the procedure, but hierarchies may appear in any order. The sequence that the hierarchies are specified in is used to determine which hierarchy to aggregate up if the cell being evaluated is at an aggregated level in more than one hierarchy. In this circumstance, aggregation is performed up the first specified hierarchy that the cell is at an aggregated level in, and the other hierarchies are ignored. <*aggtype*> specifies the aggregation type to be used. The <*aggtype*> must be one of the standard aggregation types (see Appendix in back). If any hierarchy that is in the scope of the measure being calculated is not explicitly specified, the aggregation type of that hierarchy is assumed to be total. Such hierarchies are assumed to be sequenced after all hierarchies that are explicitly referenced, and ordered from innermost to outermost.

Note that the value of the <*cachemeasure*> is used at the base intersection of the measure being calculated. If, for a given cell, the aggregation type to be used is recalc, the value is also obtained directly from the <*cachemeasure*> at that level, which will normally have an aggregation type of recalc.

*Note that **aggregate** is a procedure and thus cannot be combined with functions, modifiers, or other procedures in any manner. As a procedure it requires a different syntax: "<-" instead of "=" when being assigned.*

Inverse: The **aggregate** procedure does not have an inverse.

Examples:

- result <- **aggregate**(x, [clnd].recalc)
  For cells at the base intersection, the value is calculated from the measure x. For cells at an aggregated level in the calendar hierarchy, the value is also obtained from the measure x (which we can assume has an aggregation type of recalc, and thus the result of the procedure is as if the aggregation type were recalc, using the usual expression to calculate measure x). If the cell is not at an aggregated level in the time hierarchy, and assuming in this example that the other hierarchies are product and location, in that priority, the value for a cell at an aggregated product level is calculated as the total of all cells for products descended from that product, for the same location and time. Otherwise the value for the cell is calculated as the total of all cells for locations descended from the cell's location, for the same product and time.
- result <- **aggregate**(x, [loc].average)
  In a similar manner to the previous example, cells at aggregated levels in the location hierarchy will be calculated by averaging the values of cells for all descendent locations, otherwise the value will be totaled up the product or time hierarchy as appropriate.
- result <- **aggregate**(x, [clnd].average)
  Totals up all hierarchies except time, which uses an average aggregation type.
- result <- **aggregate**(x, [prod].average, [clnd].first)
  Averages up the product hierarchy if possible, otherwise takes the first child value up the calendar hierarchy, otherwise totals up the other hierarchies.
- result <- **aggregate**(x, [prod].average, [clnd].last)
  Averages up the product hierarchy if possible, otherwise takes the last child value up the calendar hierarchy, otherwise totals up the other hierarchies.

Multi-level Calculation Example:

- Consider a measure calculated from the expression **aggregate**(x, [prod].total, [loc].avg, [clnd].first). The measure is assumed to have a base intersection of Class/Store/Week. Examples of the calculations that would be applied at various levels are:
  - Class/Store/Month: first from Class/Store/Week
  - Class/Region/Month: avg from Class/Store/Month
  - Department/Region/Month: total from Class/Region/Month

Normalization & Resizing Functions

norm

Normalizes a time series: recalculates the values for individual time periods to a new total, whilst retaining the 'shape'.

Syntax: **norm**(<expression>[, <total>[, <start>[, <end>]]])

Where <*expression*> is a measure or expression whose time series is to be used, and <*start*> and <*end*> are expressions that calculate numbers. <*total*> is an expression that returns a numeric value. <*start*> or <*end*> are assumed to be calendar index numbers: if their values are numeric but non-integer, only the integer portion will be used. If <*end*> is less than <*start*>, or either parameter is non-numeric or outside the scope of the calendar index numbers for the dimension being calculated, an ***error*** is generated. <*expression*> is mandatory, the other parameters are optional. If <*total*> is not specified, the default is 1. If <*start*> is not specified, the default value is **first** (i.e. 0). If <*end*> is not specified, the default value is **last.**

The function returns a time series that is normalized such that the shape of the values for individual time periods is retained, but the sum of the individual time periods, for the horizon covered by the <*start*> and <*end*> is equal to the supplied <*total*>. For time periods outside the horizon covered by the <*start*> and <*end*> (if there are any) the function will return zero – if values other than this are required, or if no update to those periods is required, the function should be wrapped in an if function that can set the appropriate value or use the **ignore** clause, as appropriate.

See the **resizenorm** function for a function that normalizes a time series of different length to the target time series.

Inverse: The **norm** function does not have an inverse.

Examples:

- **norm**(profile, 1)
  This produces a normalized time series of the profile measure, with the sum of all time periods equal to 1.
- **norm**(profile, last+1)
  This produces a normalized time series of the profile measure, with the sum of all time periods equal to the number of periods, that is, the average value of the time periods is one. This form of normalized profile is typically used as a multiplier for seasonalizing or deseasonalizing a time series.

- **norm**(lag(profile,startweek), targetsales, startweek, startweek-1+numweeks)
  This example uses a profile to generate a sales plan for an item for a specified length of time, from a specified period of time. The measure *profile* is assumed to have a profile (shape) for the sales of an item (this measure would probably be dimensioned at a higher level in the product hierarchy than that being evaluated), starting in the first period with values for the required number of periods (*numweeks*). The measure *targetsales* is the total sales required for the item whose sales are being generated over the period of time. *startweek* is an index number of the period from which sales should be generated for the item. *numweeks* has the length of the sales profile to be generated. Periods before the *startweek* or after the *startweek-1+numweeks* will have a result of zero. The periods from *startweek* will have the result of the first *numweeks* of the *profile* measure, normalized to total *targetsales.*

resize

Uses the 'shape' of a time series to produce another time series of a different length, but with the same shape.

Syntax: **resize***(<expression>, <start>, <fromlength>, <tolength>)*

Where <*expression*> is a measure or expression whose time series is to be used, and <*start*>, <*fromlength*> and <*tolength*> are expressions that calculate numbers. <*start*> is assumed to be a calendar index number: if its value is numeric but non-integer, only the integer portion will be used. If <*fromlength*> or <*tolength*> are less than 0, or either parameter is non-numeric or when added to <*start*>-1 is outside the scope of the calendar index numbers for the dimension being calculated, an ***error*** is generated. If <*fromlength*> or <*tolength*> are non-integer, only the integer portion will be used.

The function returns a time series that is resized such that the overall shape of the values is retained, but the number of time periods is stretched or shrunk from <*fromlength*> or <*tolength*>. For time periods outside the horizon covered by <*start*> and <*start*> -1 + <*tolength*> (if there are any) the function will return zero – if values other than this are required, or if no update to those periods is required, the function should be wrapped in an if function that can set the appropriate value or use the **ignore** clause, as appropriate.

The function stretches or shrinks the section of the time series by interpolation or decimation. The algorithm uses upsampling, convolution and then downsizing. The filter used in convolution is a finite impulse response (FIR) lowpass filter, using a hamming window, with cut-off frequency and length determined from greatest common denominator of the source and destination time series lengths.

The values generated for individual cells through this process are not normalized (for a similar function that normalizes the result, see the **resizenorm** function), and will be of similar magnitude to the cell values for the source cells.

Inverse: The **resize** function does not have an inverse.

Examples:

- **resize**(profile, first, 10, 17)
  The first 17 periods of the result time series will have values with a shape the same as the first 10 periods of the measure *profile*. All other periods will be zero.
- **resize**(lag(profile,startweek), startweek, profilelength, numweeks)
  This example should be compared with the similar example of the normalize function. It uses a profile to generate a sales plan for an item for a specified length of time, from a specified period of time. The profile is not necessarily the same length as the period for which sales are to be generated. The measure *profile* is assumed to have a profile (shape) for the sales of an item, starting in the first period with values for a number of periods given by the measure *profilelength*. *startweek* is an index number of the period from which sales should be generated for the item. *numweeks* has the length of the sales profile to be generated. Periods before the *startweek* or after the *startweek-1+numweeks* will have a result of zero. The periods from *startweek* to *startweek-1+numweeks* will have the result of the first *profilelength* weeks of the *profile* measure, stretched or shrunk to fit the appropriate number of periods.

resizenorm

Uses the 'shape' of a time series to produce another time series of a different length, but with the same shape, normalized to a specific total.

Syntax: **resizenorm**(<expression>, <start>, <fromlength>, <tolength>[, <total>])

<*total*> is an expression that returns a numeric value. If <*total*> is not specified, it is assumed to be the sum of the cells of <*expression*> from *startweek* to *startweek-1+fromlength*. See the **resize** function for an explanation of the other parameters.

This function is identical to the **resize** function, except that the calculation engine automatically normalizes the resized values to the specified <*total*> by the calling the **norm** function on the resized values. This function is provided to ensure that the correct parameters are automatically passed to the **norm** function, and to be more efficient than wrapping the **resize** function in a **norm** function.

Inverse: The **resizenorm** function does not have an inverse.

Examples:

- **resizenorm**(profile, first, 10, 17)
  The first 17 periods of the result time series will have values with a shape the same as the first 10 periods of the measure *profile*. All other periods will be zero. The values of the cells will be such that sum of the 17 generated periods of the result time series will be the same as the first 10 periods of the measure *profile*.
- **resizenorm**(lag(profile,startweek), startweek, profilelength, numweeks, targetsales)
  This example should be compared with the similar example of the resize function. The generated sales will be normalized so that their sum is the value of the *targetsales* measure.

Other Functions

cover

Returns the number of future periods for which ‘stock’ covers ‘sales’.

Alternately phrased, that is a ‘forwards weeks of supply’, or the number of future periods of ‘sales’ that could be satisfied from the ‘stock’ with no further receipts

The **cover** function allows for two ‘sales’ expressions, where the second is a ‘wrap around’ expression to provide a well defined cover for periods at or near the end of the calendar horizon that would otherwise ‘run out’ of forward sales. An offset is also specified to allow the **cover** function to behave appropriately for both opening and closing stock.

Syntax: **cover**(<stockexpression>, <salesexpression>[, <offsetexpression>, [<wraparoundsalesexpression>]])

Where <*stockexpression*> is an expression or measure that represents the ‘stock’. <*salesexpression*> is an expression or measure that represents the ‘sales’. <*offsetexpression*> is an expression that calculates a number that represents the offset to apply. If the value is non-integer, only the integer portion is used. If the value is non-numeric, an ***error*** is generated. If <*offsetexpression*> is not provided, the default value will be 1. <*wraparoundsalesexpression*> is an expression or measure that represents the ‘wrap around sales’. If <*wraparoundsalesexpression*> is not provided, there will be no wraparound, and the function will generate an ***error*** if there is insufficient ‘forward sales’ to calculate the cover.

The <*salesexpression*> can be considered to define a time series of sales data values, starting at the current period offset by the <*offsetexpression*>, and stretching until the end of the calendar horizon. If this time series is too short to evaluate the cover value, it can be considered to be extended by one *or more* copies of the time series implied by the <*wraparoundsalesexpression*>, if specified, from the start until the end of the calendar horizon. The cover value is calculated by summing down the time series until a sum is reached which is equal to or greater than the value of the <*stockexpression*>. If the sum is equal to the <*stockexpression*>, the number of periods used is returned. If the sum is greater than the <*stockexpression*>, the value returned is the number of periods used minus 1, plus the proportion of the last period reached that is required to exactly reach the value of the <*stockexpression*>. If the <*offsetexpression*> causes the start of the time series to be before the start of the calendar horizon, or no <*wraparoundsalesexpression*>, is specified, and there is insufficient ‘forward sales’ to determine the cover, an error is generated.

Inverse: The **cover** function has an inverse function, the **uncover** function**.** This function returns the amount of ‘stock’ required to give a specified number of ‘forward periods cover’. There is no inverse function that ‘solves’ this relationship for ‘sales’ (which is used as a time series, rather than a single value). Note that the inverse can only apply if the <*stockexpression*> is a single measure, rather than an expression.

Examples:

- **cover**(EOP, Sales)
  This provides an EOP based forward cover. There is no ‘wraparound’ sales expression, so this function will generate ***errors*** towards the end of the plan horizon.

- **cover**(BOP, Sales + MD, 0)
  This provides a BOP based forward cover, using Sales plus markdowns as the expression to be covered. There is no 'wraparound' sales expression, so this function will probably generate ***errors*** towards the end of the plan horizon.
- **cover**(EOP, Sales, 1, Sales)
  This provides an EOP based forward cover. Sales itself is used as the 'wraparound sales expression' (this is typical where the plan horizon is a year, since the Sales measure has the appropriate seasonality; where this is not the case, another measure, such as 'next season sales' would be used) so this function will return 'reasonable' values towards the end of the plan horizon, when the cover is greater than the number of weeks remaining.

Notes:

The **cover** function is always calculated at the current time dimension. Thus, for example, in a plan where the bottom time dimension is week, a measure with an aggregation type of 'recalc' that is calculated from a **cover** function at the month level will calculate 'forward months of supply'. If 'forward weeks of supply' are required to be calculated for the month dimension, it would be more appropriate to specify the measure with an aggregation type of first or last, so that aggregation, rather than calculation through the rule, is used to generate the values at the month dimension.

It should be noted that it is the responsibility of the rule writer to ensure that the wrap around expression, if used, is seeded with appropriate values.

Note that both the 'stock' and the 'sales' used in the **cover** function are expressions. This supports various business needs, such as using covers based on 'sales plus markdowns'. Note that if the 'stock' is provided as an expression, rather than just a single measure, the function will not have an inverse.

The offset expression is used to define the offset: which period to start using the sales expression from. It is assumed to be an offset from the current period, so that a value of zero means that the sales for the current period should be used in evaluating the cover (which is appropriate for an 'opening stock' based cover), and an offset of 1 means start in the period following the current period (which is appropriate for a 'closing stock' based cover). Values other than 0 and 1 may be used.

uncover

Returns the amount of ‘stock’ required to cover ‘sales’ for the specified number of forward periods.

The **uncover** function allows for two ‘sales’ expressions, where the second is a ‘wrap around’ expression to provide a well defined cover for periods at or near the end of the calendar horizon that would otherwise ‘run out’ of forward sales. An offset is also specified to allow the **uncover** function to behave appropriately for both opening and closing stock.

Syntax: **uncover**(<coverexpression>, <salesexpression>[, <offsetexpression>, [<wraparoundsalesexpression>]])

Where <*coverexpression*> is an expression or measure that represents the ‘cover value’. <*salesexpression*> is an expression or measure that represents the ‘sales’. <*offsetexpression*> is an expression that calculates a number that represents the offset to apply. If the value is non-integer, only the integer portion is used. If the value is non-numeric, an ***error*** is generated. If <*offsetexpression*> is not provided, the default value will be 1. <*wraparoundsalesexpression*> is an expression or measure that represents the ‘wrap around sales’. If <*wraparoundsalesexpression*> is not provided, there will be no wraparound, and the function will generate ***errors*** if there is insufficient ‘forward sales’ to calculate the stock.

The <*salesexpression*> can be considered to define a time series of sales data values, starting at the current period offset by the <*offsetexpression*>, and stretching until the end of the calendar horizon. If this time series is too short to evaluate the stock value, it can be considered to be extended by one *or more* copies of the time series implied by the <*wraparoundsalesexpression*>, if specified, from the start until the end of the calendar horizon. The stock value is calculated by summing down the time series for a number of periods equal to the integer portion of the <*coverexpression*> and adding the value of the next period, multiplied by the fractional portion of the <*coverexpression*>. If the <*offsetexpression*> causes the start of the time series to be before the start of the calendar horizon, or no <*wraparoundsalesexpression*>, is specified, and there is insufficient ‘forward sales’ to determine the stock, an ***error*** is generated.

Inverse: The **uncover** function has an inverse function, the **cover** function**.** This function returns the number of forward periods of cover implicit in the specified stock. There is no inverse function that ‘solves’ this relationship for ‘sales’ (which is used as a time series, rather than a single value). Note that the inverse can only apply if the <*coverexpression*> is a single measure, rather than an expression.

Examples:

- **uncover**(WOS, Sales)
  This provides an EOP stock value that gives the specified weeks of supply. There is no ‘wraparound’ sales expression, so this function will generate ***errors*** towards the end of the plan horizon.
- **uncover**(WOS, Sales + MD, 0)
  This provides a BOP stock value that gives the specified weeks of supply, using Sales plus markdowns as the expression to be covered. There is no ‘wraparound’ sales expression, so unless the value of WOS is less than 1, this function will generate ***errors*** towards the end of the plan horizon.

- **uncover**(WOS, Sales, 1, Sales)
  This provides an EOP stock value that gives the specified weeks of supply. Sales itself is used as the 'wraparound sales expression' (this is typical where the plan horizon is a year, since the Sales measure has the appropriate seasonality; where this is not the case, another measure, such as 'next season sales' would be used) so this function will return 'reasonable' values towards the end of the plan horizon, when the cover is greater than the number of weeks remaining.

Notes:

The **uncover** function is always calculated at the current time dimension. Thus, for example, in a plan where the bottom time dimension is week, a rule or mapping rule that uses an **uncover** function at the month level will calculate the 'stock' on the assumption that the <*coverexpression*> provides a 'forward months of supply'.

It should be noted that it is the responsibility of the rule writer to ensure that the wrap around expression, if used, is seeded with appropriate values.

Note that both the 'cover' and the 'sales' used in the **uncover** function are expressions. This supports various business needs, such as using covers based on 'sales plus markdowns'. Note that if the 'cover' is provided as an expression, rather than just a measure, the function will not have an inverse.

The offset expression is used to define the offset: which period to start using the sales expression from. It is assumed to be an offset from the current period, so that a value of zero means that the sales for the current period should be used in evaluating the cover (which is appropriate for an 'opening stock' based cover), and an offset of 1 means start in the period following the current period (which is appropriate for a 'closing stock' based cover). Values other than 0 and 1 may be used.

min

Returns the minimum value from a series of expressions or set of measures.

Syntax: **min**(<expression1>, <expression2> [, <expression3> … <expressionn>])

Where < *expression1-n*> are expressions or a set of measures (denoted by {<*measureset*>}) which return numeric values. The function returns the minimum value of the expressions.

Inverse: The **min** function does not have an inverse.

Examples:

- **min** (A, B, C)
  Returns the minimum of the measures A, B and C

max

Returns the maximum value from a series of expressions or set of measures.

Syntax: **max**(<expression1>, <expression2> [, <expression3> … <expressionn>])

Where < *expression1-n*> are expressions or a set of measures (denoted by {<*measureset*>}) which return numeric values. The function returns the maximum value of the expressions.

Inverse: The **max** function does not have an inverse.

Examples:

- **max** (A, B, C)
  Returns the maximum of the measures A, B and C

sum

Returns the sum of a series of expressions or measure set.

Syntax: **sum**(<expression1>, <expression2> [, <expression3> … <expressionn>])

Where <*expression1-n*> are expressions or a set of measures (denoted by {<*measureset*>}) which return numeric values. The function returns the summed value of the expressions or measure set.

Inverse: The **sum** function does not have an inverse.

Examples:

- **sum**(A, B, C)
  Returns the sum of the measures A, B, and C

lag

Returns the value of an expression from the previous time period in the dimension being evaluated.

Syntax: **lag**(<*expression*>)

Where <*expression*> is any valid expression. The function returns the value of the expression in the previous period. If the current period being evaluated is the first period in the calendar horizon (so that there is no previous period) an ***error*** is generated. For that reason, **lag** functions are usually embedded in **if** functions or **prefer** functions to check for that case.

Inverse: The **lag** function does not have an inverse.

Examples:

- **lag**(EOP)
  Returns the value of the measure EOP from the next period.

Notes:

lag is deliberately intended as a 'simple' version of the **timeshift** procedure, for one of the most frequently used cases, which is that the offset is one period in the past. Use the **timeshift** procedure for lagging with a variable offset.

The **lag** function has special 'cycle breaking' logic that enables a series of expressions to be calculated in a manner that allows them to be evaluated 'period wise', thus allowing an apparent 'deadly embrace' to be broken. Thus the two expressions:

EOP = BOP + Rec – SLs – MD and

BOP = lag(EOP)

Are legal, and can be calculated in the same rule group, even though EOP appears to depend on BOP, which appears to depend on EOP. Note, however, that the cycle breaking logic does not support the measure being calculated being lagged on the RHS of the expression. Thus the following expression is illegal:

AccumSls = Sls + lag(AccumSls)

lead

Returns the value of an expression from the next (following) time period in the dimension being evaluated.

Syntax: **lead**(<*expression*>)

Where <*expression*> is any valid expression. The function returns the value of the expression in the following period. If the current period being evaluated is the last period in the calendar horizon (so that there is no following period) an ***error*** is generated. For that reason, **lead** functions are usually embedded in **if** functions or **prefer** functions to check for that case.

Inverse: The **lead** function does not have an inverse.

Examples:

lead(BOP)
Returns the value of the measure BOP from the next period.

Notes:

lead is deliberately intended as a 'simple' version of the **timeshift** procedure, for one of the most frequently used cases, which is that the offset is one period in the future. Use the **timeshift** procedure for leading with a variable offset.

In a similar manner to the **lag** function, the **lead** function has special 'cycle breaking' logic that enables a series of expressions to be calculated in a manner that allows them to be evaluated 'period wise', thus allowing an apparent 'deadly embrace' to be broken.

Even when an error is generated because the current period is the last period in the calendar horizon (see Syntax,) the **lead** function itself, if not guarded by **if** or **prefer** functions, returns the re-evaluated NA value of the measure. For example, for the following expression group:

A = **lead**(B)

B = A + 1

Assume that the NA value for both A and B is 0. The system first re-evaluates B's NA value to be A's NA value + 1 = 1 based on the second expression. Note that the system will attempt to retrieve the time period after B's last time period when A = **lag**(B) is evaluated. Since that time period does not exist, the **lead** function will return B's re-evaluated NA value instead, which is 1.

timeshift

Procedure that returns the value of a measure from a time period in the dimension being evaluated that is lagged by a designated number of periods.

This procedure has special uses and restrictions in RPAS 11.0:

- Measures used in this procedure can be modified with the **master** modifier
- Currently **timeshift** cannot be used in calculation rule groups
- Used for lagging the values of a measure by more than one period
- Used for retrieving values from time periods outside the scope of the workbook
- Used for addressing 52-53 week year differences

Syntax: <output> <- **timeshift**(<input>, { <lagvalue> | <lagmeas> | <lagmap> })

<*input*> is the measure that is being lagged and must have the same base intersection as <*output*> or must be forced to evaluate at the base intersection of <*output*> by using the **level** modifier. <*input*> must include a dimension in the calendar hierarchy and must be the same data type as <*output*>.

<*lagvalue*> is a scalar value that designates the number of periods each position in <*input*> is shifted; negative value refers to shift forward in calendar dimension (lead), positive value refers to shift backward in calendar dimension (lag).

<*lagmeas*> is a numeric measure that contains values that determine how each position is shifted. <*lagmeas*> cannot have a calendar dimension and all non-calendar dimensions must be identical to <*input*>. Note that this implies that if either <input> or <*lagmeas*> measure is modified with the **master** modifier, then the other measure must also be modified with the **master** modifier.

<*lagmap*> is a string measure used for sophisticated mappings. The measure contains position names that indicate how each time period is mapped, and must only contain positions from the dimension from the calendar hierarchy. Multiple positions can be specified by separating them using a space. In other words <*lagmap*> defines a mapping of positions from the input measure to the destination measure along time. Entries in <*lagmap*> that are not the names of valid positions in the dimension from the calendar hierarchy are ignored. Note that this implies that if either <input> or <*lagmap*> measure is modified with the **master** modifier, then the other measure must also be modified with the **master** modifier.

This mapping technique is primarily used when lagging measures between 52 and 53-week years. When mapping multiple positions to a single position (such as mapping the last 2 weeks in a 53-week year to the last week in a 52-week year) the resulting value is the sum of the source values, i.e. the sum of the last 2 weeks of the 53-week year. When mapping a single position to multiple positions (such as mapping the last week in a 52-week year to the last 2 weeks in a 53-week year) the source value is replicated to the resulting values, i.e. weeks 52 and 53 in the 53-week year are updated to week 52 in the 52-week year.

Note: timeshift is a procedure and thus cannot be combined with functions, modifiers, or other procedures in any manner. As a procedure it requires a different syntax: "<-" instead of "=" when being assigned.

Inverse: The **timeshift** procedure does not have an inverse.

Examples:

- salesly <- **timeshift**(sales.**master**, -52)
  Updates the positions in the workbook measure for last year's sales with the values from the domain measure sales where each position is lagged by 52 periods
- salesly <- **timeshift**(sales.**master**, saleslag)
  Where sales and salesly have a base intersection of SKU-week, the numeric measure saleslag contains a value for each SKU that indicates the number of periods to lag by SKU
- salesly <- **timeshift**(sales.**master**, salesmap)
  Where salesmap is a string measure that contains position names indicating which position in sales to use for each position in Salesly; the current year in the workbook contains 52 weeks, the previous year that is not in the workbook contains 53 weeks; the 52nd position in SalesLY contains the position names "week52" and "week53" and results in the sum of the 2 positions

round

Returns the value of an expression rounded up or down to the nearest multiple.

Syntax: **round**(*<expression>*[, *<multipleexpression>*])

Where *<expression>* is any valid expression, which specifies the value to be rounded. *<multipleexpression>* is an expression that calculates a number that represents the multiplier to use. If the value is not specified, it is assumed to be 1. The value may be non-integer. If the value of either expression is non-numeric, an ***error*** is generated. The rounding is up or down to the nearest multiple of the multiplier. If there are 2 multiples equally near to the value (e.g. rounding 1.5 to the nearest integer), then rounding is up, i.e. away from zero.

Inverse: The **round** function does not have an inverse

Examples:

- **round**(qty)
  Returns the value of the measure qty, rounded up or down to the nearest integer. If the qty is 14.324, this returns the result of 14, if the qty is 14.824, this returns the result of 15.
- **round**(qty, packsize)
  Returns the value of the measure qty, rounded up or down to the nearest multiple of the pack size. If the qty is 14.324 and the packsize is 6, this returns the result of 12. If the qty is 16.824 and the packsize is 6, this returns the result of 18.

roundup

Returns the value of an expression rounded up to the nearest multiple.

Syntax: **roundup***(<expression>*[*, <multipleexpression>*]*)*

Where *<expression>* is any valid expression, which specifies the value to be rounded. *<multipleexpression>* is an expression that calculates a number that represents the multiplier to use. If the value is not specified, it is assumed to be 1. The value may be non-integer. If the value of either expression is non-numeric, an ***error*** is generated. Rounding is always up, i.e. to the nearest multiple of the multiplier further away from zero.

Inverse: The **roundup** function does not have an inverse

Examples:

- **roundup**(qty)
  Returns the value of the measure qty, rounded up to the nearest integer. If the qty is 14.324 or 14.824, this returns the result of 15.
- **roundup**(qty, packsize)
  Returns the value of the measure qty, rounded up to the nearest multiple of the pack size. If the packsize is 6 and the qty is 14.324 or 16.824, this returns the result of 18.

rounddown

Returns the value of an expression rounded down to the nearest multiple.

Syntax: **rounddown***(<expression>*[*, <multipleexpression>*]*)*

Where *<expression>* is any valid expression, which specifies the value to be rounded. *<multipleexpression>* is an expression that calculates a number that represents the multiplier to use. If the value is not specified, it is assumed to be 1. The value may be non-integer. If the value of either expression is non-numeric, an ***error*** is generated. Rounding is always down, i.e. to the nearest multiple of the multiplier closer to zero.

Inverse: The **rounddown** function does not have an inverse

Examples:

- **rounddown**(qty)
  Returns the value of the measure qty, rounded down to the nearest integer. If the qty is 14.324 or 14.824, this returns the result of 14.
- **rounddown**(qty, packsize)
  Returns the value of the measure qty, rounded down to the nearest multiple of the pack size. If the packsize is 6 and the qty is 14.324 or 16.824, this returns the result of 12.

Notes on Round functions:

- The round functions have no inverses. Great care should be used in designing rule groups that use these functions, and the preferred technique for rounding is often to not round during calculation, but to round values on display only.
- The round functions cause problems because they can compromise the integrity of rule and expression relationships. Consider a typical relationship between value, units and price. If the units are calculated through a round function (on the apparently reasonable assumption that units should be integers) after a change to, say, the value, then the integrity of the rule relationships is immediately compromised, as the price is no longer the value divided by the units.

navalue

Returns the NA value of the specified expression.

Syntax: **navalue***(<expression>)*

<*expression*> can be a constant, a measure, or an expression.

The **navalue** function does not directly generate ***errors***, but it can propagate errors generated by <*expression*>.

Inverse: The **navalue** function does not have an inverse.

Examples:

- **navalue**(<meas>)
  This returns the NA value of <meas>.
- **navalue**(<meas1> + <meas2>)
  This returns the NA value of the expression "<meas1> + <meas2>". In this example, if the NA value of <meas1> is 2 and the NA value of <meas2> is 5, then the result of the navalue function will be 7.

propspread

Multiple result function that spreads a value across a collection of measures while retaining their relative proportions. The multiple results are not named, and are therefore positional only. The typical usage of this function is to allow spreading of 'hierarchical measures'.

Syntax: ***propspread****(<totalexpression>, <childexp1>, … <childexpn>)*

Where *<totalexpression>* is an expression that returns a numeric value to which to balance the results of the function. *<childexp1>* - *<childexpn>* are expressions that provide the 'shape' of the results. They will typically be the same measures as those assigned to the result of the function, but using the **old** modifier (a measure defined as a result cannot be used on the right hand side without **old**).

The function generates *n* positional results, where *n* is the number of 'child expressions'. The results will sum to the *<totalexpression>*, using the 'shapes' of the child expressions, in the order of the child expressions.

The number of results should be equal to the number of child expressions, meaning there should be one more argument on the right hand side than output measures on the left hand side; additional child expressions are ignored. If too few child expressions are defined then the function will fail. Currently there is no validation to warn an individual when this condition is met.

If the sum of the child expressions is zero, the spread will be even.

Inverse: The **propspread** function does not have an inverse.

Example:

The **old** modifier can be used in conjunction with the **propspread** function to implement a hierarchical relationship among measures. In the following example, Total sales (TotalSls) is the "parent" measure and regular sales (RegSls), promotional sales (PromSls), and markdown sales (MkdSales) are the "child" measures. Using **old** and **propspread** to configure this relationship allows the manipulation of any combination of these measures before calculating, except all of them.

In the following example and in other such hierarchical measure relationships the order of the expressions within a rule is critical for the measures to be correctly calculated.

TotalSls = RegSls + PromoSls + MkdSls

RegSls, PromoSls, MkdSls = **propspread**(TotalSls, RegSls**.old**, PromoSls**.old**, MkdSls**.old**)

PromoSls, MkdSls = **propspread**(TotalSls - RegSls, PromoSls**.old**, MkdSls**.old**)

RegSls, MkdSls = **propspread**(TotalSls - PromoSls, RegSls**.old**, MkdSls**.old**)

RegSls, PromoSls = **propspread**(TotalSls - MkdSls, RegSls**.old**, PromoSls**.old**)

RegSls = TotalSls - PromoSls - MkdSls

PromoSls = TotalSls - RegSls - MkdSls

MkdSls = TotalSls - RegSls - PromoSls

passthrough

Multiple result function that is used to encapsulate any number of normal computations into a single expression.

Syntax: **passthrough**(<*exp1*>, <*exp2*>, …, <*exp-n*>)

Where <*exp1*> - <*exp-n*> are normal expressions used to calculate the resulting measures.

All measures on the left hand side must be computed at the same base intersection. The number of results should be less than or equal to the number of calculation expressions (additional calculation expressions are ignored); if too few calculation expressions are defined then function will fail. Currently there is no validation to warn an individual when this condition is met.

There are two main reasons for using this function:

1. Use **passthrough** in an expression for a rule when you need to compute values for multiple measures without having to write (develop) a multiple-result function or procedure.

2. For performance reasons: If many measures are computed using the same or similar set of RHS measures, combining those calculations using 'passthrough' may be faster because there is less physical input/output with the data.

Inverse: the **passthrough** function does not have an inverse.

Examples:

- A, B = **passthrough**(C + D, C - D)
  Computes the sum and difference of two measures simultaneously.
- SalesA, SalesB = **passthrough**(SalesA.**old** * TotalSales / TotalSales.**old**, SalesB.**old** * TotalSales / TotalSales.**old**)
  Proportionately spread TotalSales down to its components, SalesA and SalesB.

String Functions

uppercase

Converts a string to upper case.

Syntax: **uppercase**(<*expression*>)

Where the value of <*expression*> is returned as a string with upper case characters. Useful in making string comparisons.

lowercase

Converts a string to lower case.

Syntax: **lowercase**(<*expression*>)

Where the value of <*expression*> is returned as a string with lower case characters. Useful in making string comparisons.

Math Functions

pow

Returns the value of a number raised to the power of another number (x to the power of y).

Syntax: **pow**(<*x*>, <*y*>)

<*x*> and <*y*> are expressions that return real numbers. <*y*> designates the exponent to which <*x*> is raised.

exp

Returns the value of the transcendental number “e” raised to the power of a number (e to the power of x).

“e” is the base of natural logarithms.

Syntax: **exp**(<*x*>)

<*x*> is an expression that returns a real number to which the number “e” (value 2.71828183) is raised.

sqrt

Returns the square root of a number.

Syntax: **sqrt**(<*x*>)

<*x*> is an expression that returns a real number. This function returns the equivalent of “pow(x, 0.5)”.

log

Returns the logarithm of a number.

This function returns the exponent that indicates the power to which a number is raised to produce a given number.

Syntax: **log**(<*x*>, [<*base*>])

Where <*x*> and <*base*> are expressions that return real numbers. If <*base*> is not specified the default value is 10.

Examples:

- **log**(100)
  The logarithm of 100 to the base 10 is 2.
- **log**(125, 5)
  The logarithm of 125 to the base 5 is 3.

ln

Returns the natural logarithm of a number.

This function returns the logarithm of <*x*> to base “e” (2.71828183).

Syntax: **ln**(<*x*>)

<*x*> is an expression that returns a real number.

mod

Returns the remainder as the result of the division of 2 numbers.

The result of this function is the remainder of <*x*> divided by <*y*>.

Syntax: **mod**(<*x*>, <*y*>)

<*x*> and <*y*> are expressions that return real numbers.

Example:

- **mod**(5, 2)
  The remainder of 5 divided by 2 is 1.

abs

Returns the absolute value of a number.

Syntax: **abs**(<*x*>)

<*x*> is an expression that returns a real number.

Appendix A – Aggregation/Spread Types

The following section describes the aggregation and spread types that are supported in RPAS 11.

Aggregation Type	Description	Valid Data Types	Recommended Spread Types
recalc	recalculate measure at each level, via recalc expression	numeric, string, date, Boolean	none
total	sum of all values	numeric	prop
total_pop	sum of all populated values	numeric	prop_pop
average	average of all values	numeric	prop
average_pop	average of all populated values	numeric	prop_pop
min	minimum of all values	numeric, date	repl
min_pop	minimum of populated values	numeric, date	repl_pop
max	maximum of all values	numeric, date	repl
max_pop	maximum of populated values	numeric, date	repl_pop
median	median of all values	numeric	repl
median_pop	median of populated values	numeric	repl_pop
pst	first value in time hierarchy, total in others hierarchies	numeric	ps
pet	last value in time hierarchy, total in other hierarchies	numeric	pe
and	and of all values	Boolean	repl
or	or of all values	Boolean	repl
ambig	ambig of all values (all values equal, otherwise ambig)	string	none
ambig_pop	ambig of all populated values	string	none
popcount	count of populated values in base	numeric, string, date, Boolean	none

Spread Type	Description	Valid Data Types
none	values are not spread	numeric, string, date, Boolean
prop	spread value proportionally (previous total non-zero) or evenly (previous total zero)	numeric
prop_pop	spread value proportionally (previous total non-zero) or evenly (previous total zero) to all populated cells	numeric
even	spread value evenly	numeric
even_pop	spread value evenly to all populated cells	Numeric
delta	increment/decrement each cell evenly. Effectively the 'even' spreading of the change ('delta').	numeric
delta_pop	increment/decrement each cell evenly over the populated cells. Effectively the 'even' spreading of the change ('delta') over the populated cells.	numeric
repl	replicate the value to each cell (this is not possible with recalc measures as a default spread type in RPAS 11.1)	numeric, string, date, Boolean
repl_pop	Replicate the value to each cell that is populated (not possible as a default spread type with recalc measures in RPAS 11.1)	numeric, string, date, Boolean
ps	apply delta to starting period	numeric
pe	apply delta to ending period	numeric

Please note the following points about specifying "none" as the default spread method:

- For recalc measures the aggstate and basestate measure attributes are set to the value specified by the user; these settings do not depend on the default spread method that is designed for the measure
- For the popcount aggregation type the aggstate measure attribute is always set to read-only no matter what is set by the user
- For measures that do not fit into the above criteria the aggstate is always read-only when the default spread method is set to "none"

Please note that the definition of a "populated" cell is one with a value different than the *naval* of a measure.

Appendix B – Arithmetic Operators

The following arithmetic operators are supported:

Unary Operators

-	Real	negation
!	Boolean	compliment

Binary Operators

=	real, Boolean, string, date	assignment
+	Real	addition
-	Real	subtraction
*	real	multiplication
/	real	division
&&	Boolean	Boolean and
||	Boolean	Boolean or
==	real, Boolean, string, date	equality
!=	real, Boolean, string, date	inequality
<	real, Boolean, string, date	less than
<=	real, Boolean, string, date	less than or equal to
>	real, Boolean, string, date	greater than
>=	real, Boolean, string, date	greater than or equal to