Calculated Columns in a database

Software Support - 2024-04-02 - Databases

Calculations

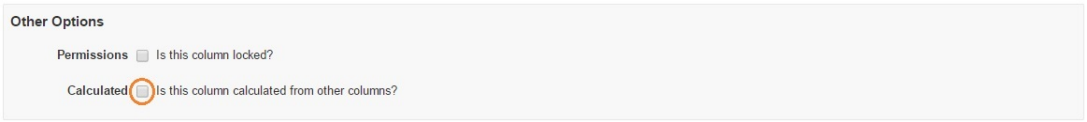
You can set a text, long text, formatted text, number or date column in a Kahootz Database to be a calculated column instead of a user having to enter values directly.
This means that the values in that column are calculated based on other values in the entry each time it is saved.
For example, adding up a set of other number columns or combining some text and displaying that value within the calculated column for you.

Your calculation can be a simple expression, such as adding or multiplying values, or it can contain code and logic.
You can also use a range of operators and functions within your calculation to obtain the required data.
More information about the operators and functions you can use is given below.

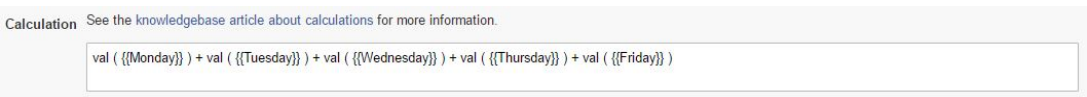
If you add or update a calculation on a column, the existing database entries will be updated in the background.
For a Database with many rows, this can take some time; calculations will also be updated when you add or change an entry.

> A calculation column is added after initially creating a database by following these simple steps below.

1. Open the Database you want to add a calculated column, and under the "Actions" section, select "manage database" link.
2. Select the column and at the bottom of the page, under "other options" tick the checkbox for calculated as shown below.



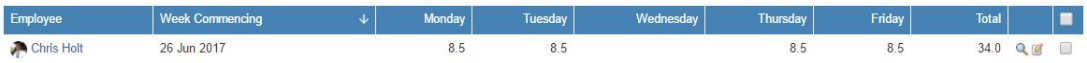
3. Add an expression or code and click save; see the screenshot below.



* This would allow the following Database below to add all the hours from each weekday, totalling them into the last column.

> **Kahootz Tip:** Adding val ({{column}}) ignores any blank columns within

your Database.



Employee	Week Commencing	Monday	Tuesday	Wednesday	Thursday	Friday	Total
Chris Holt	26 Jun 2017	8.5	8.5		8.5	8.5	34.0

Please see below for more details on formats, expressions & coding.

Basic Format

You can use the value of other columns in your calculation by putting the column name between {{ and }}.
A list of the available column names will be shown on the **add / modify column** page when you're adding calculations, and you can click on them to insert them into the calculation.

You can use round brackets () to make sure your calculation is evaluated in the order you expect - so (2 + 4) / 2 will do the addition first, then the division - giving 3.
Normal mathematical precedence will apply without the brackets, 2 + 4 / 2, so it will do 4 / 2 first, then + 2, giving 4.

You can search and sort on calculated columns.

You can use one calculated column in another calculation. They are evaluated in column order, so if you want to use one calculation in another, ensure the first one is higher up the column order.

If there is an error from your code due to particular inputs for a row, the column will be set to blank.

Simple Expressions

Examples

Adding two number columns together: **{{days on activity 1}} + {{days on activity 2}}**

Multiplying two number columns together: **{{cost per hour}} * {{total hours}}**

Showing one number column as a percentage of another: **{{hours spent on activity}} / {{total hours}} * 100**

Showing that percentage as a whole number (no decimal places):
int({{hours spent on activity}} / {{total hours}} * 100)

Work out the number of days between two dates: **daysBetween({{start date}}, {{end date}})**

Numeric operators

You can use the following operators to combine numbers:

Addition	***number1* + *number2***	Example: To calculate the total number of days spent on activity1 and activity2: `{{days on activity1}} + {{days on activity2}}`
Subtraction	***number1* - *number2***	Example: To calculate the cost after discount: `{{Original cost}} - {{discount}}`
Multiplication	***number1* * *number2***	Example: To calculate the total cost of a number of items: `{{Cost per item}} * {{number of items}}`
Division	***number1* / *number2***	Example: To calculate the cost per hour: `{{Total cost}} / {{number of hours}}`
Integer division	***number1* \ *number2***	How many times one number can be divided by another in whole numbers, ignoring the remainder. eg:`5 / 2 = 2.5` but `5 \ 2 = 2`
Division-remainder	***number1* mod *number2***	The remainder after dividing one number by another. eg: `5 mod 2 = 1` `(2*2=4, with 1 remaining)`
Blank column/value	**val ({{column}})**	In all these operators, if a numeric column/value is blank it'll be treated as an error, to treat empty columns as 0. Use this expression - please refer to the example/screenshot above.

Number Functions

You can use the following functions to manipulate numbers:

Absolute value	**abs(*number*)**	The absolute value of a number is the number without a sign, eg: `abs(2) = 2` and `abs(-2) = 2`

Rounding	**round(*number*)**	Gives the closest whole number, rounding up or down as nearest. eg: `round(1.1) = 1` and `round(1.9) = 2` Halves will be rounded to the nearest even number to avoid bias eg: `round(1.5) = 2` and `round(2.5) = 2` and `round(3.5) = 4`
Rounding up	**ceiling(*number*)**	Gives the closest whole number, always rounding up. eg: `ceiling(1.1) = 2` and `ceiling(1.9) = 2`
Rounding down	**int(*number*)**	Gives the closest whole number, always rounding down. eg: `int(1.1) = 1` and `int(1.9) = 1`
Maximum	**max(*number1*, *number2*)**	Return the maximum of *number1* and *number2*. Only handles two numbers, not more.
Minimum	**min(*number1*, *number2*)**	Return the minimum of *number1* and *number2*. Only handles two numbers, not more.

Text Operator

Joining text	***test1* & *text2***	Note that this does not use + which is for adding numbers. You will also need to put in spaces explicitly where wanted. eg: `{{first name}} & " " & {{surname}}`

Text Functions

Comparing text	**compare(*text1*, *text2*)** **compareNoCase(*text1*, *text2*)**	Performs a case-sensitive or insensitive comparison of two text columns. Return a negative number if text1 is less than text2; returns 0 if text1 is equal to text2; returns a positive number if text1 is greater than text2.
Find position	**find(*text_to_find*, *text*)** **findNoCase(*text_to_find*, *text*)**	Finds the first occurrence of a *text_to_find* in *text*. find is case sensitive, findNoCase is not. Returns the position of *text_to_find* in *text*; or 0, if *text_to_find* is not in *text*

Insert at position	**insert(*text_to_insert*, *text*, *position*)**	Return text with *text_to_insert* inserted into *text* after character *position*. If position=0, it prefixes *text_to_insert* to *text*. eg:`insert(" My ","Hello Friend",5)` `returns "Hello My Friend"`
Remove from position	**removeChars(*text*, *start_position*, *num_chars*)**	Return a *text* with *num_chars* removed starting at position *start_position*. eg:`removeChars("Hello Friend",5,7)` `returns "Hello"`
Convert to lower case	**lcase(*text*)**	Return *text* converted to lower case.
Convert to upper case	**ucase(*text*)**	Return *text* converted to upper case.
Reverse	**reverse(*text*)**	Return *text* in reverse order. eg: `reverse("kahootz") returns "ztoohak"`
Length of text	**len(*text*)**	Return the length - how many characters - are in *text*. Includes spaces and other punctuation.
Characters from left	**left(*text*, *num_chars*)**	Return the leftmost *num_chars* characters of *text*. Counting includes spaces and other punctuation.
Characters from right	**right(*text*, *num_chars*)**	Return the rightmost *num_chars* characters of *text*. Counting includes spaces and other punctuation.
Characters from position	**mid(*text*, *start_position*, *num_chars*)**	Return *num_chars* of characters from *text* starting at position *start_position*.eg: `mid("kahootz",3,4) returns "hoot"`
Find and replace	**replace(*text*, *remove*, *insert* [, *scope*])** **replaceNoCase(*text*, *remove*, *insert* [, *scope*])**	Return *text* with occurrences of *remove* replaced by *insert*. If the *scope* is "1" then just the first occurrence is replaced. If the *scope* is "ALL" then all occurrences are replaced. (Versions using Regular Expressions for very advanced use are available - ask support!)
Substring until	**spanExcluding(*text*, *characters_to_exclude*)**	Return characters from *text*, from the beginning until the first character in *characters_to_exclude*. The search is case sensitive, so if you want to stop at either A or a, then put both in *characters_to_exclude*. eg: `spanExcluding("kahootz.doc",".,/")` `returns "kahootz"`
Substring until not	**spanIncluding(*text*, *characters_to_include*)**	Return characters from *text*, from the beginning until the first character that is NOT in *characters_to_include*. The search is case sensitive, so if you want to include both A and a, then put both in *characters_to_include*. eg: `spanIncluding("aardvark","aeiou")` `returns "aa"`
Trim spaces	**trim(*text*)**	Return *text* with any leading and trailing spaces removed.
Trim leading spaces	**ltrim(*text*)**	Return *text* with any spaces at the beginning removed.
Trim trailing spaces	**rtrim(*text*)**	Return *text* with any spaces at the end removed.
Convert to number	**val(*text*)**	Return *text* converted to a number. Handles decimal places. Text that can't be returned to a number will cause an error, and thus a blank calculated column (but see conditional operator 'isNumeric() in the code section below)

Dates and Times

Date values in the following functions can either be taken from columns (of date, date and time, month and year, entry creation date / date-time or entry modify date / date-time types) or entered as explicit dates in the format *yyyymmdd* - eg 20170401 is 1st April 2017

Time values in the following functions can either be taken from columns (of date and time, time, entry creation date-time or entry modify date-time types) or entered as explicit times in the format *hhmmss*

To show a calculated value in a 'date' column the result must be a valid date, but you can use the other result formats in text or number columns.

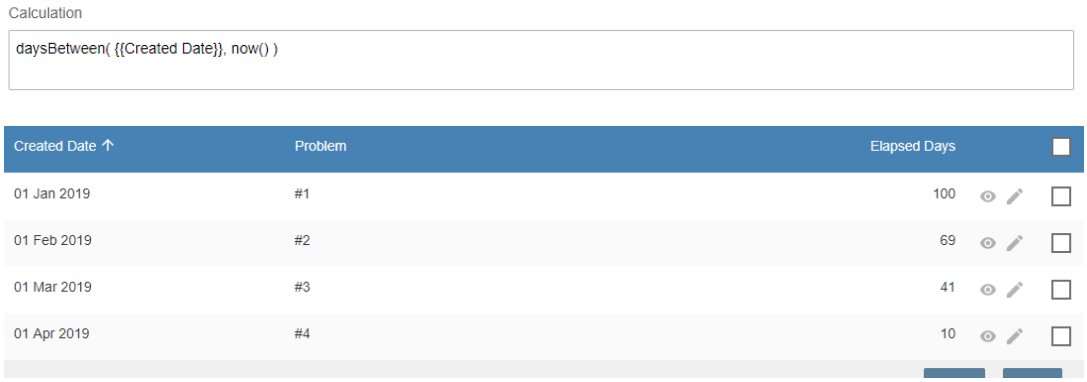
Date Functions - returning a number

Day of Week	**dayOfWeek(*date*)**	Return a number for the day of the week of *date* in the range 1 (Sunday) to 7 (Saturday)
Day Of Year	**dayOfYear(*date*)**	Return a number of the day of the year, in the range 1 (1st Jan) - 365 (31st Dec - or 366 in leap year)
Days in Month	**daysInMonth(*date*)**	Returns the number of days in the specified month (ie: 28, 29, 30 or 31)
Days In Year	**daysInYear(*date*)**	Return the number of days in the specified year (ie: 365 or 366 for leap years)
Parts of a Date / Time	**year(*date*)** **month(*date*)** **day(*date*)** **hour(*time*)** **minute(*time*)**	Return a number for the appropriate part of the specified date/time. Year is returned in four figures (2017); Month as 1-12; Day as 1-31; Hour in 24-hour notation as 0-23; Minute as 0-59

Days after	**daysAfter(*date1*, *date2*)**	Return the number of days that *date2* is after *date1*. If *date2* is before *date1*, a negative number is returned. If either is not a valid date, then empty text is returned.
Days between	**daysBetween(*date1*, *date2*)**	Return the number of days between *date1* and *date2*. It doesn't matter which date is earlier, and will always return a positive number. If either is not a valid date, then empty text is returned.
Date / Time Difference	**dateDiff(*datepart, date1*, *date2*)**	Return the number of "units" by which *date1* is less than *date2*. *datepart* should be one of the following strings "yyyy": Years"q": Quarters (any 3 month period)"m": Months"d": Days"ww": Weeks"h": Hours"n": MinutesIf *date2* is before *date1*, a negative number is returned. If either is not a valid date, then empty text is returned.
Date / Time Comparison	**dateCompare(*date1*, *date2*)**	Return -1 if *date1* is earlier than *date2*; Return 0 if *date1* is the same as *date2*; Return 1 if *date1* is later than *date2*; Accurate to the second if used with date-times or times.

Current Date	**now ()**	Uses the date the entry was last saved or updated of which can be used in various calculations, see below.

For example, you have a database using a "date" column and you want to return the total number of days the entries have been open/outstanding.
You can use this function to show the elapsed days between the created date and today's date by adding "now ()" to the calculation, as shown below.



> **Kahootz Tip:** The example above **will not update automatically**, therefore, when you view the database the next day - the values will not have changed.
> The calculation for "current date" uses the date of when the calculation was last saved/updated - (please **remember** this if you're going to use this value)

Date Functions - returning a date

Add to / Subtract from a date	**dateAdd(*datepart, number, date*)**	Return a new date by adding the specified *number* of units to *date*. *datepart* should be one of the following strings "yyyy": Years"q": Quarters"m": Months"d": Days"w": Weekdays (Mon-Fri, skipping Sat and Sun. Simple addition, not aware of public holidays etc)"ww": Weeks"h": Hours"n": MinutesIf *number* is positive you'll get dates after date, ie: forwards in time. To go backwards in time use a negative value for *number*.
Create Date	**createDate(*year, month, day*)**	Create a date from three numbers, eg: *CreateDate(2017,2,14)* represents 14th Feb 2017
Create Date - Time	**createDateTime(*year, month, day, hour, minute, second*)**	Create a date-time from six numbers, eg: *CreateDateTime(2017,2,14,15,5,17)* represents 14th Feb 2017 15:05:17 - just after 3pm

Comma-Separated Lists

ListFirst	**ListFirst({{column}}, delimiter)**	Returns the first element in a list: This function will return the first element in a list delimited by the character specified in the expression. For example: {{ column }} is a,b,c,d,e,f using the expression ListFirst({{column}}, ',') will return a
ListLast	**ListLast({{column}}, delimiter)**	Returns the last element in a list: This function will return the last element in a list delimited by the character specified in the expression. For example: {{ column }} is a,b,c,d,e,f using the expression ListLast({{column}}, ',') will return f
ListRest	**ListRest({{column}}, delimiter)**	Returns all but the first element from a list: This function will return the list without the first element in the list as delimited by the character specified in the expression. For example: {{ column }} is a,b,c,d,e,f using the expression ListRest({{column}}, ',') will return b,c,d,e,f
ListGetAt	**ListGetAt({{column}}, pos, delimiter)**	Returns the element in the specified position from a list: This function will return a single element from the delimited list at a position specified in the expression. For example: {{ column }} is a/b/c/d/e/f using the expression ListGetAt({{column}}, 3, '/') will return c
ListFind	**ListFind({{column}}, value, delimiter)**	Returns the index of the list that is matched by the value supplied. If no match is found, 0 will be returned. For example {{ column }} is Orange,Apple,Banana using the expression ListFind({{ column }}, 'Banana', ',') will return 3. As ListFind is case sensitive if the expression were ListFind({{ column }}, 'banana', ','), the value returned would be 0.
ListFindNoCase	If you are unsure of the case of the value use: **ListFindNoCase({{column}}, value, delimiter)**	Returns the index of the list that is matched by the value supplied whilst ignoring the case of the value and column. If no match is found, 0 will be returned. For example {{ column }} is oRaNGe,aPPLe,BaNaNa using the expression ListFind({{ column }}, 'banana', ',') will return 3
ListDeleteAt	**ListDeleteAt({{column}}, position, delimiter)**	Returns the list after removing the element specified with the position value. For example: {{ column }} is hop,skip,run,jump, using the expression ListDeleteAt({{column}}, 3, ',')will returnhop,skip,jump`
ListAppend	**ListAppend({{column}}, value, delimiter)**	Returns a list with a new value added to the end of the list. For example: {{column}} is Hola/Bonjour/Salve, using the expression ListAppend({{column}}, 'Hello', '/') would return the Hola/Bonjour/Salve/Hello
ListPrepend	**ListPrepend({{column}}, value, delimter)**	Returns a list with a new value added to the start of the list. For example: {{column}} is Hola,Bonjour,Salve, using the expression ListPrepend({{column}}, 'Hello', ',') would return the Hello,Hola,Bonjour,Salve
ListRemoveDuplicates	**ListRemoveDuplicates({{column}}, delimiter)**	Returns a value where any duplicate values have been removed from the list. For example {{column}} is dog,cat,bird,dog,fish,rabbit, using the expression ListRemoveDuplicates({{column}}, ',') would return dog,cat,bird,fish,rabbit.

ListSort	**ListSort({{column}}, sortType, sortOrder, delimiter)**	Returns a list where each element has been sorted. Options for sortType are: numeric and text. The options for sortOrder are: asc and desc. To sort a {{column}} whose value is 8,3,9,12,1,5,2 in ascending numerical order the expression would be ListSort({{column}}, 'numeric', 'asc', ',') and the value would be 1,2,3,5,8,9,12
ListLen	**ListLen({{column}}, delimiter)**	Returns the number of elements in a list: This function will return the number of elements in a list as delimited by the character specified in the expression. For example: {{ column }} is a;b;c;d;e;f using the expression ListLen({{column}}, ';') will return 6
PatternFind	**PatternFind({{column}}, mask, startPos)**	Returns a value from the column that matches the pattern specified in the expression. If a startPos is supplied, any characters before the startPos position will be ignored when matching for patterns. Using this expression, it is possible to utilise pattern masks that behave like input masks used for sanitising user input into forms. More information on Masks can be found at https://help.kahootz.com/kb/articles/input-masks

A mask can form a series of characters to represent the characters matched by the pattern. PatternFind will only return the first match found, so if multiple possible matches are in a column, only the first match will be returned.

For example: Where {{column}} is abc1 def2 ghi3

The expression patternFind({{column}}, 'AAAN', 1) attempts to match any string that has 3 mandatory letters followed by a number.
While each element in the column would match the pattern, the value abc1 would be returned from the function.

PatternFindPosition({{column}}, mask, startPos) Returns the start position of a value from the column that matches the pattern specified in the expression.
If a startPos is supplied any characters before the startPos position will be ignored when matching for patterns.

Writing Code

You can also write code to make decisions, as well as simple expressions.

There is a range of tags and logical operators for this. You can also use variables in your code to store intermediate values.

When you write code, you must set a variable called *calcResult,* which will be displayed in the Database cell.

> **Kahootz Tip:** when writing code, there is a maximum character limit of 50,000

Tags

<SET *var_name* = *expression*> - Set the variable *var_name* to the result of calculating

expression.

Kahootz Tip: Note that all variable names must begin with the string "var_".

Conditional Operators - return true or false, used in IF or ELSEIF conditions

The logical operators **AND**, **OR** and **NOT** are supported, returning true or false

value1* EQ *value2 - Test if *value1* equals *value2* - works on both numbers and text (case insensitive)

value1* NEQ *value2 - Test if *value1* is not equal to *value2*- works on both numbers and text (case insensitive)

number1* GT *number2 - Test if *number1* is greater than *number2*

number1* GTE *number2 - Test if *number1* is greater than or equal to *number2*

number1* LT *number2 - Test if *number1* is less than *number2*

number1* LTE *number2 - Test if *number1* is less than or equal to *number2*

isNumeric(*text*) - Test if *text* can be converted to a number - true if it can, false if it can't.
(eg: can be used to check if something is a valid number and explain the error if it can't rather than let the calculation fail and return blank.)

Using Conditions

<IF condition> **CODE** **</IF>**	Executes CODE if the *condition* is true. The format of a condition block starts with <IF condition> and it requires an end marked </IF> .
<IF condition1> **CODE1** **<ELSEIF condition2>** **CODE2** **<ELSEIF condition3>** **CODE3** **<ELSE>** **CODE4** **</IF>**	This is an example of conditional branching, it executes one of the CODE blocks depending on which condition is true. If none of the conditions are true then the CODE following <ELSE> is executed (CODE4). You can have as many <elseif *condition*> blocks as you like, but only one <else>. Again the conditional IF statement must finish with the end marker </IF> .

Example

<SET var_daysAfterTargetDate = daysAfter({{delivery date}}, {{planned delivery date}})>

<IF var_daysAfterTargetDate EQ "">

<SET calcResult = "Bad date!"><ELSEIF var_daysAfterTargetDate LT 0>

<SET calcResult = "Early"><ELSEIF var_daysAfterTargetDate EQ 0>

<SET calcResult = "On time"><ELSE>

<SET calcResult = "Late"></IF>

Related Content

- Evaluated Columns in a Database
- How-To use Linked Databases
- Linking Databases Together
- Database Column Types & Maximum Character Limits
- Using a database for time recording