

KNIME Expressions Guide

KNIME AG, Zurich, Switzerland
Version 5.5 (last updated on)



Table of Contents

- Introduction. 1
- Expression nodes 1
 - General Behaviour 1
 - Expression node 2
 - Expression Row Filter node 3
 - Variable Expression node 4
- Expression Language. 5
 - Value types and literals. 6
 - Input data access 11
 - Operators 13
 - Functions 22
 - Constants. 24

Introduction

In this guide you will find documentation about:

- The **Expression nodes**: Learn how to perform versatile data manipulation in KNIME workflows using the KNIME Expression Language.
- The **KNIME Expression Language**: Find guidance for the syntax, semantics, and usage of the KNIME Expression Language.

Expression nodes

There are currently three nodes available in KNIME Analytics Platform that allow you to use the KNIME Expression Language to manipulate your data within KNIME workflows:

- **Expression**: Enables row-by-row data manipulation to add or replace columns.
- **Expression Row Filter**: Filters rows based on a condition.
- **Variable Expression**: Allows you to create or modify flow variables.

Simply drag and drop one of the nodes from the node repository and connect it.

General Behaviour

You can use the KNIME Expression nodes to perform manipulation of your data. The nodes use the KNIME Expression Language that you can find explained in the [next section](#).

You can write your expression in the expression editor, using the input data available from the input panel which can be found on the left side. On the right side you can find the catalog of all the available functions with documentation about their usage. You can filter them and also expand or collapse the available categories.

Most Expression nodes support multiple expressions that will be evaluated in sequence. You can add a new expression by clicking the `Add expression` button. Each single expression editor has a *control bar* in the top-right corner that allows you to move the expression up or down, duplicate it, or delete it.

Every expression editor has an output section attached below it that allows to configure the settings of the output. Editors will be evaluated from the top to the bottom, so you can use the result of one editor in the next one.

There is a button to evaluate the expression and generate a preview of the result. This is

useful to check if the expression is correct and to see the result of the manipulation.

Additionally, the node integrates with the KNIME AI Assistant extension, which offers AI-assisted expression generation and modification, further simplifying the process. By asking K-AI for assistance, you can receive suggestions for expressions based on the column names and column types in your table.



Even with K-AI enabled, no data from the table is sent to the AI service.

Expression node

You can use the Expression node to perform row-by-row manipulation of your data and add or replace column data.

Open the node configuration dialog and you will see something like the following:

Inputs & Outputs
Available input data (via table or flow variable) usable in the expression. Double click or drag&drop to insert.

Expression editors
Multiple code editors to craft expressions. Provides auto completion and displays errors. Copy, reorder or delete editors with the control bar in the top right.

Output target
Choose where the result of the expression is stored. Replace an existing column or create a new one.

Function catalog
Built-in functions to manipulate data. Click on a function to see an extensive documentation about the usage of every function. Double click or drag&drop to insert.

Result preview
Evaluate the expression for a limited number of rows to see whether your expression works as expected.

The screenshot shows the configuration dialog for the Expression node. It includes an 'Inputs & Outputs' section on the left, two 'Expression' editors in the center, a 'Function catalog' on the right, and an 'Output target' dropdown. At the bottom, there is a console, an 'Evaluate first 10 rows' button, and a table preview showing the results of the expressions.

RowID	Universe_0_0 Number (double)	Universe_0_1 Number (double)	Universe_1_0 Number (double)	Universe_1_1 Number (double)	Cluster Membership String	mean_of_the_whole_universe Number (double)
Row0	0.557	0.211	0.962	0.432	Cluster_0	0.541
Row1	0.641	0.138	0.973	0.92	Cluster_0	0.668
Row2	0.616	0.305	0.9	0.525	Cluster_0	0.586
Row3	0.627	0.343	0.951	0.48	Cluster_0	0.377
Row4	0.648	0.226	0.645	0.635	Cluster_0	0.538
Row5	0.815	0.289	0.176	0.956	Cluster_0	0.559
Row6	0.666	0.353	0.903	0.48	Cluster_0	0.666

Figure 1. KNIME Expression node overview

In the *Output column* section at the bottom of each expression editor you can choose if you want to output the result of the expression in a new column and give the column a desired name or replace the existing column.



Find an example on how to use the expression node on [KNIME Community Hub](#).

You can click the *Evaluate first 10 rows* button or select the amount of rows you want to evaluate by clicking the icon. You can choose between 10 (default), 100, 1000.



Take into account that this will take more time based on the number of rows to be evaluated.

Expression Row Filter node

You can use the Expression Row Filter node to remove rows based on a condition defined in the KNIME Expression Language.

Open the node configuration dialog and you will see something like the following:

The screenshot shows the configuration dialog for the Expression Row Filter node. It includes several panels and components:

- Inputs & Outputs:** Located on the left, it lists available input tables (e.g., Universe_0_0, Universe_0_1) and output tables. A red arrow points to the 'Flow variables' section.
- Expression editor:** The central area where the filter expression is written. It contains a multi-line expression using the 'and' operator to filter rows based on column values and row indices. A yellow button labeled 'Evaluate first 10 rows' is visible below the editor.
- Output panel:** Located at the bottom, it displays a preview of the filtered data as a table with columns and row values.
- Function catalog:** On the right, it provides a search bar and a list of built-in functions categorized by type (e.g., Math, Aggregate, Trigonometry). A yellow arrow points to the 'sum' function.
- Result preview:** Below the function catalog, it shows examples of how the 'sum' function is used with different arguments and returns values.

Annotations with lines pointing to specific parts of the dialog include:

- Inputs & Outputs:** Available input data (via table or flow variable) usable in the expression. Double click or drag&drop to insert.
- Expression editor:** Single code editor to craft an expression. Provides autocomplete and displays errors in case the expression does contain errors.
- Output panel:** The node filters table rows depending on the given boolean expression. This operation does not need additional output settings.
- Function catalog:** Built-in functions to manipulate data. Click on a function to see an extensive documentation about the usage of every function. Double click or drag&drop to insert.
- Result preview:** To check if your expression is filtering rows correctly, review the output with a limited number of rows. Keep in mind, the set row limit applies to the input rows. If any rows are filtered out, the preview will display fewer rows than the set limit.

Figure 2. KNIME Expression Row Filter node overview

The Expression Row Filter node has a single expression editor where you define the condition for filtering. If the expression evaluates to `FALSE`, the row will be removed from the output table. If it evaluates to `TRUE`, the row will remain in the output. The output of the expression must therefore be a `BOOLEAN` value (for more details regarding types, see the [Types](#) section).

If you need to remove rows based on multiple conditions, you can use logical operators like `and`, `or`, and `not` to combine these conditions (see the [Logical Operators](#) section for more details and note the example in [Figure 2](#)).

You can click the *Evaluate first 10 rows* button or select the amount of rows you want to evaluate by clicking the \vee icon. You can choose between 10 (default), 100, 1000.



The limit is applied to the input rows, not the output rows. If any rows are filtered out, the preview will display fewer rows than the set limit. Take into account that the more rows you evaluate, the more time it will take.

Variable Expression node

You can use the Variable Expression node to create or modify flow variables using the KNIME Expression Language. This node is useful when you want to create a new flow variable or modify an existing one based on the values of other flow variables. There is no table input for this node.

Open the node configuration dialog and you will see something like the following:

The screenshot shows the configuration dialog for the Variable Expression node. It includes several sections with annotations:

- Inputs:** Points to the 'Flow variables' section on the left, which lists available input flow variables like 'int_flow_variable', 'long_flow_variable', and 'knome.workspace' with their respective data types (INTEGER, STRING).
- Expression editors:** Points to the central 'Expression (1)' text area containing KNIME Expression Language code. It also points to the 'Output variable' dropdown menu at the bottom of the editor, which has options like 'Append', 'Replace', and 'New int_flow_variable'.
- Function catalog:** Points to the right-hand pane showing a search bar and a list of built-in functions such as 'atanh', 'ceil', 'count', 'lower_case', etc. An annotation explains that users can double-click or drag&drop functions to insert them into the expression.
- Output type:** Points to the 'Return value' section in the function catalog, which specifies the data type (e.g., INTEGER, MISSING) and provides examples of how the output is converted to an IntType.
- Result preview:** Points to the 'Preview' tab at the bottom, which displays a table of the results generated by the expression. The table has columns for Owner ID, Data Type, Variable Name, and Value.

Owner ID	Data Type	Variable Name	Value
	IntType	new_int_flow_variable	3
	StringType	knome.workspace	/home/kampmann/eclipse-workspace-24-03
3:1	LongType	long_flow_variable	2
3:1	IntType	int_flow_variable	1

Figure 3. KNIME Variable Expression node overview

In the *Output flow variable* section at the bottom of each expression editor you can choose if you want to output the result of the expression as a new flow variable and give the flow variable a desired name or replace an existing flow variable.

You can click the *Evaluate* button to evaluate the expression and generate a preview of the result. This is useful to check if the expression is correct and all flow variables are available as expected.



As explained **later** in this guide, the KNIME Expression Language only supports one type of integral numbers (INTEGER) while KNIME Analytics Platform supports two types of integral numbers for flow variables: IntType and LongType. Per default the output of a numerical expression will be of the flow variable type LongType. If you want to output a IntType, there is a dropdown menu in the output section of each expression editor where you can select the desired type.

Expression Language

The KNIME Expression Language is a specialized language designed for data manipulation and analysis within KNIME workflows. Its purpose is to provide an intuitive and efficient way for users to perform calculations, string manipulations, and row or column-based operations without the need for extensive programming knowledge. This document serves as a guide for the syntax, semantics, and usage of the KNIME Expression Language.

Value types and literals

The KNIME Expression Language supports several basic value types, each serving a specific kind of data. Some operations are only valid for a subset of value types. This is described where relevant. Every type can be optional, see the section about `MISSING` for more information.



¹ KNIME Expression Language uses slightly different types than KNIME Analytics Platform uses for **column types** and **flow variables**. The latter two differ only in the identifier names. In the following the types of the expression language are described and how they map to the combined types noted as *[column type , flow variable type]* in KNIME Analytics Platform.

BOOLEAN

The value type `BOOLEAN` is used for logical values, that are either true or false. If handling optional values, i.e., type `BOOLEAN | MISSING`, **Kleene's three-valued logic** will be applied. For more details see section **General rules for comparison operators**.

`BOOLEAN` literals are either `TRUE` or `FALSE`.

`BOOLEAN` in the expression language maps to the `[Boolean, BooleanType]`¹ in KNIME Analytics Platform.

Number types - INTEGER and FLOAT

The KNIME Expression Language only supports one type of integral numbers (`INTEGER`) and one type of floating point numbers (`FLOAT`). For simplicity, different precisions are not supported.



For operations applied to `INTEGER` and `FLOAT`, such as `5 * 3.14`, the `INTEGER` value is converted to `FLOAT` automatically. This may cause a loss of precision for very large numbers.

INTEGER

The value type `INTEGER` is used for whole numbers. `INTEGER` literals are written in decimal form as digits between 0 and 9 with optional `_` for visual separation. The first digit cannot be

0 unless the number is 0 itself.



The values are represented as 64-bit signed two's-complement integers resulting in a value range from -9_223_372_036_854_775_808 to 9_223_372_036_854_775_807 (inclusive).

KNIME Analytics Platform types [Number (Integer), IntType] and [Number (Long Integer), LongType]¹ are mapped to INTEGER in the expression language without loss of precision. The output of an expression that returns the INTEGER expression type will be of the column type Number (Long Integer). For flow variables, there is a dropdown menu in the output section of each expression editor where you can select the desired type, i.e., IntType or LongType.

FLOAT

The value type FLOAT is used for numbers with fractional parts.

A FLOAT number is written with a decimal point. The decimal point can be at any position in the number, even at the beginning or end, like 0.123 or .123 or 123. You can use underscores _ to separate digits to make large numbers easier to read, like 1_000.567_890 is the same as 1000.567890

You can write FLOAT numbers using scientific notation, which is useful for very large or very small numbers. In scientific notation, e or E is used to denote "times ten to the power of". You can also use a plus + or minus - sign after e or E to indicate positive or negative exponents, so that 1.23e4 or 1.23E+4 means $1.23 * 10^{(+4)}$ or 12300



The syntax for FLOAT literals is similar to the syntax used in [in the Python programming language](#). The values are represented as double-precision floating point numbers (64bit IEEE 754) This results in a value range from 4.9E-324 to 1.8E+308 (inclusive) and a precision of about 15 decimal digits.

FLOAT in the expression language maps to the [Number (Float), DoubleType]¹ column type in KNIME Analytics Platform, and has the same precision.

STRING

The value type STRING is used for sequences of Unicode characters (text). The values are

represented as a sequence of characters enclosed in double quotes "text" or single quotes 'text'.



New lines in strings are permitted, so the string can span multiple lines without using a special character.

```
"multi-line
string"
->
multi-line
string
```

Escape sequences

The backslash \ can be used for escape sequences. A backslash that does not match one of the following escape sequences is an invalid syntax.

Table 1. Escape sequences

Escape sequence	Description	Example
\<newline>	Backslash and new line in input text ignored	"xyz \ abc" → xyz abc
\\	Escaping the backslash itself	"\\something\\" → \something\
\'	Escaping single quotes	"\'quoted text\'" → 'quoted text'
\"	Escaping double quotes	"\"quoted text\"" → "quoted text"
\b	ASCII backspace causes the cursor to move backwards across the previous character	"Hello, W\bWorld!" → Hello, World!
\r	ASCII carriage return causes the cursor to move to the beginning of the line	"Hello,\rWorld!" → World!

Escape sequence	Description	Example
<code>\n</code>	ASCII linefeed causes the cursor to move to the next line. Note that on unix-like systems, this is the only character used for newlines and on Windows systems, it is used in combination with <code>\r</code>	"Hello,\nWorld!" → Hello, World!
<code>\t</code>	ASCII horizontal tab causes the cursor to move to the next tab stop	"Hello,\tWorld!" → Hello, World!
<code>\uxxxx</code>	Unicode characters can be used as escape sequences. The <code>xxxx</code> part is a 16-bit hex value	"\u0041" → A "\u00E4" → ä "\u2328" → ☨

Escape sequences are replaced from left to right and the resulting character of an escape sequence cannot be part of another escape sequence.

`STRING` in the expression language maps to `[String, StringType]`¹ in KNIME Analytics Platform.

Date-time types

The KNIME Expression Language supports several different date and time types, collectively referred to as 'date-time types'. The supported types are:

- `LOCAL_DATE`, for dates without time information,
- `LOCAL_TIME`, for times without date information and without time zone information,
- `LOCAL_DATE_TIME`, for combined dates and times without time zone information,
- `ZONED_DATE_TIME`, for combined dates and times with time zone information.

These types cannot be written as literals, but columns of the corresponding types can be used in the expression language, and various functions and operations are available that can create and manipulate them.

Duration types

The KNIME Expression Language supports duration types. These represent amounts of time, like the difference between two dates or two times. The supported types are:

- `DATE_DURATION`, for intervals between dates, which map to the KNIME Analytics Platform's date-based duration,
- `TIME_DURATION`, for intervals between times, which map to the KNIME Analytics Platform's time-based duration.

Much like the date and time types, these cannot be written as literals, but columns of the corresponding types can be used in the expression language, and various functions and operations are available that can create and manipulate them.

MISSING

The value type `MISSING` is used for missing values. It is used to represent the absence of a value in a cell or row, either because the value was missing in the input data or because the value could not be computed.

All types above except for `MISSING` can be extended to allow missing values during evaluation. This is denoted by `<TYPE> | MISSING`, so a column of type `INTEGER | MISSING` can contain both `INTEGER` values and `MISSING` values.

Literal MISSING

The literal for a missing value is `MISSING`. It is case-sensitive and must be written in uppercase. The literal, i.e., the explicit use of `MISSING` in an expression is not interchangeable with the optional type. So, while `some_function($"column with only MISSING values")` is valid, `some_function(MISSING)` is not and will result in a syntax error. For an expression that just returns `MISSING` without any further operation also a syntax error will be raised as the type of the expression would not be defined.

Input data access

Row access

To retrieve the value from a column in the current row, two syntax options are available:

Use `$["column name"]` for any column name, including those with spaces or special characters. The column name reference between the square brackets follows to rules of `STRING` literals.

For column names consisting solely of letters, numbers, and underscores (without starting with a number), a shorthand syntax `$column_name` is allowed.

Column names are case-sensitive.

- `$["Customer ID"]` Value of the column "Customer ID"
- `$["Column with a \"double\" quote"]` Value of the column 'Column with a "double" quote'
- `$customer_id` Value of the column "customer_id"

There are also special identifiers to access the

- `$[ROW_NUMBER]` to get the current row number, starting at 1.
- `$[ROW_INDEX]` to get the current row index, starting at 0.
- `$[ROW_ID]` to get the RowID, such as "Row99".



The row number, row index, and row ID are not column names and therefore are not enclosed in quotes. Shorthand syntax is not allowed for these special identifiers.

Row offsets

Sometimes it is necessary to access values from other rows in the table to perform calculations. The KNIME Expression Language allows the use of `$["column_name", offset]` to reference previous or next rows relative to the current one.

The offset is a static number and **must not** be an expression itself.

Negative offsets point to previous rows, positive offsets to rows next the current row. Replacing a column will only take effect after evaluating the expression for the whole table.

This means that the expression only uses the original data from that column.

- `$["column_name", -1]` Value of the column "column_name" from the previous row



Using an offset will necessarily access values from rows that does not exist. In this case, the result will be `MISSING`.

Flow Variable access

Flow variables are accessed using syntax similar to row access, but with two dollar signs:

Use `$$["flow var name"]` for any flow variable name.

For flow variable names consisting only of letters, numbers, and underscores (without starting with a number), a shorthand syntax `$$flow_var_name` is permitted as for column names.

Flow variable names are case-sensitive.

Operators

The KNIME Expression Language supports a variety of operators for arithmetic, comparison, and logical operations. The following sections describe the operators available in the language and their respective rules and behaviors.

Comments

Text following a `#` symbol is considered a comment and is ignored by the interpreter. Comments can be used to annotate code for clarity. Comments can be placed on a separate line or at the end of a line of code.

```
# This is a comment
1 + 2 # This is another comment but "1 + 2" is the expression
```

Arithmetic

The following table lists the arithmetic operators available in the KNIME Expression Language, along with their descriptions and typing notes. Arithmetic operations apply also to optional types. If one or both of the operands is missing, the result is missing. For clarity, we omit the optional type in the following table.

Table 2. Arithmetic operators

Name	Operator	Description	Typing notes
Addition	+	Yields the sum of two numbers.	Applicable to INTEGER and FLOAT.
Subtraction	-	Yields the difference of two numbers. Can also be used as a unary operator to negate the operand.	Applicable to INTEGER and FLOAT.
Multiplication	*	Yields the product of two numbers.	Applicable to INTEGER and FLOAT.

Name	Operator	Description	Typing notes
Division	/	Yields the quotient of two numbers.	Applies to INTEGER and FLOAT. The result is always of type FLOAT.
Floor Division	//	Yields the quotient of two numbers floored to the next INTEGER number.	Only applicable to INTEGER values.
Exponentiation	**	Yields the power of two numbers.	Applicable to INTEGER and FLOAT.
Remainder	%	Yields the remainder from the division of the first argument by the second.	Applicable to INTEGER and FLOAT.

If both operands are of the same type, the result is of the same type if not specified otherwise for the specific operator. If one or both of the operand types is optional, the result is optional. If the operands are of type INTEGER and FLOAT (order irrelevant), the INTEGER value is converted to the closest value of type FLOAT, and the result is of type FLOAT.

Division by zero

Dividing a number by zero with the division, floor division or remainder operator yields a runtime warning. The output of the operation is defined via the following rules.

Table 3. Division by zero ruleset

Name	Operator	Condition	Output
Division	/	The first operand is 0	0. / 0 → NaN
		Both operands have the same sign	1. / 0 → INFINITY
		The operands have different signs	-1. / 0 → -INFINITY

Name	Operator	Condition	Output
Floor Division	//	-	0 Floor division returns always an INTEGER.
Remainder	%	The first operand is of type FLOAT	NaN
		Both of the operands are of type INTEGER	0

The arithmetic operators are also applicable to date-time and interval types. The following table lists the arithmetic operators available for date-time and interval types.

Table 4. Arithmetic operators for date-time and interval types

Name	Description
Addition (+)	<p>Can be applied to:</p> <ul style="list-style-type: none"> • A date-time with date information + DATE_DURATION, to yield a new date-time of the same type with the duration added to the date (which will usually leave the time information unchanged, except in cases where this is not possible due to e.g. daylight saving times). Note that the date-time must be the first operand. • A date-time with time information + TIME_DURATION, to yield a new date-time of the same type with the duration added to the time (which may alter the date information). Note that the date-time must be the first operand. • A DATE_DURATION with a DATE_DURATION or a TIME_DURATION with a TIME_DURATION to obtain an interval as long as the two operands combined.

Name	Description
Subtraction (-)	<p>Can be applied to:</p> <ul style="list-style-type: none">• A date-time with date information - <code>DATE_DURATION</code>, to yield a new date-time of the same type with the duration subtracted from the date (which will usually leave the time information unchanged, except in cases where this is not possible due to e.g. daylight saving times). Note that the date-time must be the first operand.• A date-time with time information - <code>TIME_DURATION</code>, to yield a new date-time of the same type with the duration subtracted from the time (which may alter the date information). Note that the date-time must be the first operand.• A <code>DATE_DURATION</code> with a <code>DATE_DURATION</code> or a <code>TIME_DURATION</code> with a <code>TIME_DURATION</code> to obtain an interval equal to the difference of the two operands.• Two <code>LOCAL_DATE</code> values, to obtain a <code>DATE_DURATION</code> equal to the difference of the two dates.• Any two <code>LOCAL_TIME</code>, any two <code>LOCAL_DATE_TIME</code>, or any two <code>ZONED_DATE_TIME</code> values, to obtain a <code>TIME_DURATION</code> equal to the difference of the two date-time values.

Name	Description
Multiplication (*)	<p>Can be applied to:</p> <ul style="list-style-type: none">• a <code>TIME_DURATION</code> with an <code>INTEGER</code> to obtain <code>TIME_DURATION</code> value that has the same length as the original duration multiplied by the number.• a <code>DATE_DURATION</code> with an <code>INTEGER</code> to obtain <code>DATE_DURATION</code> value that has the same length as the original duration multiplied by the number.

Comparison

Comparison operators are used to compare two numeric values (`FLOAT` and `INTEGER`) and yield a `BOOLEAN` result. The comparison operators are also applicable to date-time and interval types, with the exception of the `DATE_DURATION` type, which does not support any kind of ordering. There are two kinds of comparison operators: ordering and equality.

Comparisons never return optional results. This ensures that optionals are not propagated through comparison. Therefore, it is less likely that the result of an expression is optional.

General rules for comparison operators

Comparison operators compare numeric, interval, and date-time types, while comparing to `MISSING` always returns `FALSE`. Equality operators work on all types as long as both operands are of the same type or `MISSING`, with the exceptions that `INTEGER` and `FLOAT` can be checked for equality with one another, and `ZONED_DATE_TIMES` cannot be checked for equality at all.

Table 5. Comparison operators

Name	Operator	Kind	Notes
Less Than	<	Ordering	

Name	Operator	Kind	Notes
Less Than or Equal To	<=	Ordering	Note that MISSING <= MISSING is TRUE
Greater Than	>	Ordering	
Greater Than or Equal To	>=	Ordering	Note that MISSING >= MISSING is TRUE
Equal	= or ==	Equality	
Not Equal	!= or <>	Equality	Same as not (a == b)

When considering date-time and duration types, the comparison operators work as follows:

- < and > compare the date-time values in chronological order. A later date-time is considered greater than an earlier date-time. A longer duration is considered greater than a shorter duration. Note that DATE_DURATION is not ordered.
- <= and >= as with > and <, but including equality.
- = and != compare the date-time values for equality. Two date-time or two duration values are considered equal if they have the same type, and respectively represent the same point in time or the same duration. Note that these operators are not applicable to ZONED_DATE_TIME values.

Logical operators

The logical operators `and`, `or`, and `not` apply to `BOOLEAN` types as well as to their optional `BOOLEAN | MISSING` types. `and` and `or` are binary operators while `not` is a unary operator.

If both of the operands are of type `BOOLEAN`, the result is of type `BOOLEAN`. If the type of one or both of the operands is optional, i.e., `BOOLEAN | MISSING`, missing values are interpreted as unknown according to [Kleene's three-valued logic](#).

Table 6. Logical operators

Name	Operator	Description	Examples
Logical AND	and	Yields TRUE if both operands are TRUE. Yields FALSE if at least one operand is FALSE and MISSING otherwise .	TRUE and FALSE → FALSE TRUE and MISSING → MISSING FALSE and MISSING → FALSE
Logical OR	or	Yields FALSE if both operands are FALSE. Yields TRUE if at least one operand is TRUE, otherwise MISSING.	TRUE or FALSE → TRUE TRUE or MISSING → TRUE MISSING or FALSE → MISSING
Logical NOT	not	Yields TRUE if the operand is FALSE. Yields FALSE if the operand is TRUE and MISSING if the operand is MISSING.	not TRUE → FALSE not FALSE → TRUE not MISSING → MISSING

String concatenation

The operator `+` can also be used for string concatenation if at least one of the operands is of type `STRING` or `STRING | MISSING`.



A literal `MISSING` is not a supported type and will result in a syntax error. The output type of a string concatenation is always a `STRING`. Missing values in the input data are mapped to the string `"MISSING"`.

```

"Hello" + " " + "World"      -> "Hello World"
"Hello" + 42                 -> "Hello42"
"Hello" + $["column with missing value"] -> "HelloMISSING"
"Hello" + MISSING            -> Syntax error

```

Missing coalescing operator

The missing coalescing operator `??` is a binary operator that returns the left operand if it is not `MISSING`, otherwise it returns the right operand. Both operands must have the same type and the result is of the same type. If both operands are `MISSING` values, the result is `MISSING`. Even though it will rarely be useful, you can pass `MISSING` as one of the arguments to `??`. However, `MISSING ?? MISSING` is treated as syntax error.

```
1 ?? 2          -> 1
MISSING ?? 2    -> 2
MISSING ?? MISSING -> Syntax error
```

Operator precedence

Operator precedence defines the order in which operations are evaluated in an expression, when it contains more than one operator in series. You can always use parentheses `(,)` to enforce a specific order of evaluation. The following table lists the operators in order of precedence, from highest to lowest.

1. Missing Coalescing (`??`)
2. Exponentiation (`**`)
3. Negation (unary `-`)
4. Multiplication (`*`), Division (`/`), Remainder (`%`), Integer Division (`//`)
5. Addition (`+`), Subtraction (`-`)
6. Comparison operators (`<`, `<=`, `>`, `>=`, `!=`, `<>`, `=`, `==`)
7. Logical NOT (`not`)
8. Logical AND (`and`)
9. Logical OR (`or`)

Operations with higher precedence are evaluated before those with lower precedence. Operations with the same precedence level are evaluated from left to right except for `**` which is evaluated from right to left. In the following we give some examples to illustrate the precedence of the operators.

Table 7. Operator precedence examples

Expression	With Parenthesis	Result	Explanation
$1 + 2 * 3$	$1 + (2 * 3)$	7	multiplication is evaluated first
$1 + 2 ** 3 * 4$	$1 + ((2 ** 3) * 4)$	33	Exponentiation is evaluated before multiplication and multiplication is evaluated before addition
$2 * 2 ** 3 ** 2$	$2 * (2 ** (3 ** 2))$	1024	Exponentiation is evaluated first and from right to left
TRUE or FALSE and FALSE	TRUE or (FALSE and FALSE)	TRUE	and is evaluated first

Functions

There are two types of functions that are usable in an expression that will be distinguished in the following sections: (1) **Row-wise functions** that apply row-wise to generate a new value from one or multiple values of each input row and (2) **Aggregation functions** that apply to a whole column to generate a single value that can be used for each evaluated row.

There is a function catalog available in the editor to help with the selection of functions and their arguments by providing detailed descriptions and examples. You will find built-in **Constants** there, too.

Row-wise functions

Functions evaluated row-wise are always lower-case and calls are made using the function name followed by parentheses containing any arguments:

```
function_name(arg1, arg2, ...)
```

Each function has a specific number of arguments and types that it expects. If the arguments do not match the expected types, a type error is raised. The return type of a function is determined by the function itself and is not necessarily the same as the input types.



Every function returns some value and there are no `void` functions. Functions can be nested, i.e., a function call can be an argument to another function.

If there are multiple arguments, they must be separated by commas. Each argument can be any valid expression. You may optionally include a trailing comma after the last argument.

Examples:

```
sqrt(4)                -> 2
pow(abs(-sqrt(3.14**2)),2) -> 3.14
if(TRUE, "true branch","false branch") -> "true branch"
```

Aggregation functions

Aggregation Functions are a special set of functions prefixed with `COLUMN_` that calculate aggregations over whole columns, such as their minimum, maximum, or mean values, for example, `COLUMN_MIN("Column Name")`.



The aggregation functions take a string literal "Column name" instead of a value from a row (`$["column name"]` or `$column_name`) as input.

In aggregation functions we offer to provide arguments positionally and by name of the argument. Positional arguments are always first, followed by named arguments. Named arguments are always provided as `arg_name=value`.

Let's illustrate that for the aggregation function `COLUMN_AVERAGE(column, ignore_nan)`

- Only positional arguments: `COLUMN_AVERAGE("Column Name", TRUE)`
- Only named arguments: `COLUMN_AVERAGE(column="Column Name", ignore_nan=TRUE)`
- Mixed arguments: `COLUMN_AVERAGE("Column Name", ignore_nan=TRUE,)`

Constants

The KNIME Expression Language provides a set of predefined constants that can be used in expressions. These constants are used to represent common mathematical values and special values. The following constants are predefined and can be used in expressions:

Table 8. Constants

Name	Symbol	Type	Description
Truth value	TRUE	BOOLEAN	The Boolean <code>true</code> .
False value	FALSE	BOOLEAN	The Boolean <code>false</code> .
Euler's number <code>e</code>	E	FLOAT	Euler's number, ~2.71828, used as the base of natural logarithms and in exponential functions.
Pi or π	PI	FLOAT	The constant Pi, ~3.14159, the ratio of a circle's circumference to its diameter.
Positive Infinity	INFINITY	FLOAT	A special constant representing positive infinity.
Not a Number	NaN	FLOAT	A special constant representing "Not a Number".
Smallest positive float	TINY_FLOAT	FLOAT	The smallest positive float value representable by this computer.
Largest positive float	MAX_FLOAT	FLOAT	The largest positive value that can be represented as a FLOAT.

Name	Symbol	Type	Description
Smallest negative float	MIN_FLOAT	FLOAT	The smallest negative value that can be represented as a FLOAT.
Largest positive integer	MAX_INTEGER	INTEGER	The largest positive value that can be represented as an INTEGER.
Smallest negative integer	MIN_INTEGER	INTEGER	The smallest negative value that can be represented as an INTEGER.
Missing value	MISSING	MISSING	A special constant representing a missing value.

KNIME AG
Talacker 50
8001 Zurich, Switzerland
www.knime.com
info@knime.com